Research Summary

Joseph W. Wimmergren

Fall 2021

1 Paracrine Factor Diffusion

Paracrine signalling occurs when a neurotransmitter or paracrine factor leaks from the synaptic cleft and has some downstream effect on other neurons. The function of the paracrine diffusion model is to determine the concentration of a paracrine factor at irregularly scattered neuron locations. Modeling the various interactions and dynamics of paracrine signalling in the SCN could answer many interesting questions. In particular, we can investigate how paracrine signals play a role in the modulation of sleep and wakefulness. We consider the 20,000 neuron mouse SCN for our model host, as the SCN is known to act as a central "pacemaker" of circadian timing systems.

2 Paracrine Diffusion Model

We assume the paracrine factor is released into the extracellular space at a constant rate immediately after each appropriate neuron fires (paracrine factor producing neurons). To get an estimation of the concentration of the neurotransmitter at each neuron in the SCN, we can model this process with the standard diffusion equation

$$u_t = -D\,\nabla^2 u - ku, \tag{1}$$

where u is the concentration of the paracrine factor, D is the diffusion coefficient, and k is the decay rate of the paracrine factor. Indeed, this problem exists in 3-space, so the concentration of the neurotransmitter u is a function of time and the three spatial variables, and $\nabla^2 = \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}\right)$.

2.1 Numerical Solution to the Diffusion Equation

To update the concentration at the neuron locations, the following steps are taken. The concentration of the paracrine factor is discretized onto a regular 3-dimensional spatial grid. A finite difference method is used to update the concentration of the paracrine factor on this regular grid (by numerically solving (1) on the grid), then an interpolation scheme is required to find the concentration at the various scattered neuron locations. Finally, the concentration of the neurotransmitters must be distributed back onto the regular grid to allow for the next timestep update.

2.1.1 Crank-Nicolson method

Let dt be the timestep taken, u^n the paracrine factor concentration at the nth timestep, and u^{n+1} be the paracrine factor at the $(n+1)$th timestep. The Crank-Nicolson method relates the aforementioned terms by

$$\frac{1}{dt}(u^{n+1} - u^n) = \frac{D}{2}(\nabla^S u^{n+1} + \nabla^S u^n) - \frac{k}{2}(u^{n+1} + u^n), \tag{2}$$

where ∇^S is the discrete Laplace operator. We note that the local truncation error is bounded by $O((dt)^2 + (dx)^2/D)$ where dx is the spatial stepsize. Because the concentration of the neurotransmitter at any location is a function of the concentration at the surrounding locations, we use a 27-point stencil for the discrete Laplace operator. If our point of interest is in a regularized spatial grid, the closest available points to it are on the "cube" surrounding it, i.e. a point on each edge, vertex, and face surrounding the point of interest. This is visualized by figure 1 below, taken from [2].

The Crank-Nicolson method is an implicit method (in (2), we see that u^{n+1} exists on both sides of the equation). As such, solving for u^{n+1} means solving a linear equation in the form of $Ax = b$.

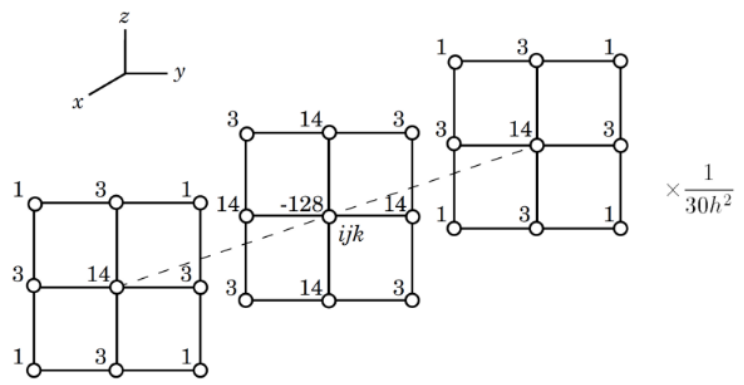

Figure 1: 27-point high order compact stencil for the discrete Laplace operator from [2]

2.1.2 Getting (2) to $Ax = b$

To solve for u^{n+1}, we must first get (2) in the form of $Au^{n+1} = b$. As such,

$$\begin{aligned}
&\frac{1}{dt}(u^{n+1} - u^n) = \frac{D}{2}(\nabla^S u^{n+1} + \nabla^S u^n) - \frac{k}{2}(u^{n+1} + u^n) \\
\iff &\frac{u^{n+1}}{dt} - \frac{u^n}{dt} = \frac{D\nabla^S u^{n+1}}{2} + \frac{D\nabla^S u^n}{2} - \frac{ku^{n+1}}{2} - \frac{ku^n}{2} \\
\iff &\frac{u^{n+1}}{dt} - \frac{D\nabla^S u^{n+1}}{2} + \frac{ku^{n+1}}{2} = \frac{u^n}{dt} + \frac{D\nabla^S u^n}{2} - \frac{ku^n}{2} \\
\iff &\left(\frac{I}{dt} - \frac{D\nabla^S}{2} + \frac{kI}{2}\right) u^{n+1} = \left(\frac{I}{dt} + \frac{D\nabla^S}{2} - \frac{kI}{2}\right) u^n \\
\iff &\left(I - 0.5D\nabla^S \, dt + 0.5 \, dt \, k \, I\right) u^{n+1} = \left(I + 0.5D\nabla^S \, dt - 0.5 \, dt \, k \, I\right) u^n.
\end{aligned}$$

Then

$$\begin{aligned}
A &= -0.5D\nabla^S \, dt + (1 + 0.5 \, dt \, k)I \\
b &= \left(I + 0.5D\nabla^S \, dt - 0.5 \, dt \, k \, I\right) u^n
\end{aligned}$$

where u^n is known. We can now solve this linear system.

2.1.3 Solving $Au^{n+1} = b$

To avoid the expensive computational cost of matrix inversion, we seek other methods of solving the problem. We first note that A is symmetric ($A^T = A$) and positive definite ($x^T Ax > 0,\ \forall\, 0 \neq x \in \mathbb{R}^n$). As such, a natural solution method to choose is the conjugate gradient descent method. Let $u^{n+1} := x$.

Two non-zero vectors u and v in $\mathbb{R}^n$ are conjugate with respect to matrix A if $u^T Av = 0$. Now suppose $P = \{p_1, p_2, ..., p_n\}$ is a set of n many mutually conjugate vectors with respect to matrix A, i.e. $p_i^T Ap_j = 0\ \forall\ i \neq j$, then certainly P forms a basis for $\mathbb{R}^n$, i.e.

$$\begin{aligned} x &= \sum_{i=1}^{n} \alpha_i p_i \\ \iff Ax &= \sum_{i=1}^{n} \alpha_i Ap_i \\ \iff p_{n+1}^T Au^{n+1} &= \sum_{i=1}^{n} \alpha_i p_{n+1}^T Ap_i \\ \iff p_{n+1}^T b &= \sum_{i=1}^{n} \alpha_i p_{n+1}^T Ap_i \\ \iff p_{n+1}^T b &= \alpha_{n+1} p_{n+1}^T Ap_{n+1}. \end{aligned}$$

Then we can determine coefficient α_k by

$$\alpha_k = \frac{p_k^T b}{p_k^T Ap_k}.$$

To generalize this process, we find a sequence of n conjugate directions and compute coefficient α_k.

As an iterative method, we start with an initial guess x_0 and set $p_0 = b - Ax_0$, where p_0 is our first search direction. We can also let our first residual be the same, i.e. $r_0 = p_0$. Now the other vectors of P must be conjugate to $b - Ax_0$. Our general formula for the residual at the kth step is

$$r_k = b - Ax_k.$$

Of course, if this residual is close enough to 0, then x_k is our solution to $Ax = b$. I used an error tolerance of 10^{-10}, in which if $r <$ (error tolerance) then leave the iteration and x_k is the solution. If $r >$ (error tolerance) then continue.

We need the search directions p_k to be conjugate to each other with respect to matrix A. We do this by requiring that the next search direction be built from the current residual **and** all previous search directions. We follow the Graham-Schmidt reminiscent sums to do this, i.e. the next search direction is

$$p_k = r_k - \sum_{i<k} \frac{p_i^T Ar_k}{p_i^T Ap_i} p_i.$$

Then our next guess is

$$x_{k+1} = x_k + \alpha_k p_k$$

with

$$\alpha_k = \frac{p_k^T (b - Ax_k)}{p_k^T Ap_k} = \frac{p_k^T r}{p_k^T Ap_k}.$$

After the iteration is done sufficiently many times, we have our explicit solution for u^{n+1}.

2.1.4 Interpolation

Now that we have the paracrine factor concentration at the regular gridpoints, we must now determine what the concentration of the paracrine factor is at the irregular neuron locations. We use the trilinear interpolation method from [5]. This is a convolution type interpolation method with convolution kernel

$$K(x) = \begin{cases} \frac{3}{2}|x|^3 - \frac{5}{2}|x|^2 + 1, & 0 < |x| < 1 \\ \frac{-1}{2}|x|^3 + \frac{5}{2}|x|^2 - 4|x| + 2, & 1 < |x| < 2 \\ 0, & \text{else} \end{cases}$$

Then if x_n is the gridpoint, and $u_n = f(x_n)$ is known, then the interpolated value $u(x)$ is given by

$$u = f * \sum_n u_n K(x - x_n). \tag{3}$$

2.1.5 Spreading

Now that the concentration of the paracrine factor is known at each neuron, we must spread the paracrine factor from neuron locations back to the regular grid point locations. This is done in a similar way as the interpolation, but inverted. Indeed, we can use the transpose of the interpolation matrix obtained from 2.1.4 and multiply our concentration vector by this transposed matrix. More work has to be done to determine if this is the most effective way to spread the paracrine factor.

---

3 Rewriting Paracrine Code

Many issues in the current code were present as a result of using the external library ArrayFire (AF). The inclusion of AF and the exclusion of friendly comments led to an unfortunately large amount of time spent trying to get the code to work on other desktops, even with many hours of time with the original writer. Finally, it was determined that by rewriting this paracrine diffusion code in native CUDA, we will be able to facilitate the peaceful transfer of knowledge and code between students, faculty, and collaborators. The glorious pursuit of knowledge shall not be hindered by a library.

3.1 1-D Heat Equation

To ensure that I have the programming skills and various numerical method mechanics correctly understood, I first solve the 1-D heat equation on a finite rod with Neumann boundary conditions,

$$\begin{aligned} u_t = 5u_{xx},\ 0 < t < 5,\ 0 < x < 5 \\ u_x(0,t) = u_x(L,t) = 0 \\ u(x,0) = 4 \end{aligned}$$

3.1.1 Crank-Nicolson Method

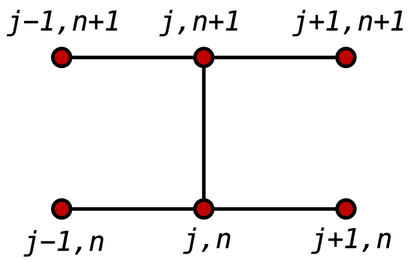

Figure 2: Stencil used for the discrete laplace operator

As seen by Figure 2, the Crank-Nicolson Method for this problem relates u^{n+1}, u^n, dt, and dx in the following way

$$ru^n_{i+1} + (1 + 2r)u^{n+1}_i - ru^{n+1}_{i-1} = ru^n_{i+1} + (1 - 2r)u^n_i + ru^n_{i-1},$$

where

$$r = \frac{a\,dt}{2(dx)^2}.$$

With the Neumann boundary conditions, at every timestep dt, and if there are J many spatial steps dx, then we have

$$\begin{pmatrix} (1+r) & -r & 0 & 0 & ... \\ -r & (1+2r) & -r & 0 & ... \\ 0 & -r & (1+2r) & 0 & ... \\ . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & -r \\ . & . & . & -r & (1+r) \end{pmatrix} \begin{pmatrix} u(0,n+1) \\ u(1,n+1) \\ u(2,n+1) \\ . \\ . \\ . \\ u(J-1,n+1) \end{pmatrix}$$
$$= \begin{pmatrix} (1-r) & r & 0 & 0 & ... \\ r & (1-2r) & r & 0 & ... \\ 0 & r & (1-2r) & 0 & ... \\ . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & r \\ . & . & . & r & (1-r) \end{pmatrix} \begin{pmatrix} u(0,n) \\ u(1,n) \\ u(2,n) \\ . \\ . \\ . \\ u(J-1,n) \end{pmatrix}$$

i.e.

$$AU^{n+1} = BU^n.$$

As U^n is given, through matrix-vector multiplication, we get

$$AU^{n+1} = b.$$

By the conjugate gradient descent method discussed in 2.1.3, we easily solve for U^{n+1}. This matrix-vector multiplication and conjugate gradient descent method have to be done at every timestep. Figure 3 visualizes the solution to the 1-D heat equation.

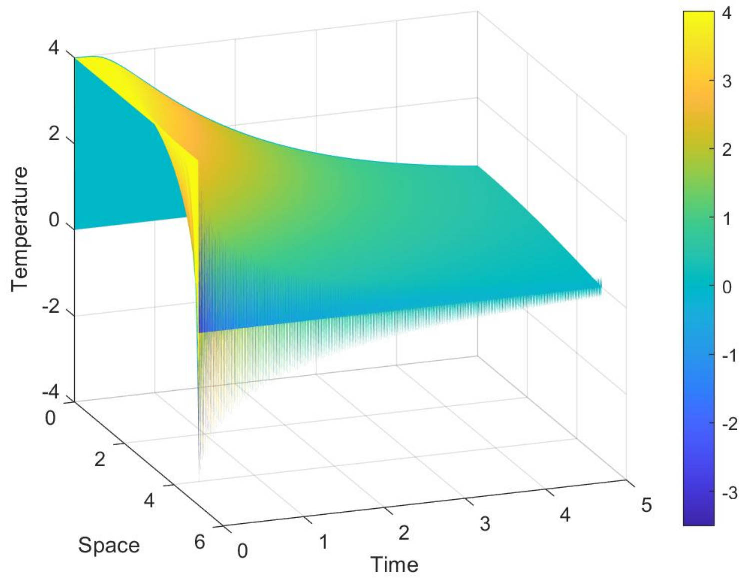

Figure 3: Numerical solution to the 1D heat equation with Neumann boundary conditions. Computation done in C++, plotting done in MATLAB.

On a CPU this is quite computationally intensive. If only there was a way to speed this up...

3.2 A Digression on Matrix Multiplication on the GPU

Fortunately for us, crypto and the gaming industry accelerated GPU technology over the years. By taking advantage of the parallel nature of matrix multiplication and the inherent parallel structure of GPUs, we can significantly speed up the above calculations. In this subsection, I briefly describe how matrix multiplication works on the GPU.A quick and relatively hand-wavy breakdown of the GPU architecture is as follows.

Threads: The lowest granularity of execution. Often, you'd like each thread in your GPU to be used (fully saturated) to have the most efficient runtime. The threads themselves execute the instructions.

Warps: Threads are composed into warps, the lowest schedulable entity. Warps execute instructions in lock-step, i.e. schedules all threads in the warp to execute an instruction. You may, however, block off some threads within the warp, if there are too many threads for the individual problem.

Thread Blocks: made of warps. Thread blocks are the lowest programmable entity. The threadblocks themselves are assigned to cores and the size of the threadblock (i.e. how many threads per threadblock) is a parameter the user can define. By doing so, the warps are automatically created to fit the number of threads per threadblock.

Grids: How a program is mapped to the GPU. Similar to Thread Block size, grid size is a parameter defined by the user. A grid is made up of thread blocks. As such, the user defines how many thread blocks there are in each dimension of the grid.

3.2.1 $AB = C$

Let A be an m by n matrix, and B be an n by p matrix. Then C is an m by p matrix. The standard method of numerical matrix multiplication is through the sequential process of nested loops, individually traversing and summing the products of the rows of A and columns of B. However, the GPU allows us to execute single instruction multiple thread commands (SIMT), instead of the classical sequential instruction execution.

For matrix multiplication, we would like each thread to be responsible for determining the value of each element of our matrix C. In this way, we can compute C with a single for loop (instead of two), effectively calculating the inner loop with a single instruction. As such, each thread should traverse 1 row of A and 1 row of B, multiplying and summing as it goes, and writing the result to C. We can think of each thread having a unique pair of rows and columns "linked" to it.

A key component of GPU architecture is how matrices are stored. As we are using CUDA to "speak" with the computer, we must follow the rules of matrix memory allocation in CUDA. Generally, matrices are stored in one of two layouts, row-major or column-major. The row-major layout is easily explained in the following visual:

$$\begin{pmatrix} M(0,0) & M(0,1) & M(0,2) & ... \\ M(1,0) & M(1,1) & M(1,2) & ... \\ M(2,0) & M(2,1) & M(2,2) & ... \end{pmatrix} \text{ is the matrix to be stored}$$

$$\begin{pmatrix} M(0,0) & M(0,1) & M(0,2) & ... & M(1,0) & M(1,1) & M(1,2) & ... \end{pmatrix} \text{ the same matrix in memory}$$

and column-major format is the opposite (columns are listed one after another instead of rows). So we note that a 2D matrix is stored as a 1D array in memory. As such, we must be clever in how we address the elements of a matrix and how it is stored in memory. In CUDA, row-major format is used.

Take element (x,y) of matrix M, for example. This element can be addressed with $x\,\text{width} + y$. For a 4x4 matrix, the element (1,1) will be found at position $1 \cdot 4 + 1 = 5$ in the 1D array of memory. In otherwords, M(1,1)=memory(5).

For our matrix multiplication problem, we can map each data element to a thread. The following mapping scheme is used to do this:
row=blockIdx.y · blockDim.y · +threadIdx.y
column=blockIdx.x · blockDim.x · +threadIdx.x
where blockIdx.a is the block we are looking at in the a-direction, blockDim.a is the length of the block in the a-direction, and threadIdx.a is the thread of interest in the a-direction.

It seems that the most difficult part of GPU programming is determining how to allocate and assign threads for each problem. We start with the simple example of a 1024 x 1024 square matrix. Our goal is to have each thread in charge of computing their respective element in our new matrix C. Seeing as multiplying two 1024 x 1024 matrices yields another 1024 x 1024 matrix, we must use 1024 threads for optimum efficiency. The question is, then, how we layout the grid on our GPU. We can follow our intution and use a square sized grid, as our matrices are square. We are then left to decide the dimensions of our square grid, and the dimensions of the thread blocks. An easy breakdown could be a gridsize of 64 blocks by 64 blocks, which leads to 4096 thread blocks. We would like the total number of threads to match the number of elements in our matric C, so a natural assignment would be

$$\frac{\text{Number of elements in C}}{\text{Number of blocks in grid}} = \text{Threads per block.}$$

By following this rule, we can assign threads per block to be 256. However, we cannot just specify total number of threads, we must specify the dimensions of the blocks themselves. Once more, the natural selection of a square block size is reasonable, i.e. a 16 x 16 thread block size. To check our assignment, we see that $64 \cdot 16 \cdot 64 \cdot 16 = 1024$, so our thread allocation was done in an efficient way. Now we can compare the standard matrix multiplication nested loop with an example matrix multiplication loop from a CUDA kernel.

Sequential

```
for (int i=0; i<1024; i++){

    for (int j=0; j<1024; j++){

    tempsum=0;

        for (int k=0; k<1024; k++){

          tempsum+=A[i][k]*B[k][j];

         }

    C[i][j]=tempsum;

     }

}
```

SIMT

```
tempsum=0;
for (int i=0; i¡1024; i++){

    tempsum += A[row][i]*B[i][column];

    }
C[row][column]=tempsum;
```

4 Plans

This 1D model must be scaled up to the 3D diffusion equation in the mouse SCN. To do this, I will build up from this 1D heat equation in steps, to ensure correctness. Clarity will be a large focus in how I write code, and I will continue refining my C++ and CUDA toolkit, as well as learning about more applications of this paracrine model.

This semester was a lot of learning and troubleshooting. As this paracrine model gets rewritten, the next step becomes using it in tandem with the whole brain mouse code. This is certainly on the horizon and I look forward to continuing full spead ahead.

5 References

[1] Lloyd N. Trefethen and David Bau. 1997. Numerical Linear Algebra, III, xii+361 SIAM

[2] Spotz, William and Carey, G. (1996). High-Order Compact Finite Difference Methods.

[3] David L. Chopp. 2019. Introduction to High Performance Scientific Computing (1st. ed.). SIAM.

[4] Ningyuan Wang's Thesis Proposal, 2017

[5] Robert Keys. Cubic convolution interpolation for digital image processing. IEEE 29(6):11531160, 1981.