Hodgkin–Huxley Conductance Based Model: from 1D to 3D

James Rosado

Abstract

The Hodgkin-Huxley conductance based model is a mathematical model that describes the initiation and propagation of a neuron action potential. For this project we model the Hodgkin-Huxley formalism for a neuron action potential, along a one dimensional domain and then on a Y-branch domain, using different numerical schemes. In particular we will utilize a numerical scheme that is efficient when implemented in the Unity VR© system on a cylinder domain and 3D Y-branch domain.

Keywords: numerical partial differential equations, numerical linear algebra, method of lines, operator splitting ,computational neuroscience, virtual reality

1. Introduction

This project consists of simulating the Hodgkin-Huxley conducatance based action potential model [4] on a one dimensional domain, first on a one dimensional line segment and then on a one dimensional Y-branch domain. Currently, there is the Neuron© simulation environment which is used to model individual neurons and neuronal networks. However for this project we attempt to replicate the capabilities of Neuron© and go the extra step of running live simulations through the Unity VR© environment. This ties into the second goal of this project, that is to map the 1 dimensional solution on the line segment to the volume and surface of the 3 dimensional cylinder that surrounds the 1 dimensional domain and then observe the action potential propagate through the cylindrical solid. Some the questions that need to be addressed are: what is the best numerical solver (in terms of run time and memory usage) to utilize in a live VR simluation? How do we map the one dimensional geometry to the 3 dimensional geometry in the VR framework? Are there numerical libraries readily available to utilize in the numerical solving of the partial differential equation model? The paper is organized as follows, our description of the model is given in Section 2 Mathematical Model, the numerical algorithms that are utilized and prototyped are given in Section 3 Numerical Methods, discussion of the simulation are in Section 4, an overview of the implementation and challenges in Unity VR© is given in Section 5 and the conclusions follow in Section 6.

2. Mathematical Model

For this project we numerically solve the Hodgkin —Huxley model equations along a linear one dimensional domain, e.g. a domain Ω = line segment, and a line segment that

branches to two other line segments. The model equations [1, 4, 2] are given below

$$I = C_m \frac{dV_m}{dt} + \bar{g}_K n^4 (V_m - V_k) + \bar{g}_{Na} m^3 h (V_m - V_{Na}) + \bar{g}_l (V_m - V_l) \tag{1}$$

$$\frac{dn}{dt} = \alpha_n(V_m)(1-n) - \beta_n(V_m) n \tag{2}$$

$$\frac{dm}{dt} = \alpha_m(V_m)(1-m) - \beta_m(V_m) m \tag{3}$$

$$\frac{dh}{dt} = \alpha_h(V_m)(1-h) - \beta_h(V_m) h, \tag{4}$$

where V_m is the neuron membrane potential, $\bar{g}_i$ are the ion conductances, V_i are the ion reversal potentials, and C_m is the membrane capacitance. The membrane is treated as a capacitor, the cell membrane is a lipid bilayer which has more positive charge in the extracellular region and negative charge in the intracellular region. The circuit diagram that corresponds to (1) is shown below

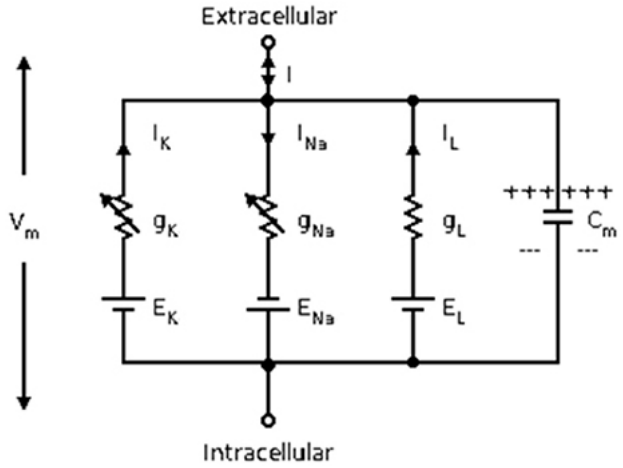


Figure 1: The conductances g_i correspond to resistors in parallel, the gate voltages E_i correspond to sources, and C_m is represented by a capacitor.

The α_i and β_i variables are ion channel rates for the voltage gate states which depend on V_m, the membrane potential. In particular the equations [4] are given by

$$\alpha_n(V) = \frac{0.01(10-V)}{\exp\left(\frac{10-V}{10}\right) - 1}, \qquad \beta_n(V) = 0.125 \exp\left(\frac{-V}{80}\right)$$

$$\alpha_m(V) = \frac{0.1(25-V)}{\exp\left(\frac{25-V}{10}\right) - 1}, \qquad \beta_m(V) = 4 \exp\left(\frac{-V}{18}\right)$$

$$\alpha_h(V) = 0.07 \exp\left(\frac{-V}{20}\right), \qquad \beta_h(V) = \frac{1}{\exp\left(\frac{30-V}{10}\right) + 1}$$

The quantities n, m, h are dimensionless quantities between 0 and 1 that are associated with the ion channel activation/inactivation. All together the variables $V_m(t), n(t), m(t), h(t)$ are the state variables of the neuron cell, they are coupled together with the PDE given by (1)-(4) which is a nonlinear system; therefore, cannot be solved analytically. These equations

model the membrane potential at a particular time, in order to model the propagation spatially through a domain Ω we let [4, 7]

$$I = \frac{a}{2R}\frac{\partial^2 V_m}{\partial x^2},$$

in (1) which is derived from the cable equation. This substitution turns our system into a partial differential equation problem, where $V_m = V_m(x,t)$, a is the axon radius, R is the resistance of the axoplasm. Initially, we will consider solving this problem with Dirichlet boundary conditions, i.e. $V_m(x,0) = -55\ mV$, $V_m(0,t) = V_m(L,t) = -55\ mV$, imposing a voltage clamp at the ends of a one dimensional segment.

3. Numerical Methods

The numerical methods that were prototyped (in Matlab©) for this project are Forward Euler for time differentiation in (1-4), and center finite differences to approximate the second order derivative in (1). For this project, I incorporated FE-CD in two schemes: the Method of Lines scheme and the Operator Splitting scheme. Later we will demonstrate the Operator Splitting Method will require fewer time steps when compared to the standard MOL with FE-CD. Stiffness is anticipated since we are considering a compartmentalized model [5], that is the axon is subdivided into components of size length h. We may encounter stiffness due to the approximation of $(V_m)_{xx}$, the diffusion term of our equation and this may also be affected by the coefficients multiplied with the diffusion term.

3.1. Forward Euler and Centered Differencing

For completion, I have included this section for a brief overview of the fundamental numerical solve calculations (FE and CD). Recall for Forward Euler time stepping, if we have the autonomous ODE problem

$$\frac{dv}{dt} = f(v(t)),$$

with $v_0 = v(t_0)$ then we can approximate the equation by

$$\frac{v^{n+1} - v^n}{k} = f(v^n)$$

where k is our time step size, v^{n+1} is our next approximation, and v^n is our current approximation. This can written as an explicit update rule

$$v^{n+1} = v^n + kf(V^n) \tag{5}$$

where $v^0 = v(t_0)$.

For the spatial solve, the standard centered difference [6] will be utilized to approximate the second order differential operator

$$\frac{\partial^2 v_m}{\partial x^2}(x,t) \approx D^2 v_m(x) = \frac{v_m(x-h,t) - 2v_m(x,t) + v_m(x+h,t)}{h^2},$$

or written as a stencil rule we have

$$D^2 v_m(x_j) = D^2 v_j = \frac{v_{j-1} - 2v_j + v_{j+1}}{h^2}, \tag{6}$$

I omitted the subscript m for membrane potential to avoid confusion and simplify the notation. This also assumes an equidistant discretization of the domain Ω. Since we are working with a one dimensional domain we can form an equidistant partition $\Omega = \{x_0, x_1, \ldots, x_{m-1}, x_m\}$ where $x_i - x_{i-1} = h$. The corresponding stencil matrix would be the tri-diagonal matrix

$$A = \frac{1}{h^2}\begin{pmatrix} -2 & 1 & & & & \\ 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ & & & & 1 & -2 \end{pmatrix}, \quad v = \begin{pmatrix} v_1 + h^{-2}v_{0,a} \\ v_2 \\ v_3 \\ \vdots \\ v_{m-2} \\ v_{m-1} + h^{-2}v_{0,b} \end{pmatrix}, \tag{7}$$

where V is the approximation to the solution vector, taking into account the boundary conditions. As a side remark, I will define

$$r(v,n,m,h) = \bar{g}_K n^4 (v - V_k) + \bar{g}_{Na} m^3 h (v - V_{Na}) + \bar{g}_l (v - V_l)$$

which are the reaction terms from our Hodgkin-Huxley equations; in particular the terms from (1).

3.2. Method of Lines

For this project I implemented the Method of Lines scheme, that is a semidiscretization. The domain $\Omega = [0,1]$ was discretized in space and I implemented this method in two ways using a nested for-loop for the spatial solving and then using a stencil matrix. If we recall the PDE system (1-4) and write it as update rules we have

$$\begin{aligned} v^{i+1} &= v^i + k\left(\frac{1}{C}D^2 v^i - r(v^i, n^i, m^i, h^i)\right) \\ n^{i+1} &= n^i + k\left(\alpha_n(v^i)(1-n^i) - \beta_n(v^i)n^i\right) \\ m^{i+1} &= m^i + k\left(\alpha_m(v^i)(1-m^i) - \beta_m(v^i)m^i\right) \\ h^{i+1} &= h^i + k\left(\alpha_h(v^i)(1-h^i) - \beta_h(v^i)h^i\right) \end{aligned}$$

where

$$r(v,n,m,h) = \bar{g}_K n^4(v - V_k) + \bar{g}_{Na} m^3 h(v - V_{Na}) + \bar{g}_l(v - V_l) \tag{8}$$

and the operations in r are *element wise* operations on the vectors. First, as an initial approached I used if-else with a for-loop nested in a for-loop to do the Forward Euler step. Below is the code:

```
for i=1:nT
    for j=1:nX
        %Forward Euler for time dependent ODEs on n,m,h
        nn(j,i+1)=nn(j,i)+k*(an(u(j,i))*(1-nn(j,i))-bn(u(j,i))*nn(j,i));
        mm(j,i+1)=mm(j,i)+k*(am(u(j,i))*(1-mm(j,i))-bm(u(j,i))*mm(j,i));
        hh(j,i+1)=hh(j,i)+k*(ah(u(j,i))*(1-hh(j,i))-bh(u(j,i))*hh(j,i));

        %If statement is for spatial-temporal
        if j==nX
            %Forward Euler with Centered Difference of 2nd Derivative
            u(j,i+1)=u(j,i)+(k/c)*(b/(h)^2*(0-2*u(j,i)+u(j-1,i))...
            -gk*nn(j,i)^4*(u(j,i)-ek)...
            -gna*mm(j,i)^3*hh(j,i)*(u(j,i)-ena)-gl*(u(j,i)-el));
        elseif j==1
            u(j,i+1)=u(j,i)+(k/c)*(b/(h)^2*(u(j+1,i)-2*u(j,i))...
            -gk*nn(j,i)^4*(u(j,i)-ek)...
            -gna*mm(j,i)^3*hh(j,i)*(u(j,i)-ena)-gl*(u(j,i)-el));
        else
            u(j,i+1)=u(j,i)+(k/c)*(b/(h)^2*(u(j+1,i)-2*u(j,i)...
            +u(j-1,i))-gk*nn(j,i)^4*...
            (u(j,i)-ek)-gna*mm(j,i)^3*hh(j,i)*(u(j,i)-ena)...
            -gl*(u(j,i)-el));
        end
    end
end
```

For this implementation I initialized four matrices n, m, h, u, where each matrix was $M \times N$ and M is the number of spatial points, and N is the number of time points. That is the rows correspond to the spatial location along the 1D neuron rod and the columns correspond to a particular time. For example $u(1,2)$ in Matlab notation would correspond to the solution at position x_1 at time t_2, $u_1^2 = u(1,2) = u(x_1,t_2)$. For the inner loop, at each spatial location I solve the Hodgkin-Huxley equations that are approximated by the Forward Euler (5) and Centered Differencing approximation (6). Below is what the solution

matrix u would be

$$V = \begin{pmatrix} v_1^0 & v_1^1 & v_1^2 & \cdots & v_1^N \\ v_2^0 & v_2^1 & v_2^2 & \cdots & v_2^N \\ \vdots & \vdots & \vdots & & \vdots \\ v_M^0 & v_M^1 & v_M^2 & \cdots & v_M^N \end{pmatrix} = \left(\begin{array}{c|c|c|c|c} \uparrow & \uparrow & \uparrow & \cdots & \uparrow \\ v^0 & v^1 & v^2 & \cdots & v^N \\ \downarrow & \downarrow & \downarrow & \cdots & \downarrow \end{array} \right)$$

and likewise matrices for the state variables n, m, h. For lines 4-6 in the code above I do a Forward Euler step on the state variables n, m, h; for example,

$$n_j^{i+1} = n_j^i + k\left(\alpha_n(v_j^i)(1 - n_j^i) + \beta_n(v_j^i)n_j^i\right)$$

which would be the FE step on (2), state variable n. The code with the if-else statements is where I perform FE on the spatial term of the PDE. The first two parts of the if statement, lines 19–17, correspond to the ends of the rod and taking into account the boundary conditions. For instance if x_j is the end of the rod (lines 9–13), then we have 0 mV; therefore,

$$v_j^{i+1} = v_j^i + \frac{k}{c}\left(\frac{b}{h^2}\left(-2v_j^i + v_{j-1}^i\right) - r_j^i\right)$$

For the else part, lines 18–22, I do the FE step:

$$v_j^{i+1} = v_j^i + \frac{k}{c}\left(\frac{b}{h^2}\left(v_{j+1}^i - 2v_j^i + v_{j-1}^i\right) - r_j^i\right)$$

where

$$r_j^i = r(v_j^i, n_j^i, m_j^i, h_j^i) = \bar{g}_K(n_j^i)^4(v_j^i - V_k) + \bar{g}_{Na}(m_j^i)^3 h_j^i(v_j^i - V_{Na}) + \bar{g}_l(v_j^i - V_l).$$

For my second approach to numerical solving, I used a stencil matrix of the form (7) to to replace the solving done by the inner for-loop and if-else statements for the spatial solve. Below is the code:

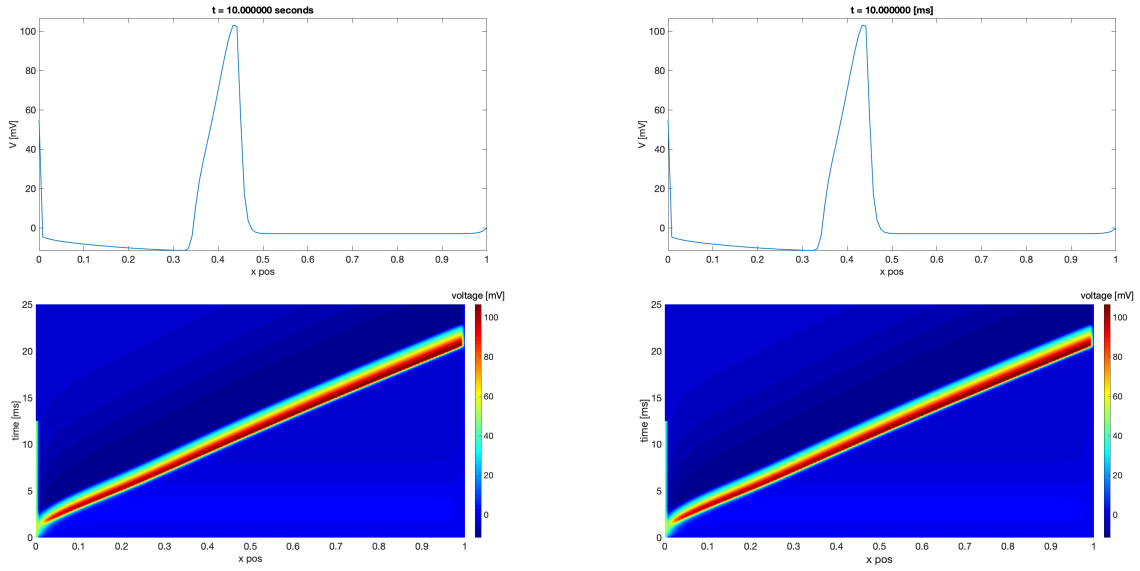
```
1  %define stencil matrix
2  sten = [1 -2 1];
3  sysMat = spdiags(ones(nX,1)*sten,-1:1,nX,nX)*b/h^2; %sparse diagonal
4
5  %reaction term
6  f = @(nn,mm,hh,v) (-1).*(gk.*nn.^4.*(v-ek)+gna.*mm.^3.*hh.*...
7      (v-ena)+gl.*(v-el));
8
9  for i=1:nT
10     %Using System matrix
11     u(1:nX,i+1)=u(1:nX,i)+(k/c)*(sysMat*u(1:nX,i)+...
```

```
12          f(nn(1:nX,i),mm(1:nX,i),hh(1:nX,i),u(1:nX,i)) );
13      %Forward Euler for time dependent ODEs on n,m,h
14      nn(1:nX,i+1)=nn(1:nX,i)+k*(an(u(1:nX,i)).*(1-nn(1:nX,i))...
15      -bn(u(1:nX,i)).*nn(1:nX,i));
16      mm(1:nX,i+1)=mm(1:nX,i)+k*(am(u(1:nX,i)).*(1-mm(1:nX,i))...
17      -bm(u(1:nX,i)).*mm(1:nX,i));
18      hh(1:nX,i+1)=hh(1:nX,i)+k*(ah(u(1:nX,i)).*(1-hh(1:nX,i))...
19      -bh(u(1:nX,i)).*hh(1:nX,i));
20  end
```

The main difference is in lines 11–12 above, compared to the previous code this was the if-else statements. In lines 11–12 we are computing

$$v^{i+1} = v^i + k(Av^i + r(v^i, n^i, m^i, h^i)); \tag{9}$$

where $v^i = \{v_0^i, v_1^i, \ldots, v_M^i\}$ is the solution voltage at time t_i.



(a) MOL with nested for-loops

(b) MOL with stencil matrix

Figure 2: The above simulations ran with parameters: 120 spatial steps, 2000000 time steps over a time interval $[0, 25]$. For both figures (2a) and (2b), the top two plots are a voltage profile at $t = 10$, the bottom profiles show the change in voltage against time along the whole 1D rod.

For figure 2 above I ran simulations for both MOL implementations with 120 spatial steps and 200,000 time steps. The results are very promising in that they do visually demonstrate the behavior of an action potential traveling down a 1D rod.

3.3. Operator Splitting

The second scheme I implemented is the Operator splitting, our system of equations (1–2) can be written more simply as

$$v_t = \alpha v_{xx} + r(v, n, m, h) \tag{10}$$
$$n_t = f_1(v, n) \tag{11}$$
$$m_t = f_2(v, m) \tag{12}$$
$$h_t = f_3(v, h) \tag{13}$$

where r is the reaction term function (8). For this scheme I divided the PDE numerical solve into two parts:

Step 1. First solve the spatial problem alone: $v_t = \alpha v_{xx}$ using the data at the beginning of the time step.

Step 2. Then solve the reaction components using the results of Step 1. as data for Step 2. This is solve $v_t = r(v, n, m, h)$ and (11–13).

For the operator splitting let $[t_j, t_{j+1}]$ be a time step then in that one time step, start with $v^j = v(t_j)$ and compute

$$v^* = (I + kA)v^j$$

where v^j is the incoming data at t_j and A is the stencil matrix for D^2. I made my own diffusion solve to accomplish this, it is a FE-CD method. This first part computes the approximation to $v_t = \alpha v_{xx}$, v^* is our intermediate solution. Now do FE on the time ODEs (11)–(13), that is

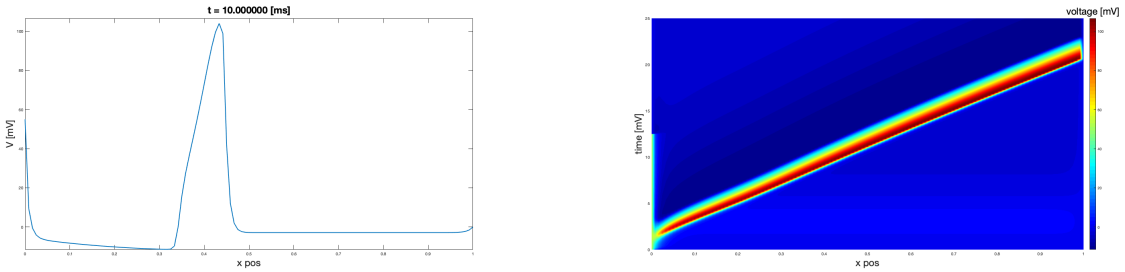
$$v^{**} = v^* + kr(v^*, n^j, m^j, h^j)$$
$$n^{j+1} = n^j + kf_1(v^*, n^j)$$
$$m^{j+1} = m^j + kf_2(v^*, m^j)$$
$$h^{j+1} = h^j + kf_3(v^*, h^j)$$

where v^* is the incoming data from our diffusion solving. Then $v^{j+1} = v^{**}$, which is the approximation to the solution at the end of the time step. Below is the implementation code:

```
% Now operator splitting
for j=1:nTime-1
    % Diffusion Solve
   [Uout,~,~]=diffSolve(U(1:nR,j),U(1,j+1),U(nR,j+1),b,nX,3,x0,...
```

```
                            xf,t(j),t(j+1));
    tmpU=Uout(:,end);

     %Now Solve Reaction Equation using forward euler
           U(2:nR-1,j+1)=tmpU(2:nR-1)+k*f1(U(nR+2:2*nR-1,j),...
               U(2*nR+2:3*nR-1,j),U(3*nR+2:4*nR-1,j),tmpU(2:nR-1));
     % Solve n,m,h using forward euler
        U(nR+1:2*nR,j+1)=U(nR+1:2*nR,j)+k*f2(U(nR+1:2*nR,j),tmpU);
      U(2*nR+1:3*nR,j+1)=U(2*nR+1:3*nR,j)+k*f3(U(2*nR+1:3*nR,j),tmpU);
      U(3*nR+1:4*nR,j+1)=U(3*nR+1:4*nR,j)+k*f4(U(3*nR+1:4*nR,j),tmpU);
end
```



(a) Action potential profile at $t = 10$.

(b) Full profile through time and space

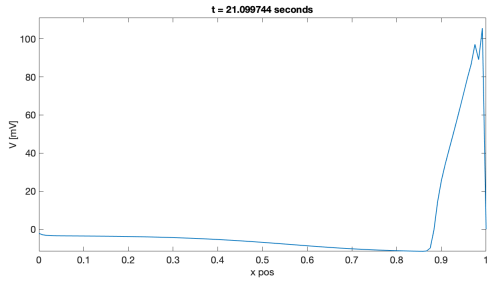
Figure 3: For both figures the number of spatial steps was 120 and the number of time steps was 200,000.

For the diffusion solve you specify the boundary conditions at the current time and you specify the number of time steps in $[t_j, t_{j+1}]$. For these simulations I used three time steps in each $[t_j, t_{j+1}]$ for the diffusion solve. Below is the implementation code for the diffusion solver:

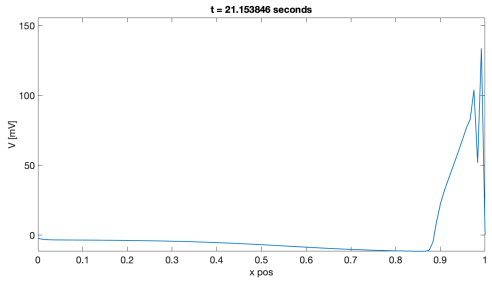
```
% for tridiagonal matrix
r=a*k/(h^2);
% tridiagonal matri
K1D=spdiags(ones(nR-2,1)*[r 1-2*r r],-1:1,nR-2,nR-2);
% used for booking keeping of dirichelet conditions
myI=eye(nR-2);
for j=1:nTime-1
    Uout(2:nR-1,j+1)=K1D*Uout(2:nR-1,j)+r*(myI(:,1)*Uout(1,j+1)...
    +myI(:,end)*Uout(end,j+1));
end
```

4. Discussion

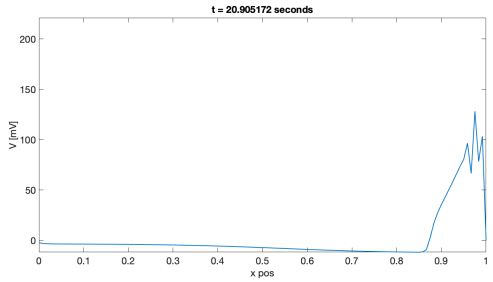
For my MOL implementations I did observe numerical instability and this depended on the time step size k and spatial step size h. For the simulations demonstrated above, I was able to get away with as little 6000 time steps before, I started observing numerical instability, below are the plots showing the instabilities that appear:



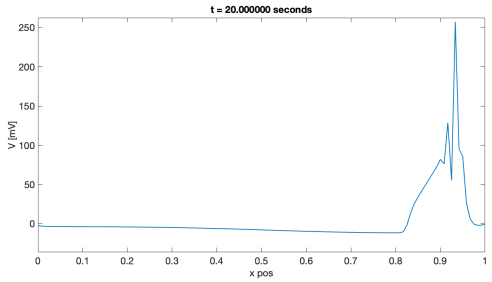
(a) 120 spatial steps, 5865 time steps



(b) 120 spatial steps, 5850 time steps



(c) 120 spatial steps, 5800 time steps

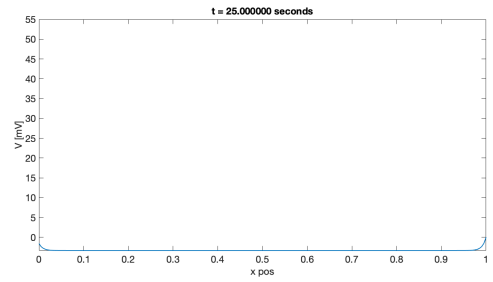


(d) 120 spatial steps, 5500 time steps

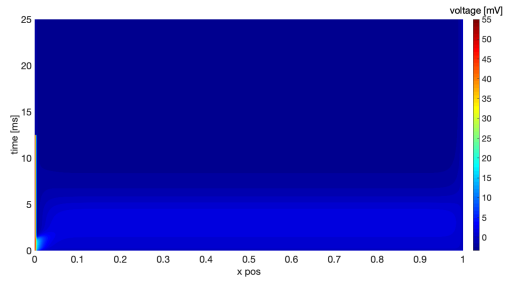
Figure 4: The above plots show the numerical instability getting worse for fewer number of time steps, larger time step size. These were run using MOL with nested for-loops.

I also noticed that for a finer grid discretization the instability became more profound. Recall that in order to obtain numerical stability with FE time stepping we need $|1+\lambda k| \leq 1$, that is we need to fall within the region of absolute stability. Therefore,

$$-2 \leq \frac{4k}{h^2} \leq 2 \Longrightarrow k \leq \frac{1}{2}h^2$$



(a) 200 spatial steps and 200,000 time steps



(b) full profile

Figure 5: As you can see above for 200 spatial steps with 200,000 time steps our solution does not get resolved.

If we were to attempt this with a finer grid discretization, i.e. 200 spatial steps on $[0, 1]$, then

$$k \leq \frac{1}{2}(1/200^2) = 1.25 \times 10^{-5}.$$

Since $k = 25/M$, the number of spatial steps we have that $M = 2000000$ spatial steps, which is very time consuming. I ran a simulation with 200 spatial steps and $M = 200000$ time steps and the solution did not get resolved as shown in Figure 4. I also did an accuracy analysis plot (in time) of the MOL implementation with a stencil matrix. I used MOL with

nested for-loops with 400,000 time steps as my "true" solution. In figure 6 the convergence is initially linear, but then as we approach the grid resolution of the true solution it drops very quickly. Even though the center differencing scheme (6) is second order accurate, it is dominated by the lower order scheme FE which is only first order accurate.

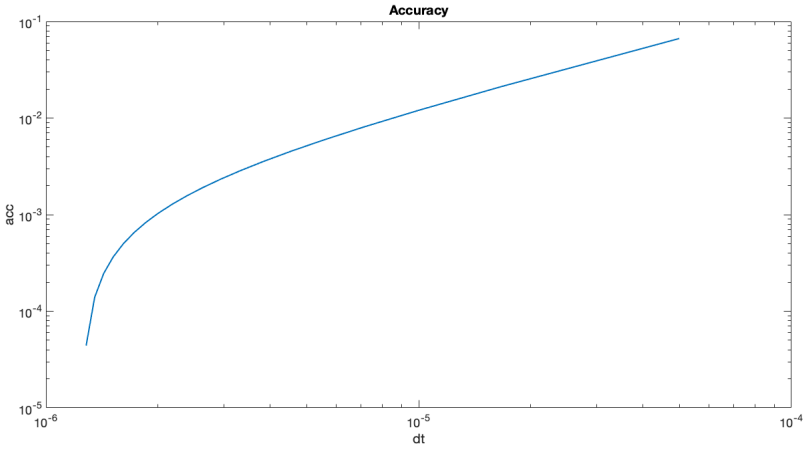
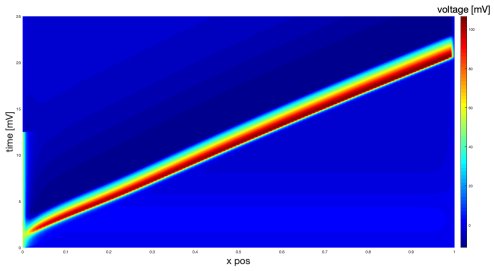
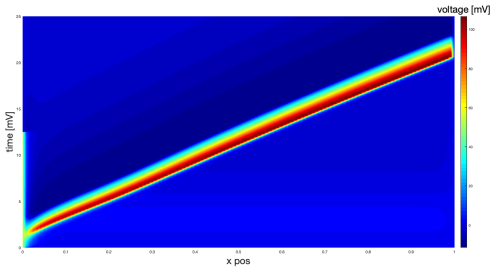


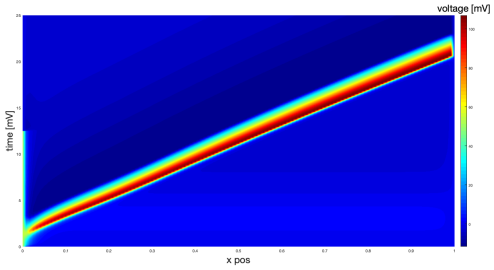
Figure 6: Accuracy plot of MOL with a stencil matrix for decreasing time step size, accuracy was checked at the end of the rod



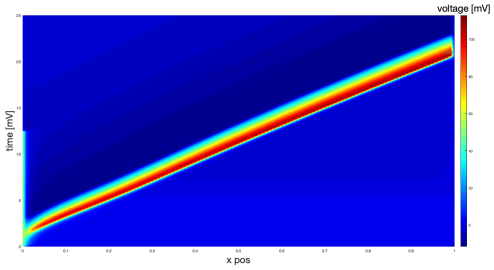
(a) with 100,000 times steps



(b) with 50,000 times steps



(c) with 10,000 times steps



(d) with 5,437 times steps

Figure 7: Operator Splitting, for (d) notice the "smearing" in the top right corner region.

For the Operator Splitting, I noticed a little nicer stability with fewer time steps. For the above four figures we have promising qualitative results in terms of the plots and we can "get away" with fewer time steps when compared to the MOL approach. Below is a figure demonstrating the instability that arises:

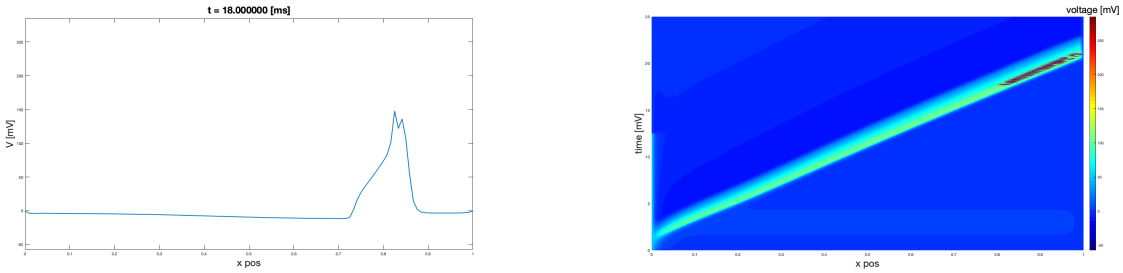
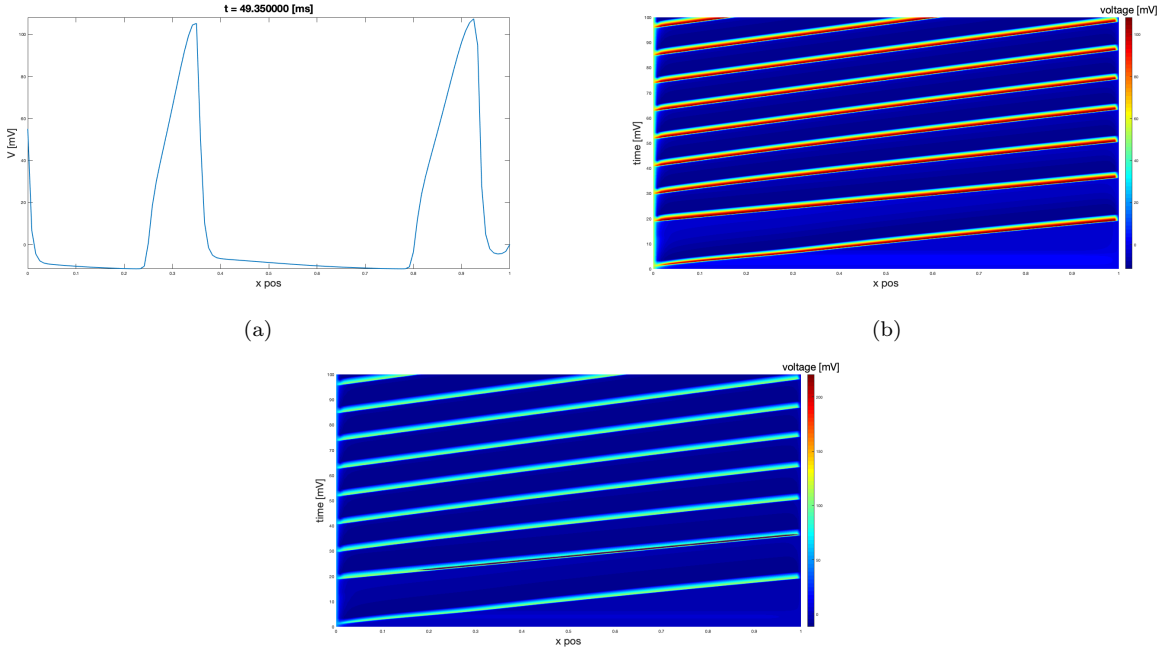


Figure 8: This run was done with 120 spatial steps and 5000 time steps, observe the sharp peaks formed, this would not be biologically accurate. Observe that the smearing is more obvious in the top right region.

4.1. Spike Trains

In this short section, I demonstrate that with a small change in the parameters, I was able to generate the characteristic "spike train" of action potentials. For these runs I use the Operator splitting implementation, with a fixed voltage of 55 mV at one end of the rod.



(a)

(b)

(c)

Figure 9: Both figures were generated with 120 spatial steps, with sodium ion conductance set to 150, and the times set to 100 ms with 100,000 time steps. For (c) the number of time steps was set to 30,000 and only one of the spikes demonstrated instability.

An interesting modification, I held the 55 mV at the middle of the cylinder and observed the following:

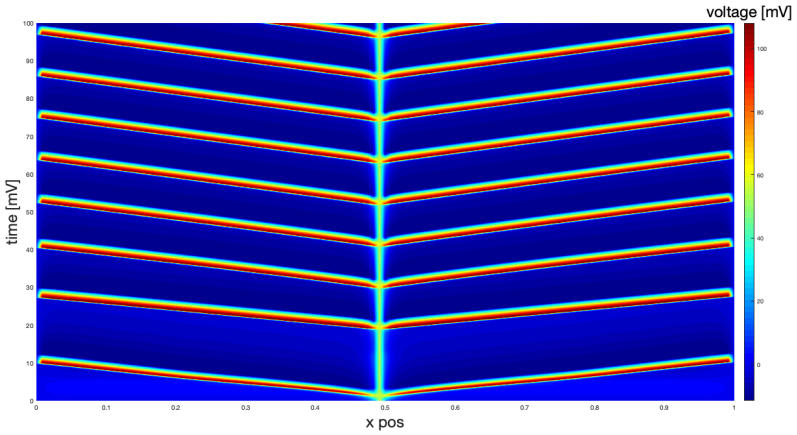


Figure 10: Source held at the middle of the cylinder

The middle part is the 55 mV source being held at the center of the geometry. Then we have two pulse trains traveling to the ends of the rod.

4.2. The Y-Branch

For the branching geometry we break the geometry into subgrids at nodes. As an example borrowed from [8, 3] a multibranch is shown below: As an example, I am considering the

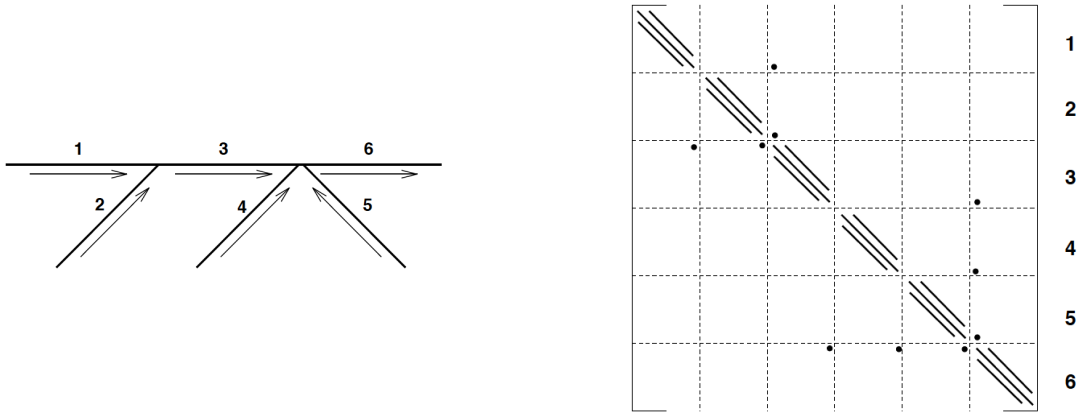


Figure 11: The diagram on the left shows the branching and the sparse matrix is shown on the right.

Y-branch shown below Essentially, an implementation would involve modifying the stencil

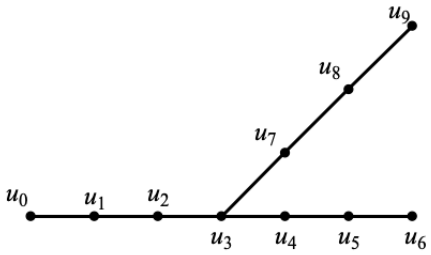


Figure 12

matrix (7). If I consider one branch to be $u_0, \ldots, u_6$ and the second branch u_3, u_7, u_8, u_9

then the stencil matrix would be of the form

$$\left(\begin{array}{cccccc|ccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -2 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right)$$

or an alternative stencil would be to divide the geometry into three branches

$$\left(\begin{array}{ccc|ccc|ccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right)$$

This part of the project is still being completed.

5. Virtual Reality

For the virtual reality implementation of this project, I utilized Unity VR© framework to run the numerical solving scripts and then the scalar values from the numerical solves are mapped to a color value for the surface of the cylindrical geometry. The steps to approaching this part of the project are as follows:

Step 1) Find a mathematics library in C# which has all the built in functionality of matrix-matrix multiplication, matrix-vector multiplication, "back-slash" solve, and matrix/vector manipulation like MatLab has.

Step 2) Generate geometries that are compatible with Unity.

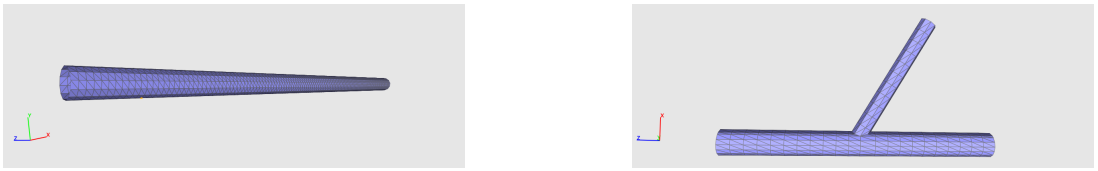
Step 3) Then code a numerical solver in C#.

Step 4) Use the numerical solver with Unity shading/texturing functionality to animate an action potential on the geometry.

Unity uses C# language for programming animations and visualizations, I was able to find a library MathNet [9]. This library contains a matrix and vector type along with a library of functions for doing calculations and manipulations. For this project I used the MOL-matrix approach in Unity, below is a snippet of the code for the solving

```
1  for (int i = 0; i < MOL_Matrix.nT; i++)
2          {
3              uCurr_out = U.SubMatrix(0, nRows, i, 1).ToColumnMajorArray();
4              Debug.Log("Here is U_curr = " + uCurr_out);
5
6              react = reactF(U.SubMatrix(0, nRows, i, 1), NN.SubMatrix(0, ...
                   nRows, i, 1),
7              MM.SubMatrix(0, nRows, i, 1), HH.SubMatrix(0, nRows, i, 1));
8
9              SYS.Multiply(U.SubMatrix(0, nRows, i, 1), temp2);
10             temp = U.SubMatrix(0, nRows, i, 1) + (k / MOL_Matrix.c) *
11                 (temp2 + react);
12
13             U.SetSubMatrix(0, i + 1, temp);
14             ...
```

In the above code, line 3 sets the current solution voltage vector v^i, a column of `U`, to an output array `uCurr_out`. In line 6, the reaction term is calculated using v^i, n^i, m^i, h^i. Line 9–11 is where we perform the FE-CD calculation which is the calculate (9). For the geometries, Unity VR needs .obj file types:



(a) (b)

Figure 13: (a) is the cylinder geometry that corresponds to a 1D rod with 120 points, and (b) is the Y-branch geometry

For the geometries, a 1D to 3D mapping file was needed to map the 1D coordinate points to the 3D points of the geometry. That is to say, a point on the 1D rod is the center of a circle. The points on the circle represent a ring of points on the cylinder.

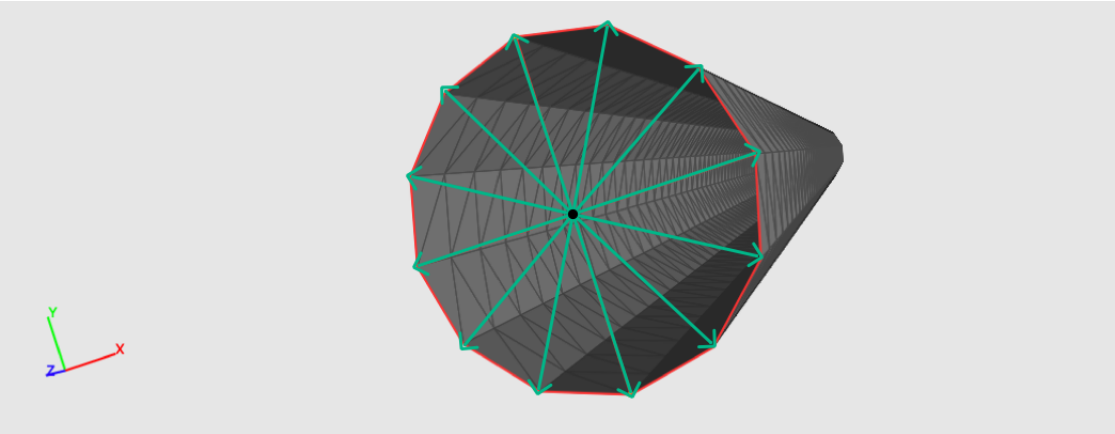


Figure 14: In this figure the black point corresponds to a 1D point and its scalar values are mapped to 3D points on the surface of a cylinder, the red ring of edges are the edges that connect the ring of points.

I will now explain how the Unity VR simulation run works. In the Unity VR when the "play" button is pressed the Simulation script is called and the Simulation script calls the Gaussian Shader routines and the numerical solver script, `MOL_matrix_v2.cs`. There are two threads running, this is a sequence of programming instructions: one thread corresponds to the Unity VR environment that is running in time and the second thread is the Simulation script that is performing the numerical solves and assigning shading/coloring to scalar values.

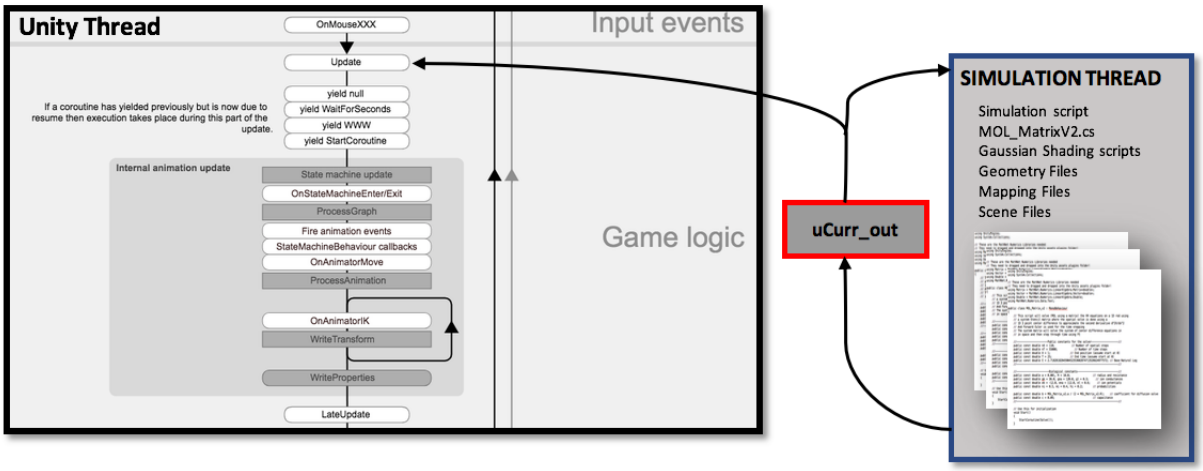


Figure 15: The left pane is the unity thread [10] and the right pane is the simulation thread

Here is a step by step run down of what is happening behind the scenes:

1) The play button is pressed, two threads are started, the simulation thread load the geometry and scene components into Unity.

2) At the first time step the simulation thread call the numerical solver.

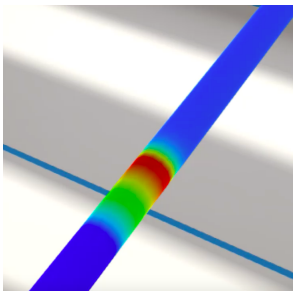
3) Then solve only solve one time step, that is one iteration of the for-loop.

4) The solution at the end of the time step is computed and placed in an output array `uCurr_out`.

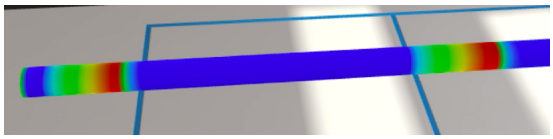
5) The Unity thread take the `uCurr_out` scalar value vector and it is mapped to a color on the surface geometry.

6) The process repeats with `uCurr_out` being sent back to the simulation script to compute the next solution the next time step.

Below are two stills from the Unity VR simulation run:



(a) Initial simulation with one action potential pulse.



(b) Simulation with action potential pulse train.

Figure 16: Demo simulation runs

6. Conclusions

In this section I will first discuss the challenges faced in the project and then parts of the project that need refinement and/or completion. The first challenge was implementing the numerical solvers in Matlab and then translating the solver into C# code. This was challenging since an appropriate math library was needed in order to do matrix-vector multiplication. In this case the MathNet numerics package was best since it is written in C# and has a diverse set of matrix,vector functions analogous to functions in MatLab. Another challenge that presented itself was the transfer of the geometries into the Unity framework, Unity for some reason would add "extra" vertices to the geometries. This was manualley remedied; however, for future work it may be more efficient to have Unity generate the geometry files and 1D to 3D vertex mapping files. The numerical solver that was implement in C# was the MOL with FE–CD using a matrix stencil to approximate the second derivative. This surprisingly worked very smoothly on the cylinder geometry; however, I plan on coding up more efficient solvers and solvers the require less number of time steps. In particular Crank–Nicolson and BE are utilized in Yale's Neuron program and suffices when numerically running action potential simulations.
Some of the items that need to be addressed in this project are

- Implementing the Y-branch geometry, currently I am working on the prototyping in MatLab of a stencil matrix that can accurately solve the Hodgkin–Huxley equations on a Y–branch geometry. Then this needs to be implemented in Unity VR. One challenge

anticipated is the mapping of the voltage scalars to the geometry since we no longer have a "linear" set of spatial locations.

- We are also working on incorporating Unity VR's "beam" feature. That is a user of the Oculus VR googles can interact with the geometry and "hold" a beam of voltage on the cell and watch it solve in real time.

- At the next step of this project, once the Y–branch geometry has been incorporate we would like to run VR simulations on a full cell. This is feasible since a neuron cell is essentially a collection of Y–branch geometries that are all connected.

- We also need to consider what numerical scheme(s) are best suited for the VR system as of now MOL with FE–CD appears "to do the job" however for more complicated geometries or long runtimes we may need a numerical solve that requires fewer time/spatial steps.

References

[1] P. Dayan and L. Abbott, *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*, The MIT Press, 2001.

[2] G. B. Ermentrout and D. H. Terman, *Mathematical Foundations of Neuroscience*, Springer, 2010.

[3] M. Hines, *Efficient computation of branched nerve equations*, International Journal of Bio-Medical Computing, 15 (1984), pp. 69 – 76, https://doi.org/https://doi.org/10.1016/0020-7101(84)90008-4, http://www.sciencedirect.com/science/article/pii/0020710184900084.

[4] A. Hodgkin and A. Huxley, *A quantitative description of membrane current and its application to conduction and excitation in nerve*, Journal of Physiology, 117 (1952), pp. 500–544.

[5] P. Koch, C. Koch, and I. Segev, *Methods in Neuronal Modeling: From Ions to Networks*, A Bradford book, MIT Press, 1998, https://books.google.com/books?id=5GMV2onekvsC.

[6] R. LeVeque, *Finite Difference Methods of Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems*, Society for Industrial and Applied Mathematics, 2007.

[7] H. Lieberstein, *On the hodgkin-huxley partial differential equation*, Mathematical Biosciences, 1 (1967), pp. 45–69.

[8] M. V. Mascagni, *Methods in neuronal modeling*, MIT Press, Cambridge, MA, USA, 1989, ch. Numerical Methods for Neuronal Modeling, pp. 439–484, http://dl.acm.org/citation.cfm?id=94605.94628.

[9] Math.Net, *Math.net numerics*, 2019, https://numerics.mathdotnet.com (accessed 2019).

[10] Unity, *Order of execution for event functions*, 2019, https://docs.unity3d.com/Manual/ExecutionOrder.html (accessed 2019).