Experimental Soft Matter Physics
Module 1: Excitable Systems Lab

Learning Objectives

1. To view the excitation of action potentials in neurons through interdisciplinary and historical lenses, putting physics into conversation with other scientific disciplines.
2. To explore the basic properties of excitable systems using classic experiments and mathematical models, emphasizing nonlinear aspects of their behavior.
3. To gain experience with basic scientific programming.

Introduction

Neurons enable an animal to rapidly transmit information from one part of its body to another. Isaac Newton, in later editions of his famous *Principia Mathematica*, hypothesized that this form of communication might be electrical in nature. Experiments performed during the eighteenth and nineteenth centuries confirmed the presence of electrical activity in neuromuscular tissues, but also sparked intense speculation and debate about the underlying scientific basis for this activity. One of the first and most famous science fiction novels, Mary Shelley's *Frankenstein* (published in 1818), was partially inspired by this debate.

We might be tempted to think of neurons as electrical wires of some sort. While this circuit metaphor is quite useful, as we shall see, neurons are actually terrible electrical conductors. What, then, is the physical basis for electrical signal propagation in a non-conducting organic material? The answer, which started to take shape in the middle of the nineteenth century, is *wave propagation*: when a neuron receives a sufficiently large stimulus, it generates a dramatic but transient spike in electrical potential, known as an *action potential*, which moves from one neuron to another the way audience waves move at large sporting events. These waves are, however, unlike anything you've ever seen in another physics course: they are *nonlinear*.

To understand what we mean by nonlinearity here, recall that sinusoidal mechanical and electromagnetic waves are solutions of a generic partial differential equation known as the wave equation. Significantly, the wave equation is *linear* and its solutions obey a superposition principle. Neural excitation is quite different. Only sufficiently large stimuli can excite an action potential for example and, once excited, the cell cannot be excited again until after it returns to its resting state. Thus, action potentials cannot be superimposed and, unlike mechanical and electromagnetic waves, cannot pass through one another as they propagate. These behaviors are characteristic of a broad class of nonlinear systems, now known collectively as *excitable systems*.

In this lab, we will explore a highly influential model that serves as a prototype for excitable systems more broadly. This simplified model was inspired by a much more complicated model that quantitatively describes the electrochemical basis for action potentials in neurons. As we discussed in class, Richard FitzHugh developed a deceptively simple-looking pair of equations, now known as the *FitzHugh-Nagumo model*, that captures the essence of neural excitation dynamics:

$$\dot{V} = V - \frac{1}{3}V^3 - W + I, \tag{1}$$

$$\dot{W} = \epsilon(V + a - bW). \tag{2}$$

Here, the variable V mimics rapid changes in potential difference across the cell membrane, while W captures slower processes that help the neuron recover following excitation. This separation between faster and slower time scales is controlled by the parameter ϵ. For $a = 0.7$, $b = 0.8$, and $\epsilon = 0.08$, these equations have a unique resting state, at which $\dot{V} = 0$ and $\dot{W} = 0$. In this lab, you will examine some of the fascinating history and experimental advances that inspired these equations. Your ultimate goal, however, will be to explore how the model works. This exploration will also serve as a broader introduction to nonlinear science and to scientific programming techniques.

Biophysical Background

Living cells actively generate a nonzero potential difference across their cell membranes, known as the *membrane potential*. This is possible because cell membranes are selectively permeable to specific ions, such as Na^+, K^+, and Cl^-, and because this selectivity enables cells to control the concentrations of these ions on either side of the cell membrane. In effect, borrowing language from basic electronics, the membrane acts like a capacitor connected in parallel with several batteries and resistors, as shown in Fig. 1 below. These "resistors" do not act like anything found in an standard electronics lab, however. In particular, they do not obey Ohm's law: their resistances change with time and as the membrane potential changes. In neurons and other electrically excitable cells, these nonlinearities provide the biophysical basis for action potential dynamics.

Background reading (experimental): The circuit diagram shown below emerged from a series of groundbreaking experiments performed by Hodgkin and Huxley in the early 1950s. These experiments featured a remarkable technique, now known as a *voltage clamp*, that allowed Hodgkin and Huxley to probe the relationships between membrane and ion currents in a giant axon taken from a squid. But how does a voltage clamp work and, more importantly, how did Hodgkin and Huxley take advantage of its capabilities? To help you explore these questions, I'm providing you with three chapters from a popular neuroscience textbook. In this source, you'll find a relatively accessible introduction to membrane dynamics and the experimental evidence for *nonlinear* processes that drive action potentials, including a useful account of the voltage clamp.

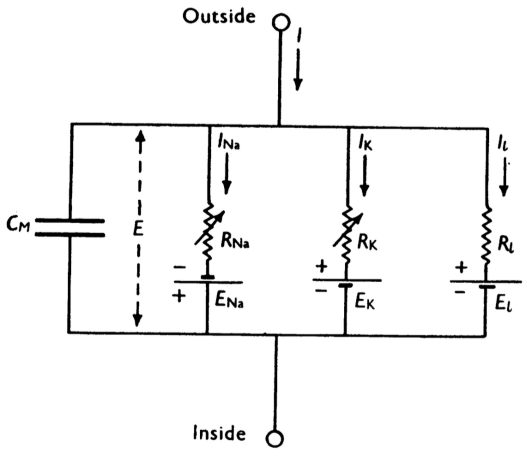

Figure 1: Equivalent circuit for a neuron, representing a small patch of cell membrane. The resistances, unlike those described by Ohm's law, vary with time and with potential. (Figure taken from Hodgkin and Huxley (1952).)

Background reading (theoretical): Hodgkin and Huxley published a series of five papers describing their work. The last of these uses the circuit diagram shown in Fig. 1 to propose a detailed ionic model, now known as the *Hodgkin-Huxley model* of neural excitation. As we discussed in our last meeting, this model was a major landmark in mathematical biology (Hodgkin and Huxley would eventually receive a Nobel Prize) and led directly to FitzHugh's later discovery of a simpler description of neural excitation. In this way, the Hodgkin-Huxley model forms a bridge connecting the history described above to the birth of *excitable systems* as an important paradigm in nonlinear science. With this in mind, spend some additional time reading about the model using any sources you find helpful. Note the mathematical complexity of the Hodgkin-Huxley equations, especially relative to the FitzHugh-Nagumo model. The rest of this lab will explore Eqns. (1) computationally.

Scientific Computing

Scientific computing is a critical component of active physics research. If you've never done any scientific programming, apart from basic data manipulation in a spreadsheet, you may be wondering how to even start. There are a number of commercial programming platforms that offer the flexibility to handle pretty much any data analysis problem your research can throw at you, without forcing you to write your own algorithms at every single step or turn. In this course, we'll be using the platform I use in my own research, Matlab. Matlab is installed on three computers in Mudd 305, which are reserved entirely for PH333 work, and on a number of computer labs elsewhere in the natural sciences complex. More importantly, you can install it on your own computer and, using one of Colby's remote-access licenses, work from your own room.

As with most things, the best way to become familiar with Matlab is to just start using it. Start a Matlab session and try out the following commands, typing one at a time and pressing enter when done. Even if you've used Matlab before, working through these will provide a quick review of some of the basic syntax.

A calculator with many, many buttons

Matlab will do all the things you're used to doing on a calculator:

```
>>  1/(2^4 + 1)
```

It knows every transcendental function you do...

```
>>  exp(3)
>>  cos(pi/3)
```

...plus many other functions you may not be aware of:

```
>>  airy(3)
>>  gamma(3)
```

Defining matrices

Here's how you define a matrix A with two rows and two columns (use square brackets to signify the start and end, use a semicolon to signify the row breaks):

```
>>  A = [1 2; 3 4]
```

What happens if you put a semicolon at the end of the line?

```
>>  A = [1 2; 3 4];
```

Here's how you define a row vector b:

```
>>  b = [1 2 3 4]
```

The transpose operation, which in Matlab is achieved with a single quote, transforms it into a column vector...

```
>>  b = [1 2 3 4]'
```

...but you could also define a column vector directly using semicolons:

```
>>  b = [1; 2; 3; 4]
```

Accessing matrix elements

Matlab uses the same row-column notation you learn in a linear algebra class. Here's how you pick out second row of A:

```
>>  A(2,:)
```

Here's how you pick out second column of A:

```
>>  A(:,2)
```

Can you guess which element of A this command picks out?

```
>>  A(2,1)
```

What do you think this command does? Since A is square, this might look a bit strange...
To be sure about your interpretation, try different numbers until you figure out the pattern!

```
>>  A(2)
```

You can also redefine any particular element any time you like:

```
>>  b(1) = 4
```

Matrix algebra

Define three different matrices like so and then try out the commands that follow:

```
>>  A = [1 2; 3 4]
>>  B = [0 1; 2 3]
>>  x = [1 2]
```

The sum of two matrices:

```
>>  A + B
```

The difference of two matrices:

```
>>  A - B
```

The product of two square matrices:

```
>>  A*B
```

The product of a square matrix and a column matrix:

```
>>  B*x'
```

A riddle, for those of you with linear algebra experience: why doesn't the previous command work without the transpose operator? Type the following and you'll get an error message:

```
>>  B*x
```

Another riddle: what is the difference between the following two commands?

```
>>  A*B
>>  A.*B
```

Excitable Systems

Let's get back into our exploration of the FitzHugh-Nagumo model now. We're going to need code that gives us control of every aspect of the dynamics, including initial conditions, elapsed time, and the various parameters in the model. Start by typing the following (and when prompted answer, yes, we do want to create a new file):

```
>> edit FHN_lab
```

This opens up an editing window, in which you will write new code that encodes the model equations in language suitable for Matlab's differential equation solvers. Copy the following commands directly into your editor:

```
function dy = FHN_model(t,y,I)

   % model parameters
   epsilon = 0.08;
   a = 0.7;
   b = 0.8;

   % model equations
   dy = zeros(2,1);
   dy(1) = y(1) - (y(1)^3)/3 - y(2) + I;
   dy(2) = epsilon*(...);

end
```

Note that the dynamical variables V and W from Eqn. (1) are here written as $y(1)$ and $y(2)$, respectively. (Instead of keeping track of separately named variables, Matlab prefers to combine them into one array.) Note also that the above is an incomplete translation of Eqn. (1): only the first of the two lines defining dy has been completed and the other one have been left for you as an exercise. When you're finished, save your work and close the editor window you've been working in. Now type the following (and, once again, we do want to create a new file here):

```
>> edit FHN_lab
```

This opens up another editing window, in which you will write new code that sets FitzHugh-Nagumo model parameters and runs one of Matlab's differential equation solvers. Copy the following commands directly into your editor:

```
% choose a value for I
I = 0;

% fix model parameters at traditional values
a = 0.7;
b = 0.8;

% find equilibrium value of membrane potential
v_eq = fzero(@(v)(v - (v^3)/3 - (a+v)/b + I),0);

% find equilibrium value of recovery variable
w_eq = (a + v_eq)/b;

% pick initial values
y0 = [v_eq, w_eq - 0.01];

% pick a time interval
tspan = [0, 50];

% use the equations you defined
ode = @(t,y) FHN_model(t,y,I);

% solve the equations
options = odeset('RelTol',1e-6);
[t,y] = ode45(ode,tspan,y0,options);

% plot results
plot(t,y(:,1))
```

Note that anything following a percent sign is treated as a comment, not as commands that are run by Matlab. Writing comments can slow you down at first but, in the end, your code will be much easier to read and use. When you're ready to go, just save your changes and type the name of this file (also known as a "script" in Matlab speak) at the prompt:

```
>>  FHN_lab
```

In its current form, the commands listed above end by producing a time series plot of the membrane potential $V(t)$. You don't need to run the code again to produce other plots though, by the way. If you want to see what the recovery variable $W(t)$ looks like, for example, type:

```
>>  plot(t,y(:,2))
```

Likewise, if you want to see the two-dimensional phase trajectory, type:

```
>>  plot(y(:,1),y(:,2))
```

If you want to change your time span or initial conditions, all you have to do is modify the appropriate lines of code in your file, save your changes, and run it again. Play around for a while now, trying out different initial conditions. Note that the code given above locates the initial condition relative to the resting state (V_{eq}, W_{eq}). You'll find this helpful. Instead of starting at $(V_{eq}, W_{eq} - 0.01)$, for example, you can systematically move away from the resting state by trying $(V_{eq}, W_{eq} - 0.02)$, $(V_{eq}, W_{eq} - 0.05)$, and so on as starting points. Does anything interesting change as you move your initial condition farther from the resting state? Use both $V(t)$ and phase trajectory plots as you explore this question.

You may, at some point, want to plot multiple curves on the same axes. There are several ways of doing this. Let's assume, as an example, that you want to compare $V(t)$ for two different initial conditions. First run your code. Then rename t and y as follows:

```
>>  tA = t; yA = y;
```

Now, change your initial condition and run your code again, generating new versions of t and y. Rename these as well:

```
>>  tB = t; yB = y;
```

Now, if you want to compare $V(t)$ for both data sets, the easiest way is to type:

```
>>  plot(tA,yA(:,1),tB,yB(:,1))
```

If this syntax is hard to remember or you ever simply want to learn more about the plot command, you can look up documentation that tells you everything you could want to know and more by typing:

```
>>  doc plot
```

Take notes on your programming choices and observations. Your notes should provide a clear record of what you did, clear enough that you could repeat the whole thing months later if you needed to. Be sure to save a few plots that capture some of your discoveries! Matlab will let you save your plots in many formats. I would strongly recommend the ".eps" format, for reasons that will be explained later. Printouts of favorite plots should go into your notebook and some of them may find an important place in a writeup later. Even now, at the beginning of your explorations, you can be thinking ahead about what figures will tell the best story later.

Charting the Threshold

One of the essential features of the FitzHugh-Nagumo model is its threshold behavior: sufficiently large perturbations to a system a rest trigger dramatic spikes, while smaller perturbations do not trigger these spikes. You probably noticed this range of behaviors in your exploration above. But how might we quantify the presence of a threshold? Thinking like an experimentalist, you might consider varying some sort of control parameter and then plotting some sort of response as a function of that parameter. This approach works very well here. Create a new script, with whatever name you like, and enter the following lines of code:

```
% create a vector containing a series of perturbation sizes
steps = 0.01:0.01:0.32;

% create another vector in which to store responses
responses = zeros(1,length(steps));

% loop over all perturbation sizes
for j = 1: length(steps)

   % pick initial values
   y0 = [v_eq, w_eq - steps(j)];

   % solve the equations
   [t,y] = ode45(ode,tspan,y0,options);

   % find maximum in membrane voltage response
   responses(j) = max(y(:,1)) - v_eq;

end

% plot results
plot(steps, responses, 'o-')
```

Study this code and see if you can figure out what each line is doing. You may have to look up documentation on specific commands or syntax. Note, in particular, the structure and syntax of the "for loop", which allows you to repeatedly run a subset of commands in an controlled way. Each iteration of the loop is associated with a value of j, which starts at 1 and is increased in steps of 1 until it reaches some final value. In this case, you run one iteration for each initial condition, each of which gets paired with a measure of the size of the response. The plot produced should give you a nice idea of why we describe this system as having a threshold. To avoid sharp corners in the curve, however, you'll need more data. What modification to the above lines will give a smoother curve? (Hint: look closely at the command used to select step sizes, i.e., explore the meaning of 0.01:0.01:0.32...)

What happens to the system's phase trajectories as you approach and pass through the steep region of the curve you've drawn? You might try plotting multiple curves on the same axes, for dramatic effect. You can do this using the syntax described above or using other useful commands. See, for example,

```
>>  doc hold
```

Additional Opportunities

At this point, you should have some sense for the excitation properties of the FitzHugh-Nagumo model and possibly a number of unanswered questions as well. Having looked at the model's threshold behavior, for example, you might be interested in exploring its refractory period as well. The easiest way to do this would be to take an existing trajectory and pick a moment at which you'd like to try exciting another spike. Let's say this point is the 300th in the time vector t. At

that moment, the phase trajectory is passing through the point $(y(300,1), y(300,2))$. You can use this as the initial value and then re-run your code using different step sizes. Again, you might want to save the t and y arrays as something else before doing this, e.g.,

```
>>  tA = t; yA = y;
```

or something like that. Alternatively, you can change the command that generates new t and y arrays to output to something with special names, e.g.,

```
[tC,yC] = ode45(ode,tspan,y0,options);
```

Can you find a relationship between how long you wait and how easy it is to generate another spike? Can you relate these observations to the shapes of the trajectories? You can use this sort of approach to test out other physiologically behaviors observed in real neurons. FitzHugh considers a number of these behaviors in his original paper, and these are also summarized in the Scholarpedia article on the model. The simplest example is a sharp transition from excitable to periodic spiking at a nonzero value of I. You find this transition empirically, just by running you code with different values of I and observing the solutions as functions of time and in phase space. We'll talk more about transitions like these in later labs.

In short, there are many additional opportunities here and please feel free to follow your interests wherever they lead you, either through reading further, doing more calculations on Matlab, or through a combination of techniques. I am including one of the Hodgkin-Huxley papers and one of FitzHugh's papers, in case this reading proves helpful and/or interesting. Please note, if you do look at these papers, that FitzHugh changed notation in later versions of the model. So his trajectories will look like a mirror reflection of yours. Above all, have fun!