Section 7. Matlab Tutorials

7.1 Introduction

Introduction to quantitative data analysis and *Matlab*

Neurophysiology invariably results in the collection of a great deal of neural data that must be analyzed to make interpretations about its meaning. There is essentially an infinite number of ways that neuronal data can be analyzed and a substantial fraction of the neuroscience community is involved in developing and improving such methods. Given this, a major part of learning how to do neurophysiological research involves learning about how to analyze the neuronal data that you collect. Indeed, most scientific research involves first thinking about and planning the analyses that will be applied to the collected data to get at the major questions of interest. This is part of good experimental design and it touches on many fields, but especially applied mathematics and statistics.

Thoughtful experimental design is especially important in neurophysiological research because the data often take a great deal of effort to collect and a researcher would not want to invest such effort only to later realize that he or she had not collected the type or amount of data that was most appropriate for the question(s) of interest. Thus, most neurophysiological data is not collected until the experimenter has in mind the kinds of analysis that will be applied to the data once it is obtained. This does not mean that those analyses are the only ones that will ultimately be applied to that data, but they are typically the first analyses that are applied.

Because neurophysiological data can be analyzed in many ways and those methods are often tailored to the details of each experiment, it turns out to be extremely useful to have very flexible analysis tools at your disposal. Basic, off-the-shelf statistical packages allow you to put data in a spreadsheet and run functions like t-tests and make basic plots (e.g. Microsoft Excel). These programs work very well for data that has a relatively simple form. For example, is the number of sick animals that improved with drug injection X greater than the number of sick animals that improved with a control saline injection? However, as mentioned above, neurophysiological data is often much more complex and thus cannot be easily handled by such packages. For example, each recorded neuron produces a voltage signal that varies in time as a function of a potentially large number of conditions that the subject was exposed to (e.g. many visual patterns presented to the eye). What aspects of that voltage signal should be analyzed? How can the data be processed to make those aspects explicit? Because this one neuron is part of a large population of neurons, how can we analyze the combined output of the entire population?, etc…

Because all modern neurophysiological data is converted to digital format (using analog to digital converters), the most flexible analysis tools are programming languages that can read that digital data and manipulate it in any possible manner. Examples of such programming languages are FORTRAN (old), C and C++. Although these programming languages have essentially infinite flexibility, and, once compiled into applications, are very fast, these advantages come at the price of having to write many basic routines from scratch (e.g. plotting routines, statistical routines, etc.). Moreover, these languages are very unforgiving in that programming errors can easily crash the application and can be difficult to detect. Finally, these languages do not offer an interactive environment in which variables can be defined, functions called, graphs created and edited, etc; instead, the entire program must be recompiled and run whenever a change is made.

It turns out that the best environments to analyze neuophysiological data are those that provide many basic, well-tested routines, are relatively robust to programming errors, can be easily transferred to different platforms, and still allow an essentially unlimited ability to analyze and view data in different ways. One example of such an environment in very widespread use today is *Matlab* (Mathworks, Natick, MA). Another example is *Python*. The main downside of such programs is that they tend to be less efficient than (e.g. C++) programs in terms of speed and computer memory use. Nevertheless, these disadvantages are typically not critical for most analyses (especially during the creative phase of developing new analyses) and, even if they do prove to be limitations, there are ways to transfer *Matlab* analysis code into compiled software that can run fast and efficiently.

Because of *Matlab*'s continued widespread use in engineering and science and its usefulness in analyzing neurophysiologic data, part of the goal of this course is for you to learn the basics of *Matlab* and to then use *Matlab* to perform basic analyses on the data you collect. The flexibility of *Matlab* will thus allow you to perform experiments and analyses that go beyond the basics provided in the lab. That is, it will free you to be creative and possibly discover things that are not built into the course.

Matlab tutorial projects

To help you learn the basics of *Matlab*, we have designed several tutorials that will progressively introduce you to *Matlab* in basic data analysis. These tutorials are critical for collecting and analyzing data from the fly vision labs. We will post the details of each tutorial assignment online, and you will submit all completed assignments through class site, as well. The due dates of each tutorial are shown in the course syllabus.

Matlab Project 0: **Running Matlab and basic plotting**
Matlab Project 1: **Spike detector**
Matlab Project 2: **Making a visual stimulus movie using Matlab**
Matlab Project 3: **Analysis of spike data to test motion direction selectivity**

One week of class will be devoted to helping you with the basics of Matlab and the completion of Matlab Project 1 (see Syllabus). If you need help with this or any Matlab tutorial, you should contact the appropriate teaching staff member.

7.2 Start Matlab: Plotting

Matlab Project 0

GOAL: The goal of Matlab Project 0 is to give you an introduction to Matlab basics. We have given you some data and the ultimate goal of the project is to plot that data.

Matlab's Integrated Development Environment (IDE)

Matlab is more than a programming language; it is a computing "environment" (formally termed the “integrated development environment” or IDE). Thus, we need to know the components of this environment and how to interact with it.

The **command prompt** (“>>”) is where to type commands you want Matlab to execute.

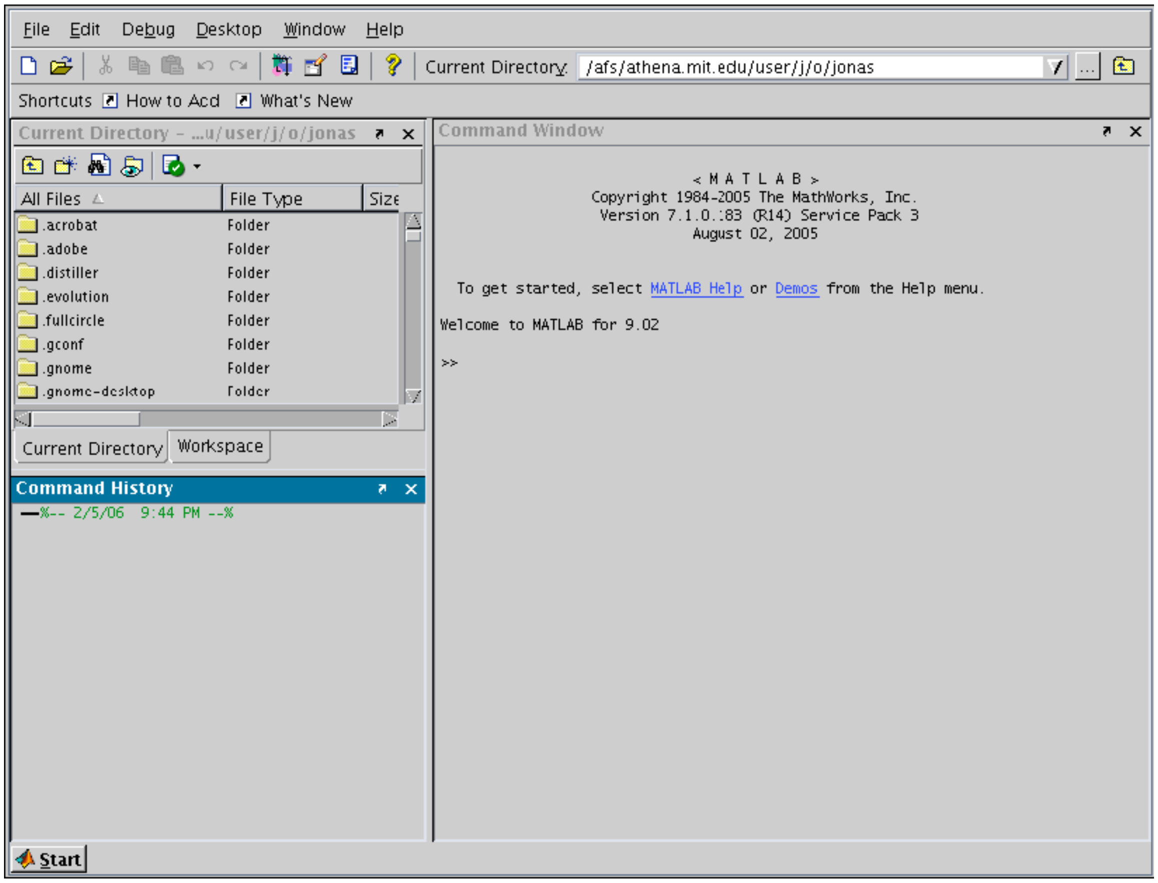
```
>>
```

The **current directory** is shown at the top of the window. It is important to know what directory you are in as Matlab will look in this directory first for the data you want to load or the commands you are inputing.

The optional partition to the left can display different information (you can choose which information by going to “Desktop” and choosing which options you’d like) including the:

-workspace: all variables that Matlab currently has to work with
-command history: a list of commands you’ve entered at the command prompt
-current directory: a list of the contents of the current directory

When you run Matlab you should see a brief splash screen, followed by a command window with the following:



Current Directory: /afs/athena.mit.edu/user/j/o/jonas

All Files	File Type	Size
.acrobat	Folder	
.adobe	Folder	
.distiller	Folder	
.evolution	Folder	
.fullcircle	Folder	
.gconf	Folder	
.gnome	Folder	
.gnome-desktop	Folder	

```
                    < M A T L A B >
        Copyright 1984-2005 The MathWorks, Inc.
         Version 7.1.0.183 (R14) Service Pack 3
                    August 02, 2005


  To get started, select MATLAB Help or Demos from the Help menu.

Welcome to MATLAB for 9.02

>>
```

```
%-- 2/5/06   9:44 PM --%
```

Changing Directories

When you run programs or load data in Matlab, Matlab must know where to look for the program or data. Matlab will first look in the current directory (which is listed at the top of the IDE). There are a few different ways you can change the current directory:

1. Type the directory into directory field at the top of the IDE.
2. Open the current directory partition and navigate to the directory.
3. Use command “cd” (“change directory) at the command prompt.

Creating Variables in the Matlab Workspace

Essentially, variables are labeled storage space for data. You can think of them as empty containers. These containers can be filled with whatever data you'd like, just one number or a vector or matrix of numbers for example. Once the variables are a part of our workspace, we can use them to do new computations and even give a value to new variables. Legal variable names are character strings consisting of letters, numbers, and underscores, and must begin with a letter; variable names are case-sensitive (“a” is a different variable from “A”).

Create your first variable, called "x". Type:

```
>> x = 3
```

You should see:

```
>> x = 3

x =

     3

>>
```

You have just created a variable named “x” and set its value equal to 3, and Matlab has echoed that value back to you. If you now type “x” (no quotes), Matlab will tell you the value of x.

```
>> x

x =

     3
```

Create another variable (e.g. y = 4). Now type “whos”. You should see:

```
>> whos
  Name          Size               Bytes  Class

  x             1x1                    8  double array
  y             1x1                    8  double array

Grand total is 2 elements using 16 bytes
```

You have just asked Matlab about what variables it now knows about. These variables live in something called the Matlab workspace. You can always see what variables are in your workspace by opening the workspace partition in your IDE or by typing "whos" at the command prompt.

If you do not want Matlab to echo the variables and values you create back at you, put a **semicolon** at then end of each statement. The character ‘**%**’ is the **comment character** – anything after this is ignored by Matlab. Also, once you have created a variable, you can always change its value later.

For example:

```
>> x = 3;                % create variable x and set its value to 3
>> x                     % ask for the value of variable x

x =
     3                   % Matlab reports variable x is equal to 3

>> x = 5;                % now x no longer equals 3 - that value has been overwritten!
```

You can use the variables in your workspace for computation. For example:

```
>> x = 3;
>> y = 6;
>> d = y-x               % subtraction

d =

     3

>> m = y*x               % multiplication

m =

    18

>> r = y/x               % division

r =

     2

>> z = x^2               % raise to a power (squaring in this case)

z =

     9
```

```
>> z2 = (x^2) + (y^2)

z2 =

    45
```

Creating and Manipulating Arrays

The variables you created above were all scalars (only one value for each variable). Matlab was actually built around the use of arrays (e.g. vectors and matrices), so it is very efficient at processing them. Thus, whenever possible use arrays in your code!

Create a vector called "v1" with three elements (note the use of the square brackets):

```
>> v1 = [ 4 2 9]

v1 =

    4    2    9
```

Create another vector called "v2" (also with three elements):

```
>> v2 = [ 3 2 10];
```

Now we can do some vector math:

```
>> v1 - v2

ans =

    1    0    -1
```

Here, because we did not assign a variable name to the output of "v1 - v2" Matlab created a variable called "ans" and assigned the output to it. Be careful! Because variables are so easily overwritten in Matlab, you should always give your variables meaningful and unique names.

What if v1 and v2 contain a different number of elements? Can you subtract them? If you do not know, try it by making a new vector "v3" that has four elements.

To create larger arrays, we use semi-colons to separate rows (here we make 3 x 3 arrays and subtract them):

```
>> a1 = [1 2 3 ; 4 5 6 ; 7 8 9]

a1 =

     1     2     3
     4     5     6
     7     8     9

>> a2 = [1 1 1 ; 2 2 2 ; 3 3 3]

a2 =

     1     1     1
     2     2     2
     3     3     3

>> a3 = a1-a2

a3 =

     0     1     2
     2     3     4
     4     5     6
```

You can also pick out pieces of an array and manipulate only those. For example, we can make a new variable called "new_var" and give it the values of the first row of a3:

```
>> new_var = a3(1,:)

new_var =

     0     1     2
```

Here, we use parentheses to index the variable a3. The order of indices is the order of dimensions: var(dim1, dim2, dim3) In Matlab, like mathematics, the first dimension is rows and the second dimension is columns. We wanted the first row, so we put a 1 in the first dimension, and we wanted all columns of that row, so we used a colon.

Saving Lists of Commands: the M-file

While typing commands at the command prompt is relatively simple, if you want to do more complicated calculations it can get tedious and frustrating. It would be better to write a whole set of commands and to execute them all at once. That is the purpose of Matlab's .m file (termed an "m-file").

Let's create our first program. First, we need to open a new m-file by going to File -->New --> M-File. Doing this opens the Matlab editor. Save the new m-file as "my_first_mfile". Then copy the lines of code that we just executed at the command prompt into the m-file, and save it:

```
clear;                          % removes all variables from the workspace
a1 = [1 2 3 ; 4 5 6 ; 7 8 9];
a2 = [1 1 1 ; 2 2 2 ; 3 3 3];
a3 = a1-a2;
```

To have Matlab execute the lines of code in your m-file, all we do is type in the name of the m-file at the command prompt:

```
>> my_first_mfile
```

Now if we look at the variables in our workspace, we have three: a1, a2, and a3. If we did not put semicolons at the ends of the lines in our m-file then Matlab would have spit out the output of each line as it executed it. Try it out! This becomes obnoxious very fast, so please remember to put semicolons at the end of your lines!

Loading and Saving Data

Matlab has its own data format, called a MAT file. MAT files contain variables. So any subset (from one variable to the whole workspace) of variables can be saved in a MAT file to be loaded at a later time. For example, if your workspace contains "x" and "y", you can type:

```
>> save xonly.mat x
```

And only the variable "x" is saved in the MAT file called "xonly.mat". To load that MAT file into your workspace, first clear your workspace and then load the MAT file:

```
>> clear
>> load xonly.mat
```

And "x" should then appear in your workspace. Remember, just like when executing m-files you should be in the directory where xonly.mat is located when you want to load it.

Plotting Data

Most plotting involves vectors. For example, suppose y = f(x) and I want to make a plot of that functional relationship. Mathematically, x is a continuous variable that can take on any real value. However, in the digital world, x and y are typically sampled at some interval.

For example, suppose $y = x^2$ and I want to plot that relationship over a range of x from 0 to 10. To start, we create a vector "x" sampled at a fixed interval (in this case the interval is 1).

```
>> x = [0:1:10]

x =

     0     1     2     3     4     5     6     7     8     9    10
```

If you do not type an interval, the default interval is 1. E.g. x = [0:10] would give you the same thing.

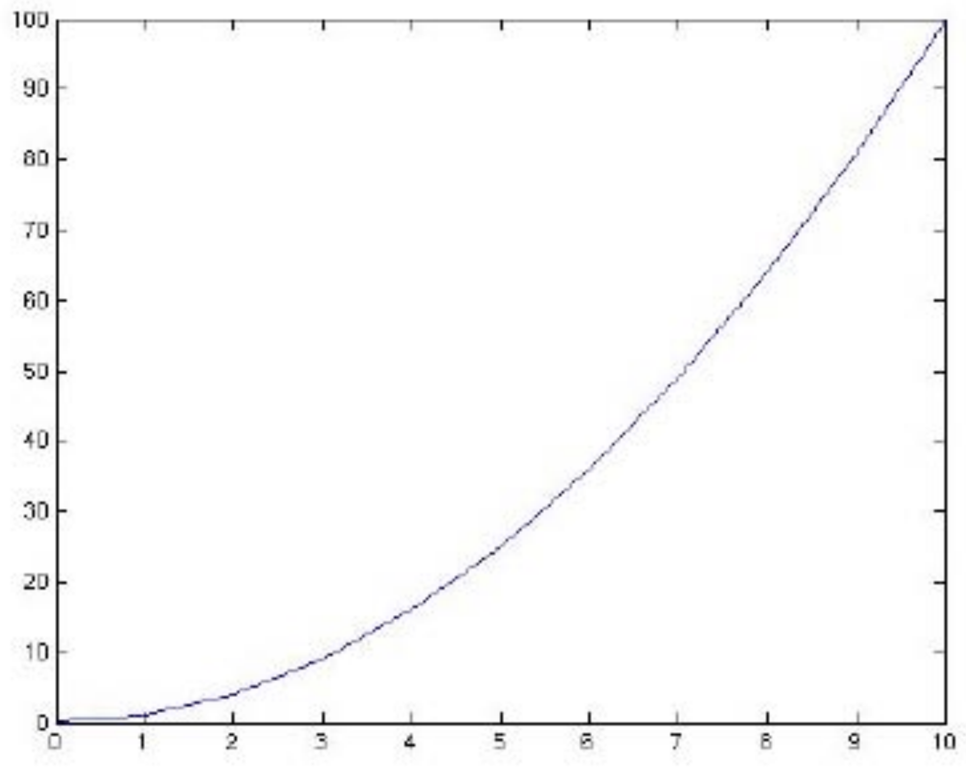
Now we create the variable "y" at all the sampled x points using the known functional relationship (squaring in this case). We want to compute the squaring element-by-element, that is, we want to compute x(1)*x(1), x(2)*x(2), etc. Matlab indicates per-element operations by use of the **.** (i.e. the period). For example:

```
>> y = x.^2

y =

     0     1     4     9    16    25    36    49    64    81   100
```

Now, it is simple to plot y as a function of x over this range

```
>> plot(x,y) % run the Matlab function called plot using your data in x, y
```

When you type this command, a new figure window will open and you should see the following figure:



Congratulations! You have just made your first plot in Matlab! You can add text and edit other properties of the figure in different ways. For example:

```
>> xlabel('my x label');

>> ylabel('my y label');

>> text(2,2,'my text')          % will add text on the plot at values x=2, y=2

>> figure                       % Will open a new figure window to make a plot

>> clf                          % Will clear the contents of the current figure window

>> hold on                      % will allow you to keep plotting to the same figure without
                                erasing what is already there
```

Try this:

```
>> figure(1)
>> clf
>> plot(x,y)
>> hold on
>> plot(x,y*2,'r')                 % make this plot red, see below
```

One of the convenient things about plotting in Matlab is that you can control a lot of the details about how the plot looks. For example, type this:

```
>> h = plot(x,y);
>> get(h)
```

In this case, plot will run as before and also return a "handle" in variable h to all the elements of the plot. "get" displays the current setting of all those elements. You can change the elements like this:

```
>> set(h, 'Color', 'r')            % set the color of the plot line to red ('r'). Other colors
                                   are blue ('b'), black ('k'), green ('g'), etc.

>> set(h, 'LineStyle', '--');      % make the plotting line a dashed line. Other common
                                   options are '-' (solid line) and ':' (dotted).

>> set(h, 'Marker', 'o');          % mark each data point with a circle. Other common
                                   options are '+' and '*'.
```

A shortcut to set Color and LineStyle is:

```
>> plot(x, y, 'r--');              % plot y as a function of x in a red dashed line
```

Note that Matlab always tries to scale the plot axes so that you can see everything in the plot. You can override this by using the command "axis":

```
>> figure(1)
>> clf
>> plot(x,y)
>> axis([-2 12 -1 20])              % format is: [minX maxX minY maxY]
```

I think you should have the basic idea from here. Get comfortable by playing with options, typing "help xx" and using the examples in the help window.

Things to try:

```
>> x = [0:(pi/10):(4*pi)];       % x runs from 0 to 4π in intervals of 0.1π
>> y = sin(x)
>> figure(1)
>> clf
>> plot(x,y)
```

Using Matlab's Help

Matlab provides a lot of support that is actually quite helpful. You can access Matlab help in a number of ways:
1. By typing >>help *functionname* to get help on a particular function (e.g. help sum)
2. Open the Matlab help window by typing "helpwin" at the command prompt, by clicking the question mark at the top of the IDE, or by selecting "MATLAB help" from the "Help" menu.

You will be using the "help" command at the command prompt often, so start getting used to this early. For example, if you weren't quite sure of the inputs or the order of the inputs to the function "plot" you can type:

```
>> help plot
```

A description of the function and how to use it will appear. The Matlab help tool has much more information than the help at the command prompt, and it allows you to search for terms, peruse an index of topics, view demos, etc.

What To Do When Stuck

We fully anticipate that some parts of the Matlab projects will be difficult. Sometimes your code won't work, and you'll spend 30 minutes staring at it without luck. *This is normal*. Before asking the your TA, we recommend the following:
1. Exhaust your existing help resources. Many of these are listed on the class site, including the OLC Matlab Stock Answers and The Mathworks Web Site
2. Take a break and work on something else. When you spend a great deal of time on a piece of code, you often miss the same error over and over again. This is why it's important to start these projects early.
3. Athena consulting : OLC will not help you with your code, but they will help you if you are having difficulty saving files, using a text editor, or have other questions about the Athena Experience

When you ask a TA for help, please include:
1. A copy of the code in question
2. A copy of the error Matlab is giving you. Just saying "my code doesn't work" isn't helpful, and

This course makes use of Athena, MIT's UNIX-based computing environment. OCW does not provide access to this environment.

will delay you getting the help you need.
3. An explanation of the things you have tried. What web resources have you looked at already? What solutions have you tried? This will save your time and the time of the TA.

Your time is valuable! Learn to debug intelligently and ask useful questions so that we can help you be successful as quickly as possible!

Matlab Project 0 Assignment Details

**The goal of this assignment is to make a plot of voltage as a function of time.

The final plot should show all the data and have appropriate labels on each axis. The plot should also have a solid, black horizontal line at voltage = 0 to show the zero level. It should also show the maximum and minimum voltage with text next to that voltage indicating the actual value of the maximum voltage.

1. Create a folder for your Matlab work in this class. Every time you open Matlab to work on 9.02, navigate to your 9.02 Matlab directory.
2. Open Matlab and navigate to your 9.02 Matlab directory.
3. Open a new m-file. Save it as "project0_*yourlastname*.m" You will eventually be turning in this m-file. So at the end of the assignment you should be able to type the name of your m-file at the command prompt and your final figure with all labels, text, etc should be generated.
4. The first two lines of your m-file should be:

```
clear;
load data_project0.mat
```

5. Add the lines of code to your m-file that make your plot of voltage as a function of time. Your plot should include:
    - -x and y-axes labels
    - -appropriate title
    - -a black horizontal line at voltage = 0
    - -a red star at the maximum voltage, with text indicating the maximum value
    - -a red star at the minimum voltage, with text indicating the minimum value

**Possibly useful functions: figure, plot, xlabel, ylabel, title, text, max, min.
Hint: It is helpful to work at the command prompt to try out lines of code to make sure they are functional. Once you have a functional line of code copy it into your m-file.

6. Save your figure using the “File --> Save As” command from the figure men containing your plot. Name your figure the same as your m-file: "project0_*yourlastname*.pdf". Ideally, save your figure as a PDF document, but if for some reason that doesn't work, use a standard image format such as JPG or BMP.
7. Upload both your m-file and your plot to the class website.

7.3 Spike Detection

Matlab Project 1

GOAL: The goal of Matlab Project 1 is to write a simple function (a set of Matlab commands) that will allow you to take voltage data recorded near a neuron and determine the times that an action potential occurred in that neuron.

Introduction: What is Spike Detection?

Since systems neuroscientists are ultimately interested in how the brain takes some input (e.g. a visual stimulus) and consequently alters behavior based on that input, it is necessary that we observe neurons while they are communicating. Action potentials, or "spikes", are the units of communication between neurons. Thus, the times when action potentials occur is the fundamental neuronal measure of most of neurophysiology experiments, especially *in vivo*.

But what actually is a spike? Remember, that ions are flowing in and out of the neuron to generate the action potential. Thus, there is a large voltage change that we can record if we place an electrode near a neuron. So the data we collect when neurophysiology experiments are performed is voltage measurements on the electrode measured at regular time intervals. In order to determine when the neuron we are recording from fired action potentials, we must analyze the voltages and come up with some analytical way of deciding at what times spikes occurred. This analytical process is called "spike detection".

In this project, you will be using Matlab to detect when spikes occurred in a data set of voltages we have given you. But eventually, you will be collecting your own voltage data from the fly visual system and you will need to use the ideas and code you develop here to help you detect the spikes in your own data set.

The Basics of Matlab Functions

To make your spike detection routine, you will need to learn how to put Matlab commands together into a function (saved as an “m file” in Matlab) so that you can later run that function on any set of data.

In the first Matlab lecture, you had some practice with functions, so I will not go into much detail here. I will just remind you of the basic structure of a function:

```
function [output1, output2]=my_function_name(input1, input2)
%comments describing the function (purpose, inputs and outputs, etc)

        <<my command/call 1>>
        <<my command/call 2>>
        …
        output1 = ??;
        output2 = ??;
```

Some things to remember:

-Always name your m-file the same as the name of your function.

-Any variables you use in the body of your function need to be created inside the function since functions use their own workspace.

-You need to assign values to your output variables somewhere in your function.

Matlab Project 1 Assignment Overview

**The goal of this assignment is to write a function that can do basic spike detection, and then to assess the performance of your spike detector graphically as well as quantitatively.

In this assignment you will:

1. Write a Matlab function that detects spikes from voltage data.
2. Write a Matlab function that creates two plots of your spikes: one of the complete voltage data with detected spikes marked, and one of the waveforms of all spikes detected.
3. Use the "testSpikeDetector" function we've given to you to quantitatively assess the performance of your spike detector and create a table of its performance on all four data sets.

The performance goal for your spike detector is: PCD>90% and FAR<2 spikes/sec on all data sets.

Step 1: Using the Poor Spike Detector

The first step to any data analysis is to understand the data you are analyzing. So, load the first data set into the workspace (remember: you need to be in the directory where the data is saved!):

```
>> clear all;
>> load('data01');
>> whos
  Name                   Size        Bytes      Class

  actualSpikeTimesMS     1x10        80         double array
  timesMS                1x20001     160008     double array
  voltagesUV             1x20001     160008     double array

Grand total is 40012 elements using 320096 bytes
```

Each of the four data sets contains these three variables. Here is a description:

Variable	**Contents**
actualSpikeTimesMS	list of times that spikes occurred (units are msec)
timesMS	list of all the times (in msec) that a voltage was measured
voltagesUV	all of the measured voltages (units = microvolts =uV)

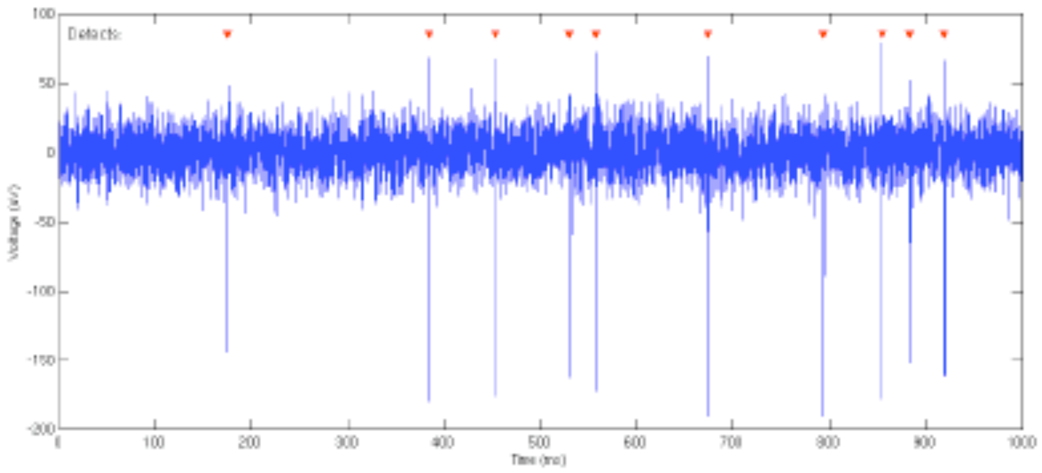
Try plotting voltage versus time to get a sense of the data. The spikes should be clearly visible.

Now let's try using the poor spike detector function that we have written for you to detect the spikes in this data set. This function is not a very good spike detector (we want to leave that job to you!), but it will give you a basic structure for how your routine should look. Note, the routine does not use the actual spike time (that would be cheating!). To run this function, you type its name and provide the input data and a variable to hold the output data. Here is an example of what to type at the command line:

```
>> detectedSpikeTimesMS = poorRoutineToFindSpikeTimes(timesMS, voltagesUV);
```

Step 2: Plotting Your Spikes

Now that we have a variable, detectedSpikeTimesMS, that contains the times of detected spikes we should try plotting these spike times on our voltage versus time plot to see how well the spike detector performed. The plot we want to make will look similar to the following figure:

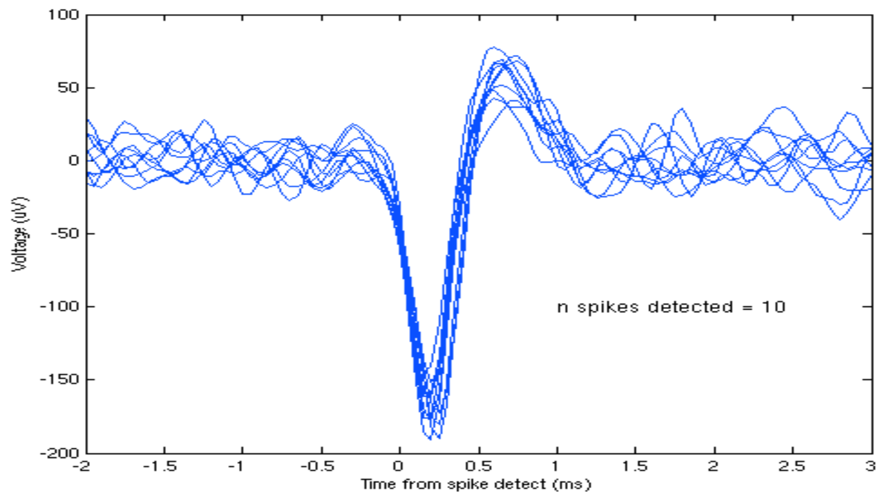


Since we will want to make a plot like this many times to test our different data sets and to test the different iterations of our spike detector, let's write a short function to create this plot. The function should be structured as follows:

```
function spike_plots(timesMS, voltagesUV, detectedSpikeTimesMS)

figure; hold
on; plot(
......  );
plot(......  );
```

This function requires three inputs: the raw data of times and voltages as well as the times of detected spikes. All the spike_plots function does is open a new figure window, turns hold to on to allow you to plot multiple things on the same set of axes (type "help hold" to get information about this function) and then plots your data (you need to fill in these lines with the specifics of what you want to plot).

You will also need to plot the individual waveforms (basically just a zoomed in version of the voltage versus time plot around the time of spikes) of all spikes in order to make sure they all look similar. The plot will look similar to the following figure:



One way to create this plot is to write a loop that plots a single waveform with each loop. Thus, the loop will be structured as follows:

```
figure;
hold on;
num_spikes = ?              %how do you determine the number of detected spikes?
for i = 1:num_spikes
        plot( ...... );     %where you need to plot the individual spike
end
```

Writing a loop like the one above is certainly not the only way (and it is certainly not the most efficient!) to create this plot, so if you have a better idea feel free to try it out!

Step 3: Assessing the Performance of Your Spike Detector

Once you have graphically looked at how the poor spike detector worked, you can assess its performance analytically by using a function we've given you, called testSpikeDetector. This function calculates two measures of performance:

1. The percent of spikes that are correctly detected (PCD) = the percentage of spikes correctly detected (A correctly detected spike is defined here as one with a reported time within +/-2 ms of the actual spike onset time.)

2. The false alarm rate (FAR) = the number of spikes reported per second that did not actually occur.

To run this function and obtain the values of these measures, you can type:

```
>> [PCD,FAR]=testSpikeDetector(detectedSpikeTimesMS, actualSpikeTimesMS);
```

Clearly, the ideal detector has a PCD = 100% and a FAR = 0 spikes/sec. When you write your own spike detector, you should be able to obtain the PCD > 90% and FAR<2 spikes/sec for each of the data sets provided (without changing the routine between runs of different data of course!!). How well did the poor spike detector do?

Step 4: Developing Your Spike Detector

Now that you have seen what a poor job the spike detector we gave you actually does, you need to write your own spike detector! There are two options for getting started:

1. You can use the file called “poorRoutineToFindSpikeTimes.m” as a starting point. To do this, open this file and then save it with a new name.

2. You can open a new m-file and just start typing. Go to "File", "New" in the Matlab IDE. Be sure to save your file with a ".m" extension!

Whichever option you chose, the structure of your function will look like the following:

```
function [spikeTimesMS]=myFindSpikeTimes(timesMS,voltagesUV)      % line 1

        <<my command/call 1>>                                     % line 2
        <<my command/call 2>>                                     % line 3
        …
        spikeTimesMS = ??;                                        % line 4
```

If you are already familiar with Matlab (or related programming languages), the skeleton provided above should be clear. If not, do not worry, we will walk you through it. Remember, you can always go to the Matlab help window to learn more about m files and functions in Matlab. Let's go through it line by line:

**There are four lines of code in this function skeleton, but that is just for illustration. Your function can have any number of lines. The only line that should be in the same place as this example is line 1.

Line 1. This is the most important line in the function. The word "function" tells Matlab that this is the start of a function. The variable name in brackets is the name of the output variable in the function; the values of this variable will be returned when the function is run. For example, when you called the function from the Command window, with the line:

```
>> detectedSpikeTimesMS = poorRoutineToFindSpikeTimes(timesMS,voltagesUV);
```

you asked Matlab to place the output of the function called "poorRoutineToFindSpikeTimes" into a variable called "detectedSpikeTimesMS". Note that you did not have to create this variable before you ran the function – Matlab created it for you. If you had created a variable with the same name before you ran the function, Matlab would have overwritten any data in your variable with the data output from the function.

The next part of line 1 after the "=" symbol is the name of the function. The name can be almost anything you want. However, any function you create should be saved as an "m file" with the same name.

For example, the function "poorRoutineToFindSpikeTimes" is contained in the file "poorRoutineToFindSpikeTimes.m".

The last part of line 1 (in the parentheses) contains the input variables. These are the names given by the function to the data that is provided by whoever called the function. In this case, the first data in the list is given the name “timesMS” and the second is given the name “voltagesUV”.

Note that a function can have any number of input arguments (or no arguments at all). In this case, the number of arguments is two. That does not mean that only two numbers can be passed in, because this function is meant to expect two vector arguments of any length (in the example above, the function accepted 20,000 numbers as input (10,000 numbers in each of the two vectors).

Line 2 and line 3 are meant to illustrate that you can do operations on the data, make calls to other functions, etc.

Line 4 indicates that, at some point in the function, you should assign some data to the output variable(s) of the function (in this case, to the variable called “spikeTimesMS”).

Spike Detection

Now that you have some understanding of the general problem and the structure of the function you need to write, your job is to come up with a mathematical way of detecting spikes from your voltages. Detecting spikes from voltage data is a non-trivial problem and research is still devoted to this problem today (for reviews on this topic, see the 9.02 class site). However, some basic methods of spike detection are straightforward and can be written especially well by MIT undergraduate students! Part of the purpose of this project is to let you think on your own about different ways that you might detect spikes in a continuous voltage trace. At the end of the day, you do not need a perfect routine, but we have provided some test data for your routine and it should be able to do a reasonably good job* (see below) of detecting the main spike signal in each of those test data sets.

You might want to try a few different algorithms before settling on your final spike detector. This is why we have given you the spike detection tester function and have had you write your plotting spikes function. Use these functions as you go to assess the performance of the spike detection algorithms you write.

**Remember, your goal is to create a spike detector that has PCD>90% and FAR<2 spikes/sec on each data set.

Potentially Useful Built-in Matlab Functions

One Matlab function that might be especially helpful to you is “find”. To get more information about find, type “help find” at the command prompt. “find” will return the indices of each element in a vector that meets a condition. For example:

```
>> x = [0 5 0 12 1 -4];

>> ind = find(x>2)            % find all the elements of x that are >2 and return the element
                                     number (i.e. index)

ind =                         % here are the element numbers (the 2nd and 4th elements of x)

    2    4

>> x(ind)                     % here is how you can easily see the values of all those elements

ans =

    5    12
```

Other math-related functions:
max, min, diff

Plotting functions:
plot, hold, xlabel, ylabel, title, text, scatter, axis

The "size" function returns the size of a variable, and can be used to find the length of a vector or size of an array. The following will assign to "m" the number of rows, and to "n" the number of columns, in a matrix A:

```
>> [m,n] = size(A)
```

The "length" function works similarly for vectors. These can be very useful for debugging when a matrix operation or plot command is producing an error.

Flow Control

We briefly met the "for" command. The syntax there is:

```
>> for i = 1:5
        statement
        statement
        ...
    end
```

The value of "i" can be accessed within the loop (for example, as an index into an array). It's best not to change the value of "i" within the loop.

Another useful construction is:

```
>> if x==0
       statements
   elseif x<0
       statements
   else
       statements
   end
```

The "elseif" and "else" portions are optional, and there can be multiple "elseif"s. Note that the double equal-sign ("==", pronounced "is equal to") is used when testing for equality; the single equal sign is used for assigning a value to a variable.

Remember, you can always get information about how to use a built-in Matlab function by typing "help *functionname*" at the command prompt. If you don't know the name of a function, but you think Matlab may have a function that does what you need, open the help window and do a keyword search.

Matlab Project 1 Assignment Details

Please upload the following to the class website:

1. The m-file that contains the code of your spike detector function.
2. The m-file that contains the code for your spike plotting function.
3. A text document (doc or pdf file only please) that contains:
    - a table of the performance of your spike detector (goal: PCD>90% and FAR<2 spikes/sec for each data set)
    - a plot of voltage vs time for *one* of the data sets with symbols marking times when spikes were detected
    - a plot of the waveforms of detected spikes from the data set you plotted in 2)

7.4 Movie Creation

Matlab Project 2

GOAL: The primary goal of Project 2 is to teach you the basics of designing visual stimuli and to make sure that you know how to use Matlab to build a basic movie.

A Brief Introduction to Designing Visual Stimuli

The “magic” of sensory systems results from to their remarkable ability to transform raw sensory representations of the world into profoundly more useful forms of representation. For example, our ability to perceive the visual world begins with two-dimensional patterns of neuronal activity on our retina (i.e. like the images in a camera). Those patterns of neuronal activity are then transformed in the nervous system to new forms of representation that are more closely related to our behavioral and cognitive goals. For example, neurons in the highest levels of the visual system are highly selective for specific visual objects (e.g. faces), but are tolerant to variables such as object position, size, pose, illumination, etc. The population of neurons at these higher levels of the nervous system contains much of the same information that entered the system at the retina (after all, visual information that does not pass through the retina cannot be created, only destroyed), but the information is in a different format. That is, it is *re-presented* in a different manner than the retina. This kind of representation is vastly more useful for behavior and memory than the original neuronal representation in the retina. Creating such useful forms of representation is the reason that our sensory systems have so many neurons working on this problem.

Given this, one of the primary goals of systems neuroscientists working in the sensory systems is to understand how neurons at each level of the system have recoded (i.e. transformed) the image data that impinges on the sensory system of interest (e.g. the retina, the cochlea, or the skin surface). From the point of view of any particular neuron under study, this means trying to figure out how that neuron transforms its sensory inputs to its output pattern of neuronal activity. Which sensory stimuli does it “like” (big response) and which stimuli does it “not like” (poor response)? Can we understand the rules that govern this behavior? The first fundamental concept in this regard is called the “receptive field” – the region of sensory space to which the neuron responds. But this is just the beginning of understanding how a neuron transforms the sensory information, the real magic is in understanding and describing its response to (ideally) any sensory pattern of input. This is a daunting task, and there is much, much more work to be done in this regard in all the sensory systems.

In the fly laboratory, we will introduce you to the basic techniques that are used to assess the sensory function of neurons in the visual system. As a model, we will use the fly visual system because some neurons in the fly visual system are relatively easy to record from and have very interesting response properties.

As you know, the fly is often flying! As you might imagine, navigating around when you are quickly flying and turning in the real world is a very challenging task and it is far from understood how the fly achieves such grace in the air. It is clear however that navigating requires an assessment of motion across the retina because this can be used to determine heading and turning velocities. It is thus perhaps not surprising to learn that the fly has many neurons that are very sensitive to the direction and speed of visual motion across their eyes. As you might also imagine, visual sensitivity to motion is not something restricted to flies but is a common feature of visual systems. Indeed, some of the most elegant recent work on the visual processing in the primate visual system has been carried out in areas that are very sensitive to motion and motion direction (e.g. area MT and MST, you will see some of this in your recitation sections).

Besides its importance in vision, motion sensitivity is one of the properties where computational approaches have made great strides in proposing possible mechanisms that could be used to compute motion (you will see some of this in another recitation section). Many of those predictions have been and are being tested in visual systems including the very same flies that you will use in the fly lab.

In your team project, we give you the opportunity to design your own visual stimuli to test the response properties of neurons that you will record in your fly. That is, you will get to design your own visual experiment (i.e. visual stimuli), collect neuronal responses to those stimuli, try to interpret the results, and describe what you find in your lab report. In this Matlab project, we will introduce you to basic visual tools at your disposal. More importantly, *we hope to give you an understanding of the flexibility of Matlab in building such movies so that you will be ready for your team to think outside the box and design any visual stimuli of your choosing.*

Matlab Setup for this Project

**Note: Try and do this project on a fast machine (i.e. not one of the SunBlade 100s) as we will be working with large video files, and the glacial pace of Matlab on this older hardware will make you very sad.

1. Once you have Matlab open and you've navigated to the proper directory (the directory where you have saved the Matlab Project 2 files), you need to run the "init" function.

```
>> init
```

2. All the "init" function does is add the MoviePlayer folder to the list of folders that Matlab knows about. To test that this was completed successfully, and everything that you will need is ready to run, try the following command:

```
>> help showMovieCube
```

You should see a description of what the showMovieCube function does. This is one way to know that Matlab found this function on its "path" of known directories.

If you instead get some kind of error message, then go back and repeat steps 1 and 2 above.

Matlab Project 2 Assignment Overview

In this assignment you will create an 10-second long 320x240 visual movie to test the basic directional selectivity of a fly visual neuron. You will do this by creating a Matlab function, *generateMotion8Movie.m*, which takes no arguments and returns your movie cube. *Specifically, we want you to generate a movie that shows eight different directions of visual motion even spaced around the "clock", with blank recovery periods in between the presentation of each direction.*

Matrix Indexing

Matlab runs most efficiently when working with matrices (the MAT in Matlab means MATrix). A matrix is a grid of *m* rows and *n* columns of numbers. In Matlab, vectors are just matrices with either a single row or column (i.e. either m= 1 or n = 1).

Here are some ways of creating new matrices:

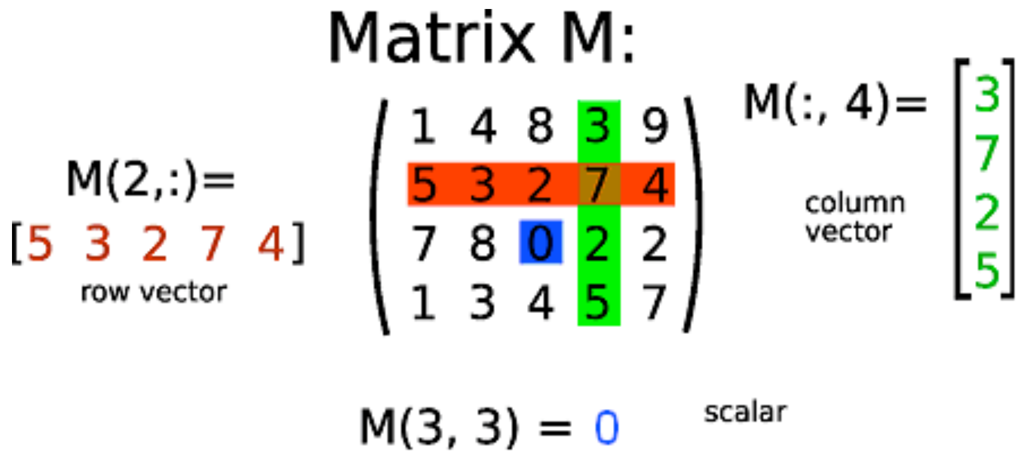
- -zeros(m, n): creates an m x n matrix of all zeros
- -ones(m, n) : creates an m x n matrix of all ones
- -rand(m, n) : creates a random m x n matrix of values between zero and one

In addition, we know that you can generate a range vector using the colon operator, such as:

```
>> 1:3
ans =
   1    2    3

>> 1:2:10
ans =
   1    3    5    7    9
```

There are a number of ways of accessing the elements within a matrix, just like with vectors:

- -One element at a time. To access the element in matrix M in the **i**th row, **j**th column, use M(i, j)
- -The : operator means "use all possible index values". For example, to access the entire **j**th row of matrix M, we use M(j, :) which tells Matlab "give us all elements where the row is equal to j"



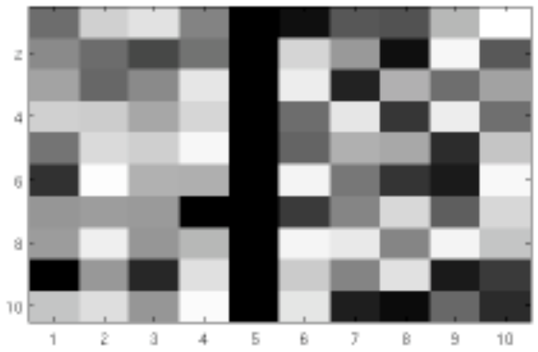
Images as Matrices

An image on a computer screen is a matrix of pixels; each entry in the matrix contains the color for one dot on your screen. For our experiments, all of our images will be monochromatic, so we adopt the following convention:

> "If a pixel's value is 0, it is black; if it is 255, it is white; intensity maps linearly from 0 to 255[1]"

For example, a value of 128 would be 50% gray. Below is an example of a small matrix and its associated image. Note that images have their upper-left-hand corner at (1, 1).

```
>> x

x =

   73  185  212   94   0   13   54   51  153  255
  101   72   43   78   0  193  117   13  243   55
  129   67  101  218   0  229   21  145   74  128
  185  181  133  194   0   73  218   32  227   75
   79  200  184  243   0   65  144  134   26  172
   29  252  146  143   0  238   82   30   17  245
  114  121  118    4   0   34   96  197   60  196
  120  231  114  153   0  240  222   96  238  170
    4  116   23  209   0  179   95  210   17   34
  170  206  114  250   0  217   19   12   68   25

>> image(x)
```



[1] [1] Why 255? Computers represent numbers in binary, and manipulate them in chunks of bits called bytes. A single byte contains eight bits, letting a byte represent values between 0 and 255 (or 2^8=256 total different values). We could of course use more than one byte per pixel, but this would double the amount of memory needed and it's unlikely our monitor can display that many different values of gray anyway.

Let's try creating a simple image. This will be a small image, 100 by 100 squares. First, create a black background:

```
>> myImage = zeros(100, 100)
```

To show a matrix as an image in Matlab, we use the image() function.

```
>> image(myImage)
```

Since the image is all zeros, it should be pretty boring. Now, let's make a single pixel white in our image. Remembering that 255 is the brightest white value, we'll set

```
>> myImage(50, 50) = 255
```

Remember, that we can access a row of a matrix by using the colon; we can set the 9th row of the image to 50% gray with

```
>> myImage(9, :) = 128
```

Look at this image to make sure it looks right. Similarly, to set a whole block equal to a color, we use range operators for both rows and columns to set the entire lower-right-hand quadrant of the image to white.

```
>> myImage(50:100, 50:100) = 255
```

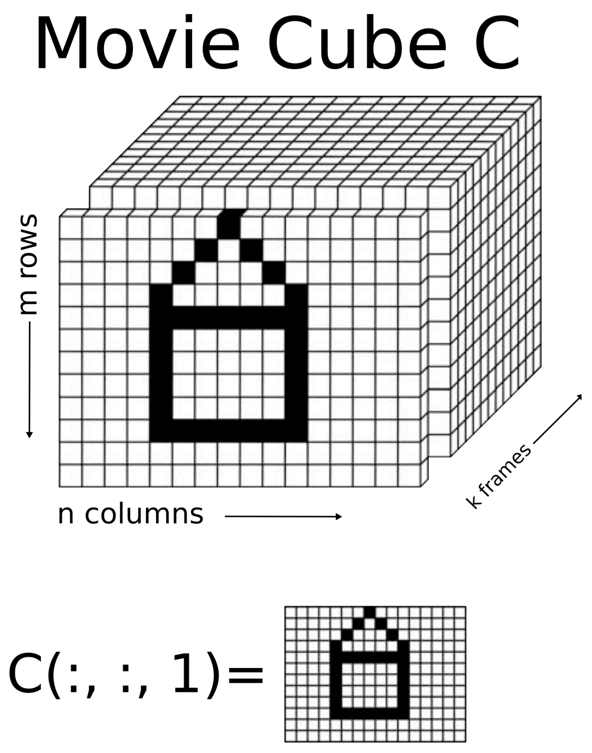
Movies as 3-Dimensional Matrices

A movie is just a series of images shown in rapid succession; each image is called a frame, and they are rapidly shown at some frame rate. If the frame rate is high enough, the illusion of motion is created.

The movies at LSC are shown at 24 frames per second (FPS); for historical reasons your TV shows images at 29.97 FPS. **For our visual stimuli, we'll be showing images at 30 FPS**. This means that a new picture is shown every 33.3 ms.

Here's where it gets interesting; just as a vector is a 1-dimensional array of numbers, and a matrix is 2-d, Matlab can work with 3-dimensional numerical objects, which we'll call "cubes"[2].

[2] They're not really "cubes", because they have different dimensions along each axis. They would more accurately be called "hypermatricies", but that's a mouthful.



Similar to above, we can create a 12x16x5 rectangular cube of numbers with

```
>> MAXX = 16
>> MAXY = 12
>> myMovieCube = zeros(MAXY, MAXX, 5)
```

Here, though, we can view a cube as consisting of a series of matrices, one after another. Each matrix is a frame in our movie. We can view the first frame with:

```
>> image(myMovieCube(:, :, 1))
```

Again, this is going to be a bit boring. So we'll spice it up by adding some lines:

```
>> myMovieCube(:, :, 1) = 255                   % first frame all white
>> myMovieCube(4,:, 2) = 255                    % a horizontal line on the second frame
>> myMovieCube(6,:, 3) = 255                    % a horizontal line on the third frame
>> myMovieCube(8,:, 4) = 255                    % a horizontal line on the fourth frame
>> myMovieCube(:,:,  5) = rand(12, 16)*255      % a random matrix for the last frame
```

A few points to note in the above code:

-We work with our movie cube by assigning values to the pixels within a frame
-Each line sets some pixels in a single frame
-Each frame is manipulated just like an image; the commands for drawing lines are identical.
-Note how the position of the horizontal line changes slightly from frame to frame; when you view the cube, you'll see the line appear to move
-The last frame is set to random noise; remember that random generates a m x n matrix of values between zero and one; we then scale the pixel values by 255. This is a "white noise" frame (more about that below).

Viewing your Movie

Now that you have your movie cube, you'd probably like to watch it as a... movie. To do this, we've provided you with the showMovieCube function, which will step through each frame in the movie and show it to you.

Example:

```
>> showMovieCube(movieCube, FPS)
```

where FPS is the number of frames per second (which is normally 30). To watch your movie cube very slowly, try lowering FPS to 1-5.

Helper Functions for Interesting Visual Stimuli

There are many common stimuli used by researchers to probe the workings of visual systems. We've written helper functions to let you quickly generate the most common ones. Each of these returns some set of frames, which you can place into your movie cube.

Repeated below are the help file strings for the complex stimuli functions. You may decide to use some of these functions for your team project movie as well. For the functions with motion the angle conventions are as follows:

```
% drift angle conventions (deg):
% 0 deg is drift upward
% 90 deg is drift rightward
% 180 is drift downward
% 270 is drift leftward
```

Spot of Light

```
function [movieFrames] = spotOfLight(maxY, maxX, nFramesToMake, backgroundGray, posY,
posX, diameter, spotGray )

% Will make a circular spot of "light"  (can be black, white, or any gray level)
% The spot will stay on for the number of indicated nFramesToMake
%
% parameters:
% maxY = number of pixels along row dimension (vertical dimension of each frame)
% maxX = number of pixels along column dimension (horizontal dimension of each frame)
% nFramesToMake  = number of frames of this movie to make.
% backgroundGray = gray level of the background, range: 0 (black) - 255(white)
% posY = the vertical location of the center of the spot (0 is top)
% posX = the horizontal location of the center of the spot (0 is left)
% diameter = diameter of the spot in pixels
% spotGray = gray level of the spot (0 (black) = 255 (white))
```

Drifting Bar

```
function [movieFrames] = driftingBar(maxY, maxX, nFramesToMake, backgroundGray, angle,
barWidthPix, barLengthPix, barGray, speed_PixPerFrame)

% Will make a single bar that drifts across the screen in a direction indicated by "angle".
% The bar length is perpendicular to the direction of drift.  The bar is
% windowed by a circular apeture of diameter min(maxX,maxY).  The output values in each
frame
% range from 0 (black) to 255 (white)
%
% parameters:
% maxY = number of pixels along row dimension (vertical dimension of each frame)
% maxX = number of pixels along column dimension (horizontal dimension of each frame)
% nFramesToMake  = number of frames of this movie to make.
% backgroundGray = gray level of the background, range: 0 (black) - 255(white)
% angle = drift angle of the bar (i.e. the direction of motion)
% barWidthPix = the width of the bar in pixels (min = 1)
% barLengthPix = the length of the bar in pixels (min = 1)
% speed_PixPerFrame = number of pixels the bar drifts on each frame
```

Drifting Sinusoid

```
function [movieFrames] = driftingSinusoid(maxY, maxX, nFramesToMake, backgroundGray,
angle, spatialPeriod_pixPerCycle, speed_PixPerFrame, contrast)

% Will make a planar sinusoid that drifts in a direction perpendicular to the "lines" in the %
pattern.  The sinusoid will always be presented in a circular apeture of diameter %min
(maxX,maxY).  The output values in each frame range from 0 (black) to 255 (white)
%
% parameters:
% maxY = number of pixels along row dimension (vertical dimension of each frame)
% maxX = number of pixels along column dimension (horizontal dimension of each frame)
% nFramesToMake  = number of frames of this movie to make.
% backgroundGray = gray level of the background, range: 0 (black) - 255(white)
% angle = drift angle of the sinusoid (i.e. the direction of motion)
% spatialPeriod_pixPerCycle = number of pixels covered by each cycle of the
%   sinusoid  (this is basically how "thin" or "fat" the lines are)
% speed_PixPerFrame = number of pixels the sinusoid drifts on each frame
% contrast = the depth of modulation of the sinusoid.  (1 = full modulation
%   from black to white, 0 = no modulation = gray screen)
```

Spot Of Motion

```
function [movieFrames] = spotOfMotion(maxY, maxX, nFramesToMake, backgroundGray,
posY, posX, diameter,  angle, spatialPeriod_pixPerCycle, speed_PixPerFrame, contrast );

% Will make a circular spot of "motion" (drifting sinusoid) centered at posX, posY.
% The spot will stay on (and in motion) for the number of indicated nFramesToMake
%
% parameters:
% maxY = number of pixels along row dimension (vertical dimension of each frame)
% maxX = number of pixels along column dimension (horizontal dimension of each frame)
% nFramesToMake  = number of frames of this movie to make.
% backgroundGray = gray level of the background, range: 0 (black) - 255(white)
% posY = the vertical location of the center of the spot (0 is top)
% posX = the horizontal location of the center of the spot (0 is left)
% diameter = diameter of the spot in pixels
% angle = drift angle of the sinusoid (i.e. the direction of motion)
% spatialPeriod_pixPerCycle = number of pixels covered by each cycle of the
%   sinusoid  (this is basically how "thin" or "fat" the lines are)
% speed_PixPerFrame = number of pixels the sinusoid drifts on each frame
% contrast = the depth of modulation of the sinusoid.  (1 = full modulation
%   from black to white, 0 = no modulation = gray screen)
```

Matlab Project 2 Assignment Details

In this assignment you need to create a 10-second long 320x240 visual movie to test the basic directional selectivity of a fly visual neuron. You will do this by creating a Matlab function which takes no arguments and returns your movie cube. Your movie should have the following properties:
1.Your movie will show eight different directions of visual motion evenly spaced around the "clock", with blank recovery periods in between the presentation of each direction.
2. The duration of each motion direction stimulus should be 500 ms and the duration of each recovery period should be 500 ms.
3. The movie should begin and end with 1 second of blank space
4. For visual 'motion' you can use drifting bars, drifting gratings, or anything moving in a single direction (you can use our helper functions or write your own!).
5. Have your visual stimulus run at 30 frames per second.
6. The luminance of the background should remain at one value for the entire movie. (Can you guess why this might be important?)

Submission: Upload your m-file to class site that contains the function to create your movie. These will be graded by running your function and playing your movie, so make sure running your function by typing in the name of the function at the command prompt successfully creates your movie!

Starting your movie creation function:
In your movie creation function, you'll want to start off with something like:

```
    totalStimulusLengthSec = 10          % stimulus length in seconds
    FPS = 30                             % number of frames per second
    MAXX = 320
    MAXY = 240
    myMovieCube = zeros(MAXY, MAXX, FPS * totalStimulusLengthSec, 'uint8')
```

Then you should have a series of lines setting frames in your movie to display your motion stimuli. Feel free to use a loop if it would make your life easier!

7.5 Data Analysis

Matlab Project 3

GOAL: The primary goal of Project 3 is to teach you the basics of analyzing neuronal data. Specifically you will be creating raster plots, histograms, and a tuning curve of spike data.

Introduction to Neuronal Data Analysis

The goal of this project is to teach you the basics of how systems neuroscientists attempt to make sense of neuronal data (recorded during an experiment). What does 'make sense' mean? Suppose we have neuronal voltage data from which we have extracted the spike times of an individual recorded neuron. In the ideal world, 'make sense' would mean being able to explain (and thus understand) why every spike occurred when it did. That is, we could completely explain the observed list of spike times. This level of explanation is rarely (if ever) achieved in real experiments, but it gives you a way to think about the goals of neuronal data analysis. Clearly, this will usually require an understanding of all the environmental variables (e.g. stimuli) that might have produced those spikes and perhaps knowledge of variables that one may not have easy access to (e.g. the attentional state of the animal, the exact temperature of the neuron, the phosphorylation of some proteins in the neuron, etc, etc.). The inability to exactly know all variables that might influence the observed response (both external and internal variables) is almost surely the reason that we cannot hope to explain every single detail about the spike times observed in the data, yet, we can still work in the face of this 'noise' (that is what we call variability in the neuronal response that is not reproducible and thus we cannot yet explain) and try to understand the effect of variables that we do have access to or control of (e.g. the visual stimulus).

OK. This is a good start for thinking about the problem, but how does explaining the spike lead to understanding? Thinking about such questions quickly becomes almost philosophical, and one could have a great graduate seminar on this topic alone. For now, let's just concentrate on what actually happens in neurophysiology labs. In essentially all neuronal experiments, the goal is far less lofty than explaining each and every spike recorded from each neuron. Instead, scientists often ask much more targeted 'hypothesis-driven' questions about neurons. A 'hypothesis-driven' question is basically a question that has a 'yes' or 'no' answer. (e.g. Does neuron X response more strongly to stimulus A than to stimulus B? Does neuron X change its response to stimulus A if I give drug Y?, etc.). In general though, a good 'hypothesis-driven' question is also motivated from some underlying theory about how things are working and the answer to that question will help separate the (sometimes many) competing theories about what is going on (i.e. the scientific method). This of course depends on the field of research and makes it clear why scientists cannot work in a vacuum – they must be in touch with the prevailing hypotheses and theories in their field. It also makes it clear that it is almost impossible to ask good hypothesis-driven questions in an area of research where one has little idea of what is going on and theories have not yet had time to develop, and one must generally start research in such new areas by trying to describe what they find in an objective, quantitative manner.

Now, if you are following closely, you may notice that 'hypothesis-driven' questions can also be thought of as just 'simpler' questions aimed at the initially stated 'big goal' – understanding all the spikes. For example, suppose one asks the hypothesis driven question: 'Does fly neuron H1 response better to visual motion of a bar of light in the front-to-back direction or the back-to-front direction?' Now suppose one then tests (e.g.) eight directions of bar motion (instead of just two). You can imagine a whole set of 'yes-no' question in that experiment, but the most concise way to describe the data is to simply show a summary of all the data. In this case, a plot might show the average response to each of the tested directions and perhaps a curve fit to predict the response (on average) that would occur to ANY of the intermediate directions that were not tested (see figures below). This plot is called a 'tuning curve' (for direction of bar motion) and such analyses are very common in sensory physiology (see below). Note how this has taken us from a simple hypothesis-driven question to a fuller understanding of spikes that might occur to ANY direction of bar motion. Is this the full understanding that we outlined at the beginning of this discussion as the big goal? Of course not. From this data and analysis, one cannot tell how the neuron would respond to arbitrary stimuli (e.g. a face, white noise, a dog, etc..). The experiment does not even tell us if the tuning curve would change if we used other types of motion stimuli (drifting sinusoids, drifting dots, etc.). Nevertheless, it is progress, and would rule out any theories of the fly visual system that predict that this neuron would show tuning largely different than that actually observed.

As a side note, it is important to understand that one cannot hope to test and understand all possible stimulus conditions. There are just too many possibilities! Instead, one typically seeks a level of understanding that *generalizes* across similar conditions (e.g. that the tuning of H1 is really for 'motion' and not something special about a bar of light). Similarly, one does not need to measure the response to 1 deg and 2 deg of visual motion if the tuning varies slowly and 0 deg and 45 deg have already been tested. Given that we cannot present all possible stimuli, the two dominant strategies are to: 1) start with simple stimuli that are the 'building blocks' of more complex stimuli (e.g. small spots of light, sinusoids, etc.) and 2) use stimuli that the organism must often deal with in the real world (i.e. 'behaviorally relevant' stimuli). Strategy 1 has been the dominant approach for several decades in sensory physiology and much of our current knowledge was derived from this approach. However, it does not help create a generalized understanding when neuronal response become non-linear functions of the stimuli (because one cannot use the neuronal response to the simple 'building block' stimuli to predict the responses to more complex stimuli). The second strategy ('behaviorally-relevant' stimuli) has come into favor among many recently; some nice discussion of the advantages can be found in the Egelhaaf fly readings listed online. In reality, these two strategies are not mutually exclusive and both are continuing to contribute to our understanding of neuronal systems.

**For this tutorial, your goal is to analyze the data given to you (simulated fly neuronal recording data), and determine a tuning curve for direction of motion. As described in the above discussion, such analyses form the beginnings of a fuller understanding of what a neuron does and how it contributes to the behavior of the organism.

Matlab Project 3 Assignment Overview

The overall goal of this project is to plot a tuning curve for motion direction using neuronal data and movie information supplied for you. Along the way, you will also plot spike rasters and spike histograms for the whole movie and for each of the motion conditions presented in the movie.

(For your lab report, you will perform similar analyses on your actual neuronal data. The data you will analyze in this project is simulated neuronal data, but the analyses will be very similar for your actual data.)

Step 1: Examine and Understand the Movie Presented to This Fly Neuron

The function used to create the movie to test the 'neuron' in this project is available to you. This function is in a file named makeBasicMovieCube.m.

```
>> mc = makeBasicMovieCube();
>> showMovieCube(mc, 30);
```

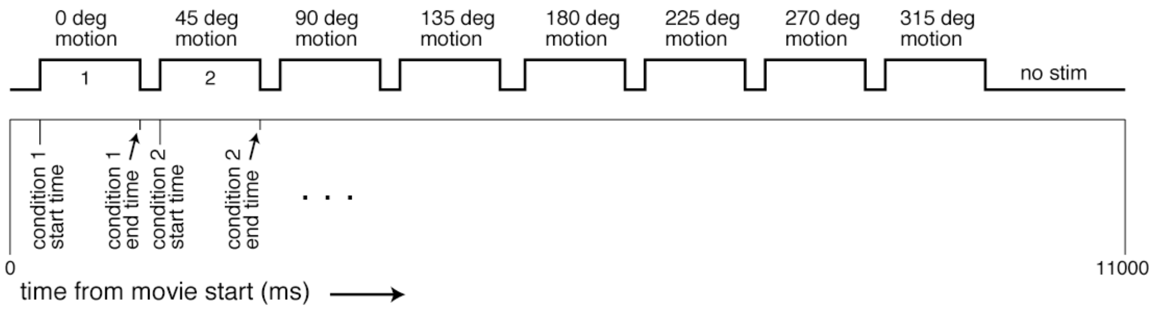
You can read the movie function by opening it.

There are extensive comments in the function describing how the movie was created. To help you out, we also provide a list of times when each 'condition' occurred in the movie:

Total movie time = 11000 ms (11 seconds)
All bars were drifted at a speed of ~50 deg/sec.
Note that, for times when no condition is listed, the screen is black (no stimulus)

Condition	Motion direction (deg)	On screen display (drifting bar stimulus)	start time (ms)	end time (ms)
1	0	Upward	200	1200
2	45		1367	2367
3	90	Rightward	2533	3533
4	135		3700	4700
5	180	Downward	4867	5867
6	225		6033	7033
7	270	Leftward	7200	8200
8	315		8367	9367
9	No stimulus	NA	9533	10533

For this data (and the data you will collect in the fly lab), these times are all relative to the start of each presentation of the movie (and the movie was presented 10 times). The following figure depicts the structure of the movie:



Step 2: Load Voltage Data Recorded near the Neuron and Extract the Neuron's Spikes from All Ten Runs of the Movie

In project 1, you loaded two vectors (two long lists of numbers), where each vector was exactly the same length:

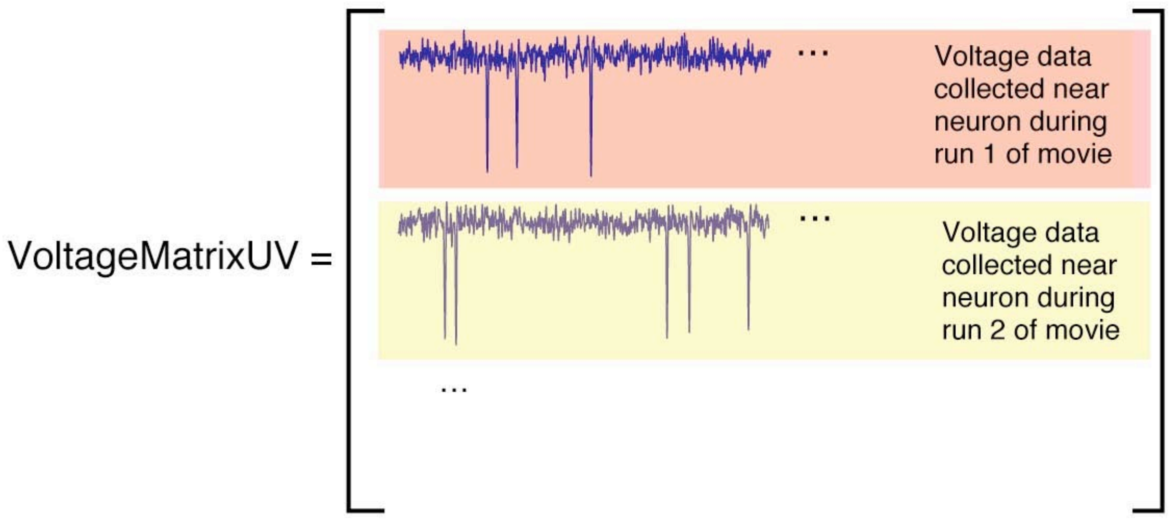
timesMS	was the list of times (in msec (MS)) when a voltage value was measured
voltageUV	was the list of voltages (in microvolts (UV)) measured at those times

In this project (project 3), the only difference is that, instead of one list of voltages, there are 10 lists of voltage values. These 10 lists correspond to the 10 simulated runs of the movie. (Side note: why do we even bother to run the movie more than once? Why not 100 runs or 1000 runs instead of 10 runs?).

Because each list of voltages has exactly the same number of elements (i.e. each list is the same length), they can be nicely stacked on top of each other to create a voltage matrix that holds all the data (called VoltageMatrixUV). (The UV means 'microvolts' to remind us of the unit of measure). This matrix has 10 rows (one for each run of the movie) and 220,001 columns (one for each time point during the movie when the voltage was measured by the Analog-to-Digital system). Because it is more convenient to record time from the start of each movie run (rather than (e.g.) time of day), we still only need one list of times (just as in project 1). This is illustrated in the figure below.

VoltageMatrixUV(1,:) = row 1 of matrix = Voltage data collected near neuron during run 1 of movie

VoltageMatrixUV(2,:) = row 2 of matrix = Voltage data collected near neuron during run 2 of movie


VoltageMatrixUV =
...
Voltage data collected near neuron during run 1 of movie
...
Voltage data collected near neuron during run 2 of movie
...

$$\text{VoltageMatrixUV} = \begin{bmatrix} 0 & 2 & 3 & 7 & 10 & 3 & -2 & 301 & 470 \ldots \\ 9 & 5 & 20 & -5 & 1 & 90 & 25 & 65 & 592 \ldots \\ \cdots \\ 67 & 56 & 23 & 7 & -3 & 89 & 645 & 30 & 4 \ldots \end{bmatrix}$$

$$\text{TimesMS} = \begin{bmatrix} 0 & 0.05 & 0.1 & 0.015 & 0.02 & 0.025 & \ldots & \ldots & 10999.95 & 11000 \end{bmatrix}$$

time from movie start (ms) $\longrightarrow$

We have provided you with data in a format identical to the format your final, recorded data will be in. This data has been acquired at a given sampling rate – this is the number of times, per second, that the computer records and digitizes the voltage waveform. In our example, the data has been sampled 20,000 times per second. Thus, we set SPS (Samples-per-second) to 20000. Similarly, our example data has ten runs, so we set numRuns to 10.

**Once you understand the format of the data that you will work with, go ahead and load the data.

Once you have loaded the data, you need to run your spike detector (project 1) on each row of the voltage data. Use the function that has been provided to you called 'spikeExtractorForVoltageMatrix', but you should first edit the function and put a call to your own spike detector (from project 1) in the function at the point in the function where it is needed.

You should resave the function with a new name such as: *MYspikeExtractorForVoltageMatrix*.

You can then run the function like this from the command line:

```
>> [spikeTimesMS] = MYspikeExtractorForVoltageMatrix(timesMS, voltageMatrixUV);
```

The output of this function is not just a single list of spike times (as you did in project 1), but 10 lists of spikes, one list for each of the 10 runs.

Because these are now lists of spikes times and neurons do not spike regularly (i.e. neurons are not clocks), it is highly unlikely that each of the 10 lists will contain the same number of elements (spike times). Thus, we cannot use a matrix to hold this data. Fortunately, Matlab has a data structure that is well suited for this purpose. It is called a cell array. Unlike the rows of a matrix, each element in a cell array can hold something totally different. In this case, each element in the cell array holds a vector listing the spike times for one run of the movie. To access the contents of a cell array, you need to use *curly brackets*:

```
>> spikeTimesMS{1}        % to see list of detected spike times for movie run 1
>> spikeTimesMS{2}(5)     % to see fifth detected spike time for movie run 2
```

Now that you have extracted spikes from the voltage matrix into organized lists of spike times (organized by movie run number), you are ready to do some basic analyses of the data.

Step 3: Plot the Rasters and Histogram for Each of the Nine Conditions

A raster plot is simply a display of all the spikes that were observed on repeated runs around some event. In the case of the fly lab, the event of interest is the start time of the movie or the start times of each of the conditions in the movie. In the case of the particular movie that you are working with for this project, the relevant 'events' are the start times of each of the motion direction conditions that were tested (see list above for these times). A histogram is just a count of all the spikes that occurred at different times relative to this event and is usually scaled so that the units are spikes/sec. A histogram is shown in the examples below. Note how the histogram depends on the bin size (the width of time used to count spikes across trials).

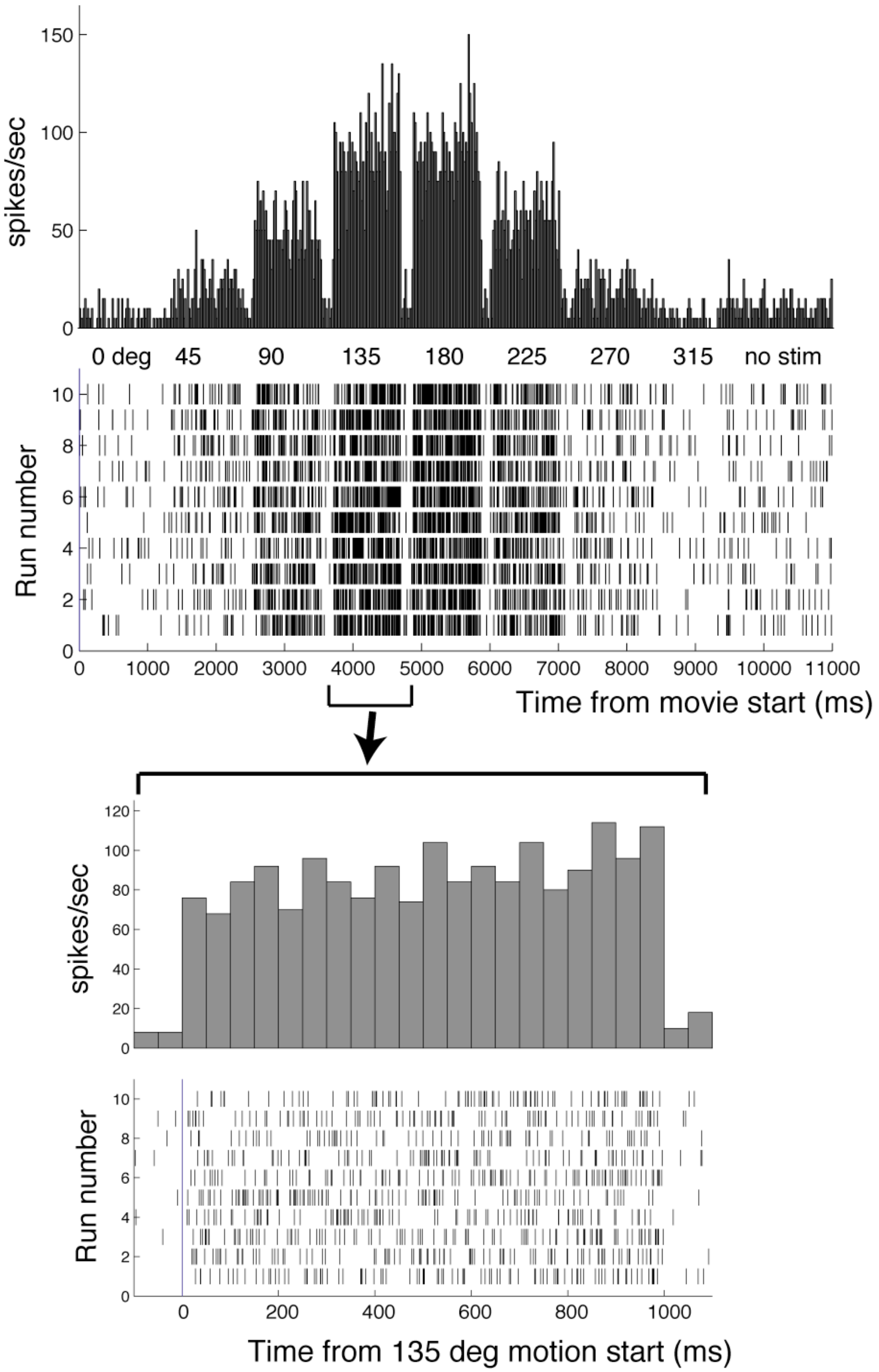
We have given you a function to help you make rasters and histograms.

```
>> plotSpikeRasterAndHistogram(spikeTimesMS, plotSynchTimeMS, preSynchDurationMS, postSynchDurationMS, widMS)
```

spikeTimesMS	A cell assembly of spike times, one spike time vector per condition.
plotSynchTimesMS	The time in the movie that will be called 0 on the plot (e.g. the start of a new condition)
preSynchDurationMS	The amount of time to show before time 0 (should be 0 or greater)
postSynchDurationMS	The amount of time to show AFTER time 0 (should be 0 or greater)
widMS	The width of each of the bins in the histogram -- try varying this!

You can type 'help plotSpikeRasterAndHistogram' at the command prompt for more info on how this function works.

We would like you to make a raster histogram of the entire movie (as illustrated in the example here). We would also like you to make a raster and histogram around each of the nine conditions for this movie (i.e. nine raster plots and nine histograms). You can use the same function with different arguments. Examples of rasters and histograms are shown below.


150
100
50
0
spikes/sec
0 deg
45
90
135
180
225
270
315
no stim
Run number
10
8
6
4
2
0
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
11000
Time from movie start (ms)
120
100
80
60
40
20
0
spikes/sec
Run number
10
8
6
4
2
0
0
200
400
600
800
1000
Time from 135 deg motion start (ms)

The plot on the top shows the spike data from the entire movie (synchronized on the time of movie onset). For this plot, the ‘plotSynchTimeMS’ was 0 (the time of the start of the movie), the ‘preSynchDurationMS’ was 0 (because there is nothing to show before the start of the movie) and the ‘postSynchDurationMS’ was 11000 (the total duration of the movie).

The plot on the bottom shows a raster for one of the nine conditions (condition number 4 in which motion was at an angle of 135 deg, see movie info above). Note that such rasters and histograms usually start a bit before the event of interest (e.g. motion onset time) so that the reader can see what the neuron was doing just before it was tested (with motion in this case). They often extend to just after the condition ends (as shown in the example). The arguments for the plot function allow you to control this. For this plot, the ‘plotSynchTimeMS’ was 3700 (the time of the start of condition number 4), the ‘preSynchDurationMS’ was 100 (because this allows us to see 100 ms of data just before condition 4 started) and the ‘postSynchDurationMS’ was 1100 (the total duration of condition 4 (100 ms) plus an extra 100 ms so that we can see how the cell returned to the ‘background’ firing rate at the end of the condition). Note that the amount of pre and post data that one shows depends on the particulars of an experiment (movie condition timings). It also depends on the time it takes the neuron under study to ‘recover’ from each condition – again, this depends on the details of the experiment and the preparation.

**NOTE: Careful inspection of the lower plot spike rasters reveals that the neuronal response does not start right at the time of motion onset (time 0 in the plot), but a little bit later. Any idea why this might be? Why does this not show up so well in the histogram?

Step 4. Determine the Average Neuronal Response in Each Condition

Think about a single presentation of one condition. In this case, the duration of each condition is 1000 ms. Often, we see that many spikes occur during this single presentation. How can we collapse this short list of spike times (the spike times during each movie run of a single condition) into a single measure of neuronal ‘response.’ There are many possible measure of neuronal response, but one of the simplest (and most widely used) is just the total number of spikes that occurred during the condition (units: spikes). This count is usually normalized (divided by) the total time of the condition to give a spike rate (units: spikes/sec). Typical neuronal firing rates are 0 to 100 spikes/sec, but they can be higher. For each presentation of each condition, you should determine the neuronal response (spikes/sec).

Here is a function to help you compute the response to each presentation of each condition. It simply counts the number of spikes in the interval that you indicate and divides that number by the duration of that interval so that the answer is returned to you in the standard units of spikes/ sec, regardless of the length of the interval. It is very simple. You are of course free to write your own function.

```
>> a = neuronalResponse(startTimeMS,endTimeMS,spikeTimesMS{run})
```

- 'a' is the response to the presentation of one condition that started at startTimeMS and ended at endTimeMS.
- 'run' is the repetition number (typically 1,2,3,…10)
- the units of 'a' are spikes/sec

Because each condition was tested 10 times, we expect you to report the mean of those 10 repetitions. A helpful Matlab function here is 'mean':

```
>> meanResponse(1) = mean([a1 a2 a3 …  a10])   % mean response to condition a
>> meanResponse(2) = mean([b1 b2 b3 …  b10])   % mean response to condition b
```

Once you have the mean response to each of the nine conditions, you can use this information to make a plot (step 6 below). It is nice to put these nine mean responses in a list (a vector in Matlab). The line above shows how to fill the first two elements of that vector.

You can use your programming skills in Matlab to do this more efficiently than shown here, or you can work through each mean in each condition one at a time and keep careful track of your work.

Step 5. Determine the Confidence in Your Estimate of the Neuronal Response in Each Condition

In step 4 you determined the average (mean) spike rate that occurred during each of the nine movie conditions. That is, the average across the 10 runs. How confident are you in each of these average values? That is, if you could repeat the entire experiment again, how similar would the results be? It is questions like these that force us to think about statistics. Unfortunately, we cannot teach an entire lesson in statistics here. For now, it is enough to know that the mostly widely used measure of confidence in these kind of plots is called the 'standard error of the mean' or 'SEM' for short. This is a measure of how confident one is about the mean that we estimated. That is, the SEM is a measure of how far the true mean could possibly be from the mean that you computed on your plot. As you might imagine, your confidence in your mean response to each motion condition depends on the number of observations you have to create the mean (in this case, we have 10 observations – one from each of the 10 runs of the movie).

The formula for the SEM is:

SEM = standardDeviation / sqrt (n)

Where n is the number of observations.

You can determine the SEM for each of your mean values using this function that we made for you if you like:

```
>> SEM(1) = computeSEM([a1 a2 a3 …  a10]);
```

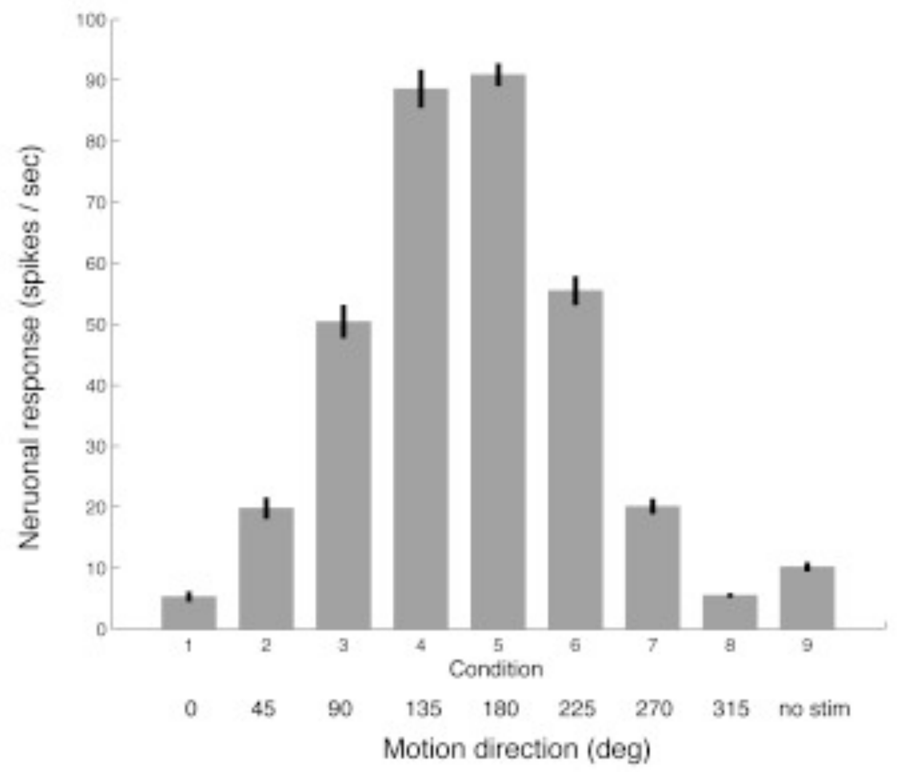
Step 6. Make a Plot of the Mean Response (and Confidence) to All Tested Conditions

**In this case, we call this a tuning curve for motion direction.

We have given you a basic function to help you make this plot:

```
>> plotTuningCurveBar(meanResponse, SEM);
```

However, you are welcome and encouraged to make your own, especially when you work with your lab data because the axis and order of the plots depends on the particulars of your experiment (movie). Note that your raster and histogram plots are just a more detailed way of looking at the same data that goes into your tuning curve. Thus, you should check that your two sets of plots make sense together. For example, if your raster/histogram plots show that the neuron responded best to motion at 180 deg (downward motion), then your tuning curve should show the same motion direction preference. An example of a tuning curve is depicted below:



Assignment Submission Details

You should be making the following figures:

1. Spike raster and histogram for the entire movie.
2. Spike rasters and histograms for each of the nine tested conditions.
3. A tuning curve for motion direction with error bars (SEM).

You should only upload one file to the class website! This file (a .doc or .pdf only!) should contain all of your figures (to accomplish this you can save your figures as images (e.g. .jpg or .bmp) and then import them into a text document.

Note, while we are not requiring you to submit the code you used to generate your figures, it is in your best interest to be organized about your coding since you will most likely be doing similar analyses on your own fly data. Thus, I would suggest writing a script or function to accomplish the steps in this project.