CHAPTER

3

Wrangling Spike Trains

Neurons are peculiarly social. Some neurons are veritable chatterboxes, some are quite quiet, and some are even trying to silence others. Some neurons prefer to talk only to close neighbors, whereas others send messages to comrades in far distant regions. In this sense, all neuroscience is necessarily social neuroscience. Of course neurons don't communicate with each other via spoken words. Rather, they use action potentials (also known as voltage *spikes*) as their universal means of long-distance communication. Every neuron sends spikes in its own idiosyncratic fashion and every neuron uses its dendritic arbor to receive signals from other neurons in turn. The interface points between neurons are known as *synapses*. A neuron may have many points (anywhere from one for some neurons in the retina to more than 100,000 for neurons in the cerebellum) of communication (synapses) with other neurons. While spikes are not exchanged directly—the signal crossing the synapse is chemical in nature in almost all synapses—it is the voltage spikes that drive the neural communication machinery. Specifically, spikes traveling down the axon of the *presynaptic* neuron trigger the chemical action in the synapse that enact further voltage changes—and perhaps more spikes in the *postsynaptic* neuron. We will consider details of this signal exchange in Chapter 6, Biophysical Modeling and Chapter 7, Regression.

For now, we will take for granted that neurons use spikes as their preferred medium of communication. It is fair to say that a major challenge faced by contemporary neuroscientists is to elucidate the meaning of these spikes. Put differently, we (the neuroscience community) are trying to crack the neural code. Our starting point in this pursuit is the signal itself—the spikes. Due to their nature as all-or-none events, we will represent the occurrence of spikes over time as numbers, specifically zeroes for "no spike" or ones for "spike." So consider the following list of numbers:

[0,0,0,0,0,0,0,0,0,1,0,1,0,1,0,0,0,1,0,0,0]

Neural Data Science.
DOI: http://dx.doi.org/10.1016/B978-0-12-804043-0.00003-9

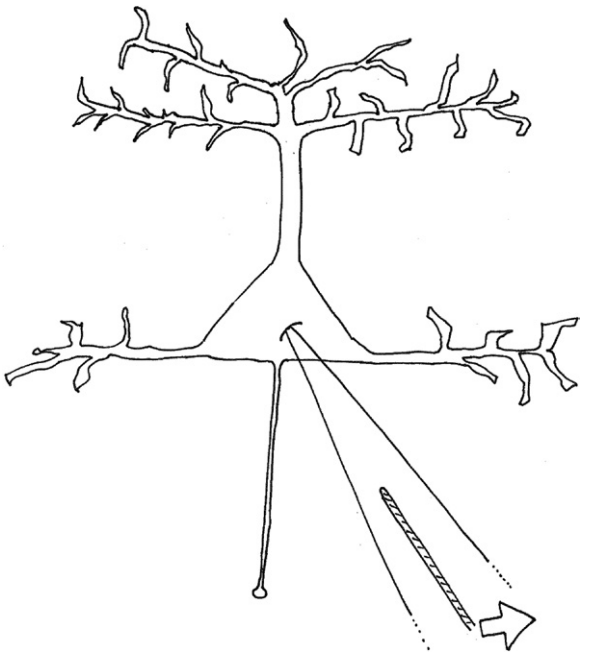

FIGURE 3.1 Extracellular electrode recording voltage spikes from an individual neuron in a dish.

By itself, this list of numbers is meaningless. However, let's assume that we have a neuron in a dish, that the neuron is alive, that it is capable of sending spikes, that this neuron will generally not send any spikes in the dark, and that this neuron will send some amount of spikes if you shine a green light (at a wavelength of 550 nm) on it. Let's also assume that we have a recording electrode near the point where this neuron sends spikes, and that our electrode has the fancy ability of telling our computer whether or not the neuron is spiking over time, as captured in the vector with 0's and 1's above. This scenario is schematized in Fig. 3.1.

Without knowing yet why the spikes occurred, we can make a couple of remarks about this list of numbers representing spikes. First, we know how many spikes are in the list:

Pseudocode	Sum up the numbers in the vector
Python	`>>> sum([0,0,0,0,0,0,0,0,0,1,0,1,0,1,0,0,0,1,0,0,0])` `4`
MATLAB	`>> sum([0,0,0,0,0,0,0,0,0,1,0,1,0,1,0,0,0,1,0,0,0])` `ans = 4`

But you didn't need POM to tell you that there are 4 ones in this list, as you can see that immediately. It is, however, convenient for us to measure how long the list is using the function *len* in Python and *length* in MATLAB:

Pseudocode	Count the number of elements in the longest dimension of the vector
Python	`>>> len([0,0,0,0,0,0,0,0,0,1,0,1,0,1,0,0,0,1,0,0,0])` `21`
MATLAB	`>> length([0,0,0,0,0,0,0,0,0,1,0,1,0,1,0,0,0,1,0,0,0])` `ans = 21`

Which means the list of numbers has 21 entries or elements. Recall that each number in the list represents whether or not the neuron sent a spike at that time, that the 0 on the far left of the list represents *time* = 0, and that each successive number on the list represents whether or not the neuron sends a spike at that time. We could say:

Pseudocode	Create a list of 21 successive integers representing time and align it with the 21 neuron states

Python	`range(21), [0,0,0,0,0,0,0,0,0,1,0,1,0,1,0,0,0,1,0,0,0]`
	`([0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20],`
	`[0,0,0,0,0,0,0,0,0,1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0])`

MATLAB	`>> [linspace(0,20,21); [0,0,0,0,0,0,0,0,0,1,0,1,0,1,0,0,0,1,0,0,0]]`
	`ans =`
	`0 1 2 3 4 5 6 7 8 9 10`
	`0 0 0 0 0 0 0 0 0 1 0`
	`11 12 13 14 15 16 17 18 19 20`
	`1 0 1 0 0 0 1 0 0 0`

We interpret this to mean that at times 0, 1, 2, 3, 4, 5, 6, 7, and 8, the neuron is not spiking. At time 9, it spikes, at time 10 it is quiet, at time 11 it spikes, at time 12 it is quiet, at time 13 it spikes, at times 14 through 16 is it quiet, and then at time 17 it spikes one last time before being quiet again.

We said earlier that this neuron tends to spike if it is illuminated with green light, but not in darkness. What we are simulating here is a tool known as *optogenetics*, where neurons will actually increase their activity in response to light (Boyden et al., 2005; Tye & Deisseroth, 2012).

So let's indicate the time points during which such a green light was on in ***italic bold***, leaving the rest of the time points unbold (representing times during which the light was off):

[0,1,2,3,4,***5,6,7,8,9,10,11,12,13***,14,15,16,17,18,19,20]

Let us assume now that each time point above is in units of milliseconds. What this means is that we started recording from the neuron at time 0 when the light was off. After 4 ms of recording, on the fifth millisecond, the green light was turned on. The light then stays on for 9 ms (through the 13th millisecond) before shutting off.

With this knowledge of the stimulus conditions, we can determine a characteristic feature of this neuron: "*first spike latency to stimulus*." This parameter is generally used by neuroscientists to establish how "fast" or "ready to spike" any given neuron is.

We now know just enough about the stimulus and the neuron to be good neuroscientists and form a hypothesis. Let's hypothesize that the neuron always fires a spike 4 ms after a light is turned on.

Let's put the string of 0's and 1's into a *variable* now:

Pseudocode	Assign data to variable spikeTrain
Python	`>>> spikeTrain = [0,0,0,0,0,0,0,0,0,1,0,1,0,1,0,0,0,1,0,0,0]`
MATLAB	`>> spikeTrain = [0,0,0,0,0,0,0,0,0,1,0,1,0,1,0,0,0,1,0,0,0];`

With the spikes now stored in spikeTrain, we can pull out spikes at different time points. That is, we can ask, say, at time $t = 5$, when the light is first turned on, is the neuron spiking?

Pseudocode	Output the contents of variable spikeTrain at the position corresponding to $t = 5$
Python	`>>> spikeTrain[5]` `0`
MATLAB	`>> spikeTrain(6)` `ans = 0`

A resounding no. But what about if we want to know if the cell spikes at any time *after* $t = 5$?

Pseudocode	Output all elements of spikeTrain after the position corresponding to $t = 5$
Python	`>>> spikeTrain[5:]` `[0,0,0,0,1,0,1,0,1,0,0,0,1,0,0,0]`
MATLAB	`>> spikeTrain(6:end)` `ans = 0 0 0 0 1 0 1 0 1 0 0 0 1 0 0 0`

Note: Here we see some subtle but critical differences in Python versus MATLAB. First, in MATLAB, the first element of a vector is element "1," whereas the corresponding first element in Python is "0." So to access the same element (here the one at time 5), we have to add 1 to the MATLAB index. Second, the colon operator: returns all elements from a starting point until an endpoint. Python assumes you want all elements until the end of the vector if no endpoint is specified, the corresponding MATLAB command to specify "all elements until the end of the vector" is "end." Finally, note that the MATLAB command uses parentheses whereas the Python command uses square brackets.

This output (in POM) represents the neuron's spiking activity after the green light turned on. If the first output value in the list were a 1, then the neuron's *latency to first spike* would be 0 ms, i.e., the neuron's response would be coincident with the light turning on. But things rarely happen instantly in biology, let alone neuroscience. Rather, whatever makes our neuron spike in response to light takes some time to flip some internal switches before the spike occurs (we'll address the internal spiking mechanisms in chapter: Biophysical Modeling). For now, we just acknowledge that the neuron takes some time before spiking. The record of the successive spiking of a neuron over time is called the "*spike train*" (see Fig. 3.2) and various measures to characterize the spike train have been proposed.

The time it takes for a neuron to be responsive, i.e., for a neuron to spike in response to a stimulus (in our case, to the light) is known as the *response latency*. Different ways exist to measure response latency, but for a single light pulse and a single recording from a neuron that spikes relatively infrequently (or sparsely), the time it takes for the first spike to occur is a reasonable measure. So let's do that, and calculate the *latency to first spike* by typing:

Pseudocode	Find the first value in the elements of spikeTrain that matches 1
Python	`>>> spikeTrain[5:].index(1)` `4`
MATLAB	`>> find(spikeTrain(6:end)==1); ans(1)-1` `ans = 4`

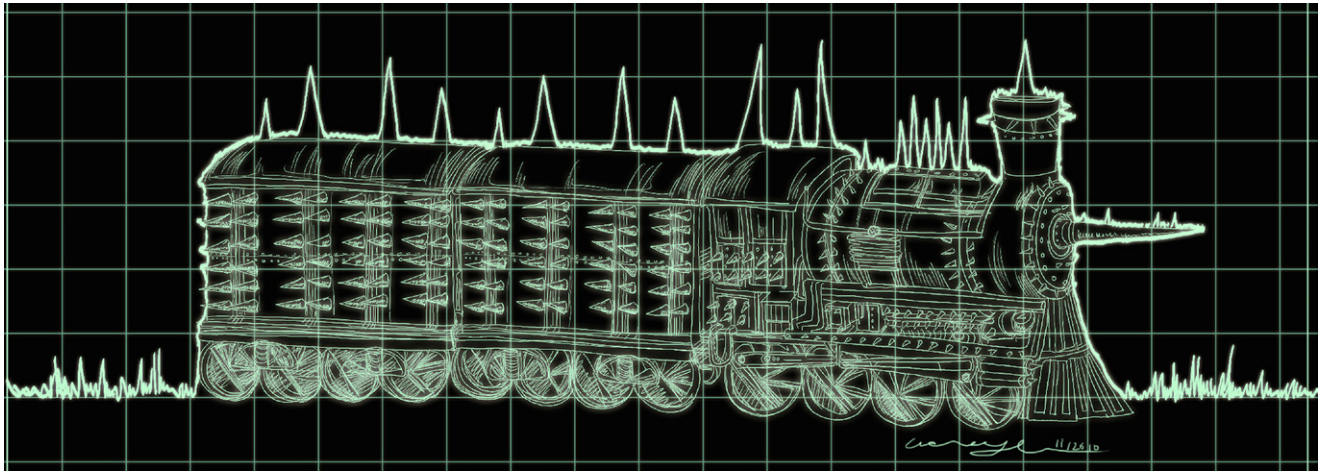

FIGURE 3.2 A *spike train*, as seen on an oscilloscope. Voltage spikes recorded from an individual neuron are plotted over time.

Note that in the MATLAB version of this code, we concatenate two commands in the same line by the use of the semicolon operator. We have to bring the result into the same timebase as the Python code by adding 1 to the start point, then subtracting it again from the result. If we had started with calling the first element time "1," we would have had to subtract one from Python to get the right index, then add 1 to the answer. Whether the first element of a vector is element zero or element one is a philosophical consideration that we will further explore in Appendix A.

Here we took our list of values in the variable spikeTrain that occurred after a time of 5 ms and used the Python function *index* (*find* in MATLAB) to find the first value in this variable that represents the spike train that matches *1* (representing a spike). We can make this more flexible by giving the light onset time *5* a variable name, and also the spike indicator value *1* a variable name. Generally speaking, it is a bad idea to hard-code values that are given to functions explicitly. It is almost always better to use variables—code that uses variables is much easier to maintain.

`Pseudocode`	Assign values to variables, then pass those to the function to avoid hard-coding

```
Python      >>> lightOnsetTime = 5
            >>> spikeValue = 1
            >>> spikeTrain[lightOnsetTime:].index(spikeValue)
            4

MATLAB      >> lightOnsetTime = 5;
            >> spikeValue = 1;
            >> mShift = 1;
            >> find(spikeTrain(lightOnsetTime+mShift:end)==spikeValue);ans(1)-mShift
            ans = 4
```

This version (without the hardcoding) also makes it more clear what is going on in the MATLAB case. Because of MATLAB's indexing conventions, we know that MATLAB indices are always shifted by 1 relative to Python. So we allocate this to a constant ("mShift") and add it to the lightOnsetTime, then subtract it again from the result. Note that the output is the same, which is reassuring, as our results should depend on the properties of the neuron we study, not the software we use to analyze the data we record from it.

Again, in order to find the latency to first spike, we take the variable *spikeTrain*, which contains all of the 1's and 0's that represent the presence or absence of spiking at a given time bin and look only at the times after the light onset, then look for the first time that the vector after light onset contains a 1. Technically speaking, this command returns the number of bins between light onset and the first spike, but as we know that the bin width is 1 ms, we can interpret the result as a time—the latency is 4 ms. In order to be able to reuse a result that we just calculated in this fashion, we should assign the results of a command to a variable. In this case, we will call it "latencyToFirstSpike." Generally speaking, it is advisable to use variable names that are meaningful as that makes code much more readable.

Pseudocode	Calculate latency to first spike and assign it to a variable with meaningful name

Python	`>>> latencyToFirstSpike = spikeTrain[lightOnsetTime:].index(spikeValue)`
	`>>> print latencyToFirstSpike`
	`4`

MATLAB	`>> temp = find(spikeTrain(lightOnsetTime+mShift:end)==spikeValue);`
	`>> latencyToFirstSpike = temp(1)-mShift`
	`latencyToFirstSpike = 4`

Note that in the example above, the MATLAB computations are done in two steps. We first declare a temporary variable temp that finds *all* instances in which spikes occur after light onset. We then, in a second step, find the index of the first element and correct for the MATLAB indexing shift by subtracting one, and assigning that to the variable latencyToFirstSpike.

Python has its own set of idiosyncrasies—these brief calculations in MATLAB involve only regular parentheses. In Python, *square brackets*, [ ], are used when specifying the range of values within a list of numbers. In contrast, regular parentheses, ( ), are used to invoke functions with the particular parameters within the parentheses as inputs. In this case, we pass the variable spikeValue to the function *index,* which is a built-in function—*index* is not the only such function. Python has many functions that we'll be using, and when we do, we'll use parentheses to give them values to operate on.

FIGURE 3.3 Cartoons can efficiently illustrate important principles.

Now we have our estimate of the neuron's latency (4 ms)! As the readers of your research papers are likely to be primates and primates are predominantly visually guided animals, we should make a plot to illustrate the spiking activity of the neuron (Fig. 3.3).

To plot the spiking activity, we need to know the time of every spike in the list. People sometimes called these the *spike timestamps* but we'll just call them *spikeTimes*:

`Pseudocode`	Find and then output the times at which the neuron spikes

`Python`	`>>> spikeTimes = [i for i,x in enumerate(spikeTrain) if x==1]`
	`>>> print spikeTimes`
	`[9, 11, 13, 17]`

`MATLAB`	`>> spikeTimes = find(spikeTrain==1)-mShift` `spikeTimes = 9 11 13 17`

The Python part of this is a whopping nest of code! Let's untangle it a bit. First, see how we put the list of numbers `spikeTrain` (a bunch of 1's and 0's) into the function `enumerate`. Don't bother typing `enumerate(spikeTrain)` into your command line or trying to print it yet. The function `enumerate(spikeTrain)` cycles through the list `spikeTrain` and keeps track of the index of each element in `spikeTrain`.

The middle part of the code `i,x in enumerate(spikeTrain)` means that we will be going through each element in `spikeTrain` and naming each element along the way "`x`," and wherever "`x`" is in the list `spikeTrain`, we'll call that location "`i`."

A diagram might help:

i will successively be each element in: [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
x will successively be each element in: [0,0,0,0,0,0,0,0,0,1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0]

Let's look at that line of code again:

```
>>> spikeTimes = [i for i,x in enumerate(spikeTrain) if x==1]
```

Note that POM make use of double equals signs to test for the equality of two values. We have already encountered this before, in Chapter 2, From 0 to 0.01, but it is worth repeating (in our experience, this is confused a lot, particularly in the beginning) that a single equal sign is an assignment operator, assigning whatever is on the right (usually the result of a computation) to the left (usually a variable).

We now understand that the line means to give us the indices of *i* where *x* is equal to 1.

i is [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
and
x is [0,0,0,0,0,0,0,0,0,1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0]

As you can see, this is the case for the indices *i* of 9, 11, 13, and 17. And now, given this line, Python can see it too and return it to you (and put it into the variable "spikeTimes" as well).

We can conceive of this problem in reverse, too.
If *x* is equal to 1, *where x is each element in* [0,0,0,0,0,0,0,0,0,1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0]
give us the corresponding value in *i:*

[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]

We underlined the *i*'s and *x*'s where *x* is equal to 1.

That's a lot of work (the equivalent MATLAB code seems much simpler) and would be simpler yet if we didn't have to bring the result into a format that conforms to Python's zero-indexing conventions—but hey, that's a small price to pay to use a real programming language, like the real hackers do. After all, social life is all about tribal signaling.

But let's not lose sight of the fact that the point of doing this was so that we can graph it. To graph (or "plot," as we will call it from now on) something in Python, we need to import a library of functions designed for this purpose. This library is called "*matplotlib*," a library that gives us all the plotting tools we need at this point.

Pseudocode	Import library of plotting functions
Python	`>>> import matplotlib.pyplot as plt`
MATLAB	No equivalent. These plotting functions already come with MATLAB

This means we've loaded the function *pyplot*, a function of the library *matplotlib*, but when we loaded it we gave it a shorthand (*plt*) so we can refer to it more easily—and with a nickname of your choice! You'll find that the strategy of importing functions as two or three letter shorthands will save you a lot of typing. We recommend it.

We will now create a figure to work on:

Pseudocode	Open figure
Python	`>>> fig = plt.figure()`
MATLAB	`>> figure`

This establishes a frame to work within. Within this frame, we can have one to many smaller windows, or *subplots*. For now, we're plotting a small set of spikes, so we only need one *subplot*.

Pseudocode	Place a subplot (or axes) into the figure
Python	`>>> ax=plt.subplot(111)`
MATLAB	No equivalent. If there is only one pair of axes, it will take up the entire figure by default. You could still type subplot(1,1,1) and place axes, but this is not necessary.

What the *111* means will be a little more clear once we have multiple subplots in a figure. For now, just know that if you want only one plot in your figure, you are saying that you want one large subplot, and therefore use *subplot(111)*. We call this subplot "ax" in honor of the axes of a Cartesian plot, which is the kind we want here.

To plot the spike times, we'll use the common visualization method of a *spike raster plot*. In such a plot, each spike is represented as a vertical line (at the time when it occurred, with time on the x-axis) (Fig. 3.4).

Pseudocode	Plot vertical lines at the times when a spike occurred, then show the figure
Python	`>>> plt.vlines(spikeTimes, 0, 1)` `>>> plt.show(block=False)`
MATLAB	`>> line(repmat(spikeTimes,2,1),repmat([0; 1],1,4),'color','k')` `shg`

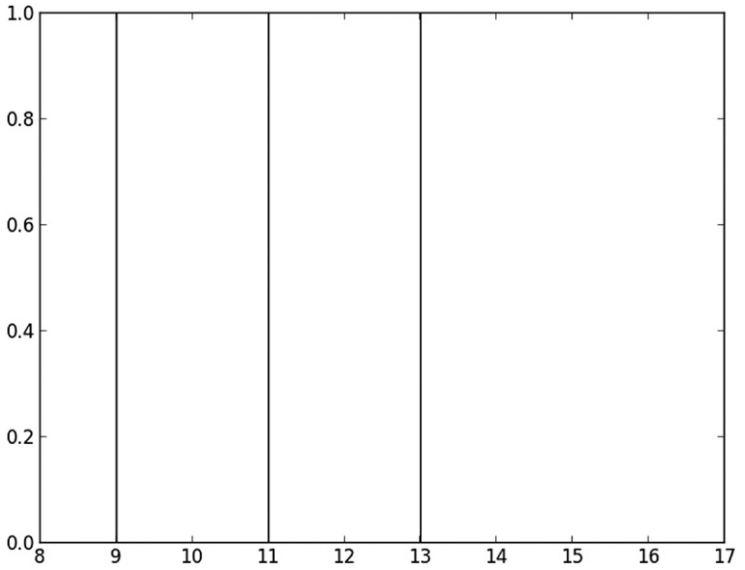

FIGURE 3.4 A bare bones raster plot of a single trial.

Voila, our first plot of neuroscience data. Obscenely, we have neither axis labels nor units … yet. Before we rectify this problem, let's discuss how the figure was brought about by the commands directly above.

In Python, we invoke the function *vlines* (a subfunction of the plotting package we called *plt*) by putting a period between them. This is generally the way to invoke subfunctions of imported packages. We then use parentheses to give values to this function. The three arguments we pass to the function *vlines* are*:* First a list of spike times, then the minimum value of the vertical lines—0, then the maximum value of the vertical lines—1.

In MATLAB, there is no function for vertical lines specifically, so we use the general purpose function *line*. It takes matrices to specify x and y coordinates and plots one line for each column whereas row-values indicate start and end positions of the lines. We have to create these matrices first, so we create two 2×4 matrices with the *repmat* function.

We also have to set the color of the lines to black as the default MATLAB color is blue in order to match the plot produced by Python. The function *shg* shows the current graph.

This plot illustrates when the neuron spikes, but doesn't contain any information about the light stimulus yet. Let's make the time during which the light is on a shaded green:

Pseudocode	Add a shaded green rectangle from times 5 to 14
Python	In :`plt.axvspan(5,14,alpha=0.1,color='g')`
MATLAB	`rectangle('Position',[5,0,9,1],'FaceColor',[0.7 1 0.7],'linestyle','none')`

This creates a shaded green box that spans the figure vertically and is bounded horizontally at 5 and 14.
Python: The *alpha* value makes the box transparent (smaller *alpha* values make the color appear lighter).
MATLAB: Specifies the color of the box by giving it an RGB triplet. A light green in this case.
Let's now specify the range of times our *x*-axis should span by invoking the *xlim* function so that we can see times during which the neuron did not fire, i.e., before visual stimulation:

Pseudocode	Setting the range of the x-axis to include the entire time interval of interest
Python	In :`plt.xlim([0,20])`
MATLAB	`xlim([0 20])`

Before showing this figure to anyone, we strongly recommend to add a label to the *x*-axis and a title to the figure.

Pseudocode	Add meaningful axis and figure labels, which is critical in science
Python	`>>> plt.title('this neuron spikes in response to a single light stimulus')` `>>> plt.xlabel('Time (in milliseconds)')`
MATLAB	`>> title('this neuron spikes in response to a single light stimulus')` `>> xlabel('Time (in milliseconds)')`

Due to the noise inherent in the system as well as the measurements, data from a single trial are rarely sufficient to reach reliable conclusions about the connection between visual stimulation and the neural response (Fig. 3.5).

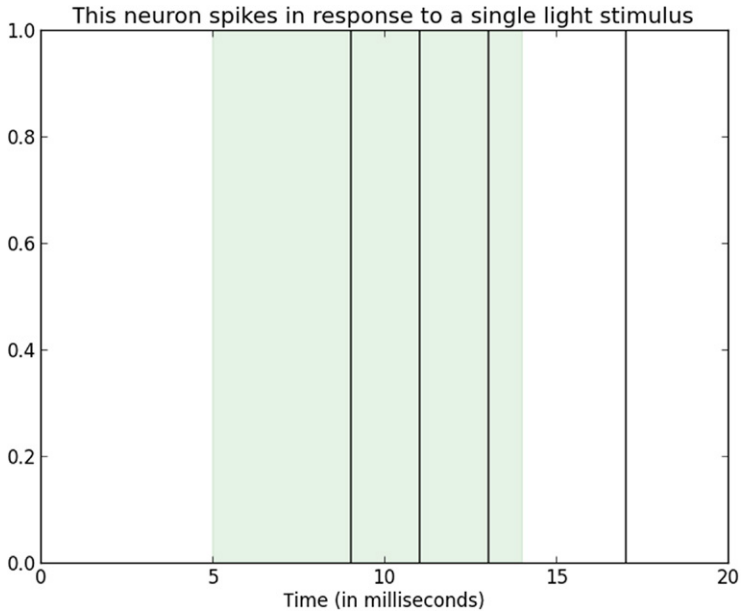

FIGURE 3.5 A raster plot of a single trial with axis labels, title, and stimulation condition.

What if there is some mechanism inside the neuron that causes it to spike highly unreliably? What if we are recording the signals of this neuron in a noisy environment? In theory, we would need to stimulate the neuron an infinite number of times and record an infinite number of responses in order to be really sure. But most people are not theorists, living in a nonplatonic world. As such (and neuroscientists at that) we have to make do with less than infinite amounts of data. Say we had just enough funding to allow us to record data from 10 spike trains and 10 identical (green) light stimuli. These data are contained in the "tenSpikeTrains" variable. It can be downloaded from the companion website, but we also print it on the next page.

Pseudocode	Representing the ten spiketrains in the respective formats
Python	`>>> tenSpikeTrains = [[0,0,0,0,0,0,0,0,0,1,0,1,0,1,0,0,0,1,0,0,0],[0,0,0,0,0,0,0,0,0,1,1,0,0,0,1,0,0,0,1,0,0,0],[0,1,0,0,0,0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,1,0],[0,0,0,0,0,0,0,0,0,1,1,0,1,0,0,0,0,0,0,1,0,0],[0,0,0,0,0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,1,1,0,0,0,1,0,0,0,1,0],[0,0,0,0,0,0,0,0,1,1,1,0,0,1,1,0,0,1,1,0,0],[1,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,1,0,0,0,0],[0,0,0,0,0,0,0,0,0,1,1,0,0,1,1,0,0,1,1,0,0,0]]`
MATLAB analogous	`>> tenSpikeTrains = {[0,0,0,0,0,0,0,0,0,1,0,1,0,1,0,0,0,1,0,0,0],[0,0,0,0,0,0,0,0,0,1,1,0,0,0,1,0,0,0,1,0,0,0],[0,1,0,0,0,0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,1,0],[0,0,0,0,0,0,0,0,0,1,1,0,1,0,0,0,0,0,0,1,0,0],[0,0,0,0,0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,1,1,0,0,0,1,0,0,0,1,0],[0,0,0,0,0,0,0,0,1,1,1,0,0,1,1,0,0,1,1,0,0],[1,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,1,0,0,0,0],[0,0,0,0,0,0,0,0,0,1,1,0,0,1,1,0,0,1,1,0,0,0]]}`
MATLAB suitable	`>> tenSpikeTrains = [0,0,0,0,0,0,0,0,0,1,0,1,0,1,0,0,0,1,0,0,0; 0,0,0,0,0,0,0,0,1,1,0,0,0,1,0,0,0,1,0,0,0;0,1,0,0,0,0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,1,0;0,0,0,0,0,0,0,0,1,1,0,1,0,0,0,0,0,0,1,0,0;0,0,0,0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0;0,0,0,0,0,0,0,0,1,1,0,0,0,1,0,0,0,1,0;0,0,0,0,0,0,0,0,1,1,1,0,0,1,1,0,0,1,1,0,0;1,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,1,0,0,0,0;0,0,0,0,0,0,0,0,1,1,0,0,1,1,0,0,1,1,0,0;0,0,0,0,0,0,0,1,0,0,1,0,1,0,0,0,0,0,1,1,0,0]`

Note that Python uses nested square brackets here. It appears, from the placement of the *square brackets*, that there are 10 lists (each inside *square brackets*) nested inside of one big, all-encompassing list (note the double *square brackets* at the very beginning and end of *tenSpikeTrains*). In fact, this is the case and *tenSpikeTrains* is technically a list of lists. We can represent these spiketrains like that in MATLAB too, by using cells (the "MATLAB analogous" code), but the most suitable way to represent these data in MATLAB is as a 10×21 matrix ("MATLAB suitable"). In other words, we represent each spiketrain as entries in 21 consecutive columns and each individual spiketrain as a separate row. So at every point in the MATLAB suitable code above where there is a],[in Python, there is a semicolon in MATLAB. This works because the time base (21 bins with a width of 1 ms) is the same for each spiketrain. If this was not case, e.g., if there were missing data, using cells would be more apt. Matrices are ordered arrangements of numbers and cells are ordered arrangements of matrices. So Cell:Matrix as Matrix:Number, in MATLAB. Cells can accommodate matrices with different dimensionalities in each entry so they are very helpful, but make it harder to do some computations on them. In addition, they complicate the notation—note the curly braces { } that indicate we are dealing with cells. We will not say more about them at this point, but will when we have to, later in this book.

Back to the data, if we look at the first entry of *tenSpikeTrains,* we find:

Pseudocode	Return the contents of the first spike train
Python	`>>> tenSpikeTrains[0]` `[0,0,0,0,0,0,0,0,0,1,0,1,0,1,0,0,0,1,0,0,0]`
MATLAB analogous	`>> tenSpikeTrains{1}` `ans = 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 0 1 0 0 0`
MATLAB suitable	`>> tenSpikeTrains(1,:)` `ans = 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 0 1 0 0 0`

The first spiketrain is contained in the first row (MATLAB suitable) or first cell (MATLAB analogous) or the 0th list (Python). This is the flipside of Python zero indexing. It made sense that the first time of a time series would be time zero. But the first element of a list is still the first element, not the zeroth element. So each indexing convention (0 for Python, 1 for MATLAB) has use cases where it is more "natural," which is why both of them are still around.

Regardless of implementation, the command doesn't return a single value, but a list. How many elements are in this master list?

Pseudocode	How many elements does the variable tenSpikeTrains have?
Python	`>>> len(tenSpikeTrains)` `10`
MATLAB analogous	`>> length(tenSpikeTrains)` `ans = 10`
MATLAB suitable	`>> size(tenSpikeTrains,1)` `ans = 10`

In the "MATLAB suitable" case, we have to count the number of rows, which is achieved by telling the *size* function to evaluate the first dimension (rows).

You might have expected way more than that (like around 200), but the *len* function in Python looks at how many things are inside the list much in the same way that a bundle of bananas only counts as one item at the grocery store when you are in the express lane. It simply returns the number of elements (in this case lists) it contains. The same is true for the MATLAB analogous case—the cell has 10 entries, each of which is a matrix (and each of which represents an individual

spike train). Accessing cell contents and doing computations on them is more involved than is worth getting into at this point, so we'll punt that to Chapter 4, Correlating Spike Trains, and continue solely with the "suitable" case below.

In order to make a raster plot for the data from *all* the trials, we use a similar approach we used for plotting the data from a single trial, except we cycle through each list in the list. To do this, we will use a *for loop*.[a]

Python	MATLAB (suitable)

`>>> fig = plt.figure()`
`>>> ax = plt.subplot(1,1,1)`
`>>> for trial in range(len(tenSpikeTrains)):`
`  ... spikeTimes = [i for i,x in enumerate (tenSpikeTrains[trial]) if x==1]`
`  ... plt.vlines(spikeTimes,trial,trial+1)`
`>>> plt.axvspan(5,14,alpha=0.1,color='g')`
`>>> plt.xlim([0,20])`
`>>> plt.show()`[b]
	`>> ax=figure`
	`>> rectangle('Position',[5,0,9,11],'FaceColor', ... [0.7 1 0.7],'linestyle','none')`
	`>> for ii = 1:size(tenSpikeTrains,1)`
	`  >> spikeTimes = find(tenSpikeTrains(ii,:)==1)-mShift;`
	`  >> line(repmat(spikeTimes,2,1),repmat([ii-0.5;ii+0.5] ,1, ... length(spikeTimes)),'color','k')`
	`>> end`
	`>> xlim([0 20])`
	`>> shg`

As for Python, there are a few noteworthy things about this code, starting with the *for loop*. We know that the length of *tenSpikeTrains* is 10, so we can think of the *for loop* line of code as:

[*aside: We put this line of code front and center to emphasize it and talk about, not for you to retype it into the command line, or think that code usually sits centered in the page*]

```
for trial in range(10):
```

We can also see that range(10) is:

```
>>> range(10)
[0,1,2,3,4,5,6,7,8,9]
```

[a] *For loops* are your friend for easy plotting, and your enemy in heavy computation. If you are cycling through a network simulation or calculation, too many nested *for loops* will bring even the most powerful computer to its knees. As you begin to learn more programming tools, always be asking yourself if the code can be written without use of a *for loop*. But we are introducing the concept just now—we'll cover later how to write code that runs faster. Right now, using a for loop is fine, particularly as it is usually much clearer to understand what is going on when using loops.

[b] You may notice that if you use the command plt.show() that you will be unable to continue coding. To be able to continue coding, use the command: plt.show(block=False). To update the figure with any additional commands (for example, if after you show the figure you want to add an axis label) use plt.draw() after your additional command to update the figure.

That's right, range(10) is equal to the list [0,1,2,3,4,5,6,7,8,9]. In MATLAB, the command 1:size(tenSpikeTrains,1) achieves exactly the same.

We can thus think of the *for loop* line of code in Python:

```
for trial in range(len(tenSpikeTrains)):
```

as equivalent to

```
for trial in [0,1,2,3,4,5,6,7,8,9]:
```

Which means we are going to go to the next line of code after the *for loop* 10 times, and each time we go to the next line the variable *trial* will increase to the next value in the list (starting with 0). It is called a *for loop* because it loops through a list of values. The principles of the for loop in Python are reflected in the for loop used in MATLAB:

```
for ii = 1:size(tenSpikeTrains,1)
```

So whats up the line after the *for loop*? There's the ellipses and some spaces before we get to the *spikeTimes* line, which is very similar to how we got *spikeTimes* before. Note in Chapter 2, From 0 to 0.01, we discussed the importance of *white space* in Python. If you are running iPython from a command line or terminal, it will automatically create an indented next line if the preceding line ends with a colon (see examples of frequently made mistakes when using loops in the terminal in Appendix B). Ellipses are a placeholder for whitespace. In effect they represent white space explicitly.

```
>>> for trial in range(len(tenSpikeTrains)):
       ... spikeTimes = [i for i,x in enumerate(tenSpikeTrains[trial]) if x==1]
       ... plt.vlines(spikeTimes,trial,trial+1)
```

It will keep creating indented lines and staying inside the grasp of the *for loop* until you hit *return* on a blank line. This indicates to Python that the for loop has ended. Notice how the line of code

```
>>> spikeTimes = [i for i,x in enumerate(tenSpikeTrains[trial]) if x==1]
```

is similar to how we got `spikeTimes` for a single trial. This time though, we have our 1's and 0's within lists inside of the list `tenSpikeTrains`, so the variable `trial`, which iterates through the values *0–9*, will call each of the *10* lists inside the list `tenSpikeTrains`. `spikeTimes` is now a temporary variable, and each time `trial` is updated in the for loop, `spikeTimes` is reassigned to a new list of spike times.

Let's reiterate what `enumerate` does. This uniquely Python function allows us to build a for loop within a single line of code, as `enumerate` returns both each value in the list (like the `for` loop above) as well as the index of the value, where `i` is the index, and `x` is the value. Additionally, within this single line we condition returning the index `i` on the value `x` equaling 1. Thus, in this single line of code, we return the indices of the values equal to one, in a Pythonic fashion known as a *list comprehension*.

The MATLAB code tries to achieve the same, but note that the ellipses (…) are at the end of the line. They indicate that the command continues in the next line. So the use of ellipses is very different in Python versus MATLAB. In Python, indicating that the command continues in the next line is done via a backslash, "\."

For each trial in `tenSpikeTrains` we plot vertical lines for the spikes, where the values *trial, trial+ 1,…,trial+ 9* are stacked sequentially and vertically, where every "row" of the plot represents a trial.

Both POM have commenting conventions, which we are starting to use here—and from now on. Anything after the hashtag (#) in Python isn't read as programming code and is ignored by the computer, meaning it is not interpreted as an instruction and thus not attempting to execute it. In general, we recommend to write a sentence at the beginning of a paragraph of code to explain what that paragraph is supposed to do and (in broad terms) how, then comment on critical pieces of code, e.g., note what a variable is supposed to contain. The analogous symbol in MATLAB is the percentage sign (%). Anything written after it is understood to be a comment.

Let us now add figure labels and formatting to the below code Fig. 3.6:

Python	MATLAB (suitable)
`plt.ylim([0,10])`	`ylim([0.5 10.5])`
`plt.title('this neuron spikes to repeated trials of the same stimulus')`	`title('this neuron spikes to repeated trials of the same stimulus')`
`plt.xlabel('time (in milliseconds)')`	`xlabel('time (in milliseconds)')`
`plt.ylabel('trial number')`	`ylabel('Trial number')`
`plt.yticks([x+0.5 for x in range(10)],[str(x+1) for x in range(10)]) #1`[c]	`set(gca,'Layer','top') %2`

Align the yticks to be in the middle of each row, (x+1) sets first trial to 1. Set the label axis to the top so that green rectangle doesn't cover the axis.

Pseudocode

```
Create figure and specify subplot
Draw the stimulus presentation area green
For each of the ten trials
Extract the spike times from the spike train variables
Plot each row in the raster plot as vertical lines
Set the x-axis limits
Set the y-axis limits
Set the title
Set the x-axis label
Set the y-axis label
```

[c]You may encounter errors when typing between lines. The line of code: plt.yticks([x+0.5 for x in range(10)],[str(x+1) for x in range(10)]) can be typed across multiple lines in a text editor as:

plt.yticks([x+0.5 for x in \ range(10)],[str(x+1) for x in \ range(10)]).

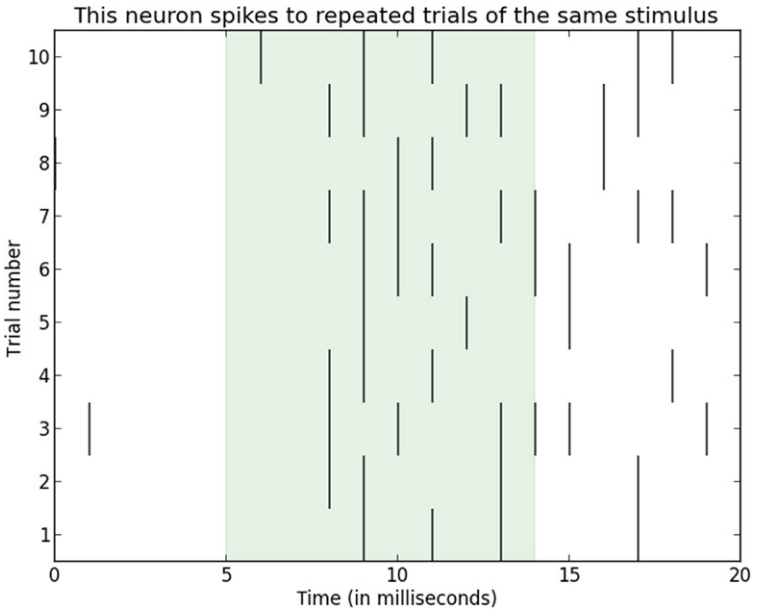

FIGURE 3.6 Raster plot of the neural response to 10 trials of a green light stimulus.

A raster plot yields a snapshot of *raw spike times* elicited by the stimulus across trials. We can see that the first value we calculated, the *first spike latency*, seems to vary considerably between trials: on some trials the first spike occurs at 8 ms, on some trials the first spike happens at 9 ms, on one trial it comes at 6, on a handful of trials it is 10, and on one trial there is a spike 4 ms before the light even comes on. We can also see that the neuron seems to discharge a spike a few times with each light stimulus, and it often also "fires" after the stimulus has turned off. There are many ways to characterize the spiking activity of neurons, some qualitative, some quantitative (like our example, *first spike latency*). Qualitatively, we ask ourselves if, once this neuron fires, does it keep firing? That is, is its ongoing activity tied to the stimulus?

There are many ways to quantify this, but it is a good habit to take a look at the raw data in the form of a raster plot to first get a qualitative sense of the spiking characteristics of the neuron. For instance, the *raster* plot in Fig. 3.6 seems to convey that the neuron responds to this particular light stimulus after 3–5 ms, and that its activity is maintained to some degree after the stimulus has turned off.

Does this falsify our hypothesis that the neuron always spikes 4 ms after the light turns on? It certainly looks like it—the spiking is not nearly as precise as we hoped it to be. But as we formed the hypothesis on a whim (based on the analysis of a single trial), we are free to change it. So let's say we hypothesize that the neuron fires *tonically at a rate of 500 spikes per second* to green light stimuli.

Let's unpack this statement. "Hypothesize that the neuron fires" is simple enough (the neuron discharges spikes) but we go on to make a prediction about the firing, that it fires "tonically." *Tonic firing* is spiking activity that is sustained or ongoing. This contrasts with *phasic* or *transient* firing, which is locked to the timing of the stimulus. We can come up with some metrics for quantifying the tonicity of the firing, but let's posit the qualitative hypothesis that it keeps firing and continue with the quantity "of 500 spikes per second." In our experiment, we didn't stimulate the neuron for an entire, continuous second, so we certainly won't have 500 spikes to count. However, the unit of spike rates is *spikes per second*, even if the instantaneous firing rate is sustained for much less than a second, just like you don't have to drive for an hour in order to go at 100 miles per hour at some point in your ride.

To know how many spikes we ought to expect in our short interval, we simply have to solve for x in the algebraic equation where x is proportional to *500 spikes* as our stimulus length (*9 ms—we really should have planned the experiment better in order to get easier math*) is to 1 s:

$$\frac{x\,spikes}{9\,milliseconds} = \frac{500\,spikes}{1000\,milliseconds} \tag{3.1}$$

which, solving for x, leads us to expect 4.5 spikes for the duration of our 9 ms stimulus. The tail end of the stated hypothesis above was "to green light stimuli," which we sort of covered, and which we'll make more complex just when we start to get a better grasp of our results.

We thus need a way to visualize and condense the many stimulus trials and responses we recorded. We turn to the *peri-stimulus time histogram* (or *PSTH*) to visualize as a bar graph the spiking activity of the neuron over time, just before and after (hence peri) each stimulus. We also make use of multiple subplots within a single figure to compare the *rasters* to the *PSTH*. This is a standard depiction of neural data in early exploratory analysis and we'll revisit it in Chapter 4, Correlating Spike Trains.

An individual subplot is a subdivision of a figure. The code below indicates that the subplots will be arranged so that there are two rows and one column of subplots, and that we'll plot in the first of these. Note that though Python indexes lists by starting at 0, subplot indexing starts at 1 (!). If this seems inconsistent, it is because it is … inconsistent.

Pseudocode	Create the figure window using plt (originally imported via matplotlib) Create the first of 2 "subplots."
Python	`>>> fig=plt.figure()` `>>> ax=plt.subplot(2,1,1)`
MATLAB	`>> fig = figure;` `>> ax = subplot(2,1,1);`

We next create code for plotting (1) a subplot of spike rasters and (2) a PSTH based on that spiking activity.

We highly advise to comment so that (1) your future me can read and remember why you programmed something the way you did and what the variables stand for—which makes the code easier (or even possible) to maintain (2) so that other programmers can look at your code and have any chance of understanding what is going on.

In Fig. 3.6 we created the variable `spikeTimes` for each `trial` and plotted those values right away, overwriting `spikeTimes` with each new `trial`.

`Pseudocode`

Looping through *trial 0,1,...,9*
  Get the index (time) of each spike and append to allSpikeTimes
  Plot vertical lines for each trial
Add the vertically spanning green box
Set the limits of the y-axis to 0 and 10
Add a title, a y-axis label, and an x-axis label to this subplot
#1 Customize the labels of the yticks

`Python`

```
fig=plt.figure()
ax=plt.subplot(2,1,1)
for trial in range(len(tenSpikeTrains)):
  spikeTimes = [i for i,x in enumerate(tenSpikeTrains[trial]) if x==1]
  plt.vlines(spikeTimes,trial,trial+1)
plt.axvspan(5,14,alpha=0.1,color='g')
plt.ylim([0,10])
plt.title('this neuron still spikes to repeated trials of the same stimulus')
plt.xlabel('time (in milliseconds)')
plt.ylabel('trial number')
plt.yticks([x+0.5 for x in range(10)],[str(x) for x in range(10)]) #1
```

`MATLAB`

```
ax = subplot(2,1,1)
rectangle('Position',[5,0,9,11],'FaceColor', [0.7 1 0.7],'linestyle','none')
for ii = 1:size(tenSpikeTrains,1)
  spikeTimes = find(tenSpikeTrains(ii,:)==1)-1
  line(repmat(spikeTimes,2,1),repmat([ii-0.5; ii+0.5],1,length(spikeTimes)),'color','k')
end
xlim([-0.5 20])
ylim([0.5 10.5])
title('this neuron still spikes to repeated trials of the same stimulus')
xlabel('time (in milliseconds)')
ylabel('Trial number')
ax.YTick = [0:1:10]
set(gca,'Layer','top')
```

We next sum across our `tenSpikeTrains` to see the total number of spikes that occur across all trials, using the function `bar` in POM. This function gives us a bar plot of the spiking as a function of time.

We also save the figure. Note the extension `.png` at the end of the string. We could also specify `.pdf` or `.jpg` or a few other image types.

Pseudocode	Now for the PSTH. We create our second subplot. Add the green background during stimulus time. Plot the bar plot #1 format bar (x-values, y-values, bar width) Add labels to the x- and y-axes of this subplot Save the figure Let's take a gander.

Python	`>>> ax=plt.subplot(2,1,2)`
	`>>> plt.axvspan(5,14,alpha=0.1,color='g')`
	`>>> ax.bar(range(21),np.sum(tenSpikeTrains,0),1) #1`
	`>>> plt.xlabel('time (in milliseconds)')`
	`>>> plt.ylabel('# of spike occurrences at this time')`
	`>>> plt.savefig('Figure with subplots of rasters and PSTH.png')`
	`>>> plt.show()`

MATLAB	`>> subplot(2,1,2)`
	`>> rectangle('Position',[5,0,9,8],'FaceColor',[0.7 1 0.7]...`
	`,'linestyle','none')`
	`>> hold on`
	`>> x=0:20;`
	`>> bar(x,sum(tenSpikeTrains));`
	`>> xlim([-0.5 20])`
	`>> ylim([0 8])`
	`>> xlabel('time (in milliseconds)')`
	`>> ylabel('# of spikes counted at this time')`

Putting it all together:

Python	MATLAB
# The Python way for Figure 3.7.	% What does the analogous Matlab code look like?
fig=plt.figure()	figure
ax=plt.subplot(211)	subplot(2,1,1)
for trial in	rectangle('Position',[5,0,9,11],'FaceColor',[0.7 1
range(len(tenSpikeTrains)):	0.7],'linestyle','none')
spikeTimes = [i for i,x in enumerate(	for ii = 1:size(tenSpikeTrains,1)
tenSpikeTrains[trial]) if x==1]	spikeTimes = find(tenSpikeTrains(ii,:)==1)-1
plt.vlines(spikeTimes,trial,trial+1)	line(repmat(spikeTimes,2,1),repmat([ii-0.5; ii+0.5],1,length(spikeTimes)
plt.axvspan(5,14,alpha=0.1,color='g')	),'color','k')
plt.ylim([0,10])	end
plt.title('this neuron still spikes	xlim([-0.5 20])
to repeated trials of the same	ylim([0.5 10.5])
stimulus')	title('this neuron still spikes to repeated trials of the same
plt.xlabel('time (in milliseconds)')	stimulus')
plt.ylabel('trial number')	xlabel('time (in milliseconds)')
plt.yticks([x+0.5 for x in	ylabel('Trial number')
range(10)], [str(x+1) for x in	set(gca,'Layer','top')
range(10)])	subplot(2,1,2)
ax=plt.subplot(212)	rectangle('Position',[5,0,9,8],'FaceColor',[0.7 1
plt.axvspan(5,14,alpha=0.1,color='g')	0.7],'linestyle','none')
ax.bar(range(21),np.	hold on
sum(tenSpikeTrains,0),1)	x=0:20;
plt.xlabel('time (in milliseconds)')	bar(x,sum(tenSpikeTrains));
plt.ylabel('# of spike occurrences at	xlim([-0.5 20])
this time')	ylim([0 8])
# You may get syntax errors	xlabel('time (in milliseconds)')
# when continuing onto the next line.	ylabel('# of spikes counted at this time')
# Use the backslash character “\”	% End MATLAB code
# in Python to allow a line break in	
# the middle of code	
# End Python code for Figure 3.7	

Pseudocode

§ Begin English explanation of code for Fig. 3.7
Create the figure area
Specify that there are 2 rows and 1 column, and we'll start with the first
Plot the vertical ticks lines
Shade the area green to indicate the time of light stimulation
Set the lower and upper bounds of the y-axis
Set the title of the plot to 'this neuron still spikes to repeated trials of the same stimulus'
Set the x-axis label to 'time (in milliseconds)'
Set the y-axis label to 'trial number'
Set the y-axis tick locations and labels
Specify that we're making a subplot layout with 2 rows, 1 column, plotting in the 2nd row
Shade the stimulus area green
Make a histogram of all the spike times with bins from 0 to 20
Set the x-axis and y-axis labels
§ End English explanation of Python code for Fig. 3.7

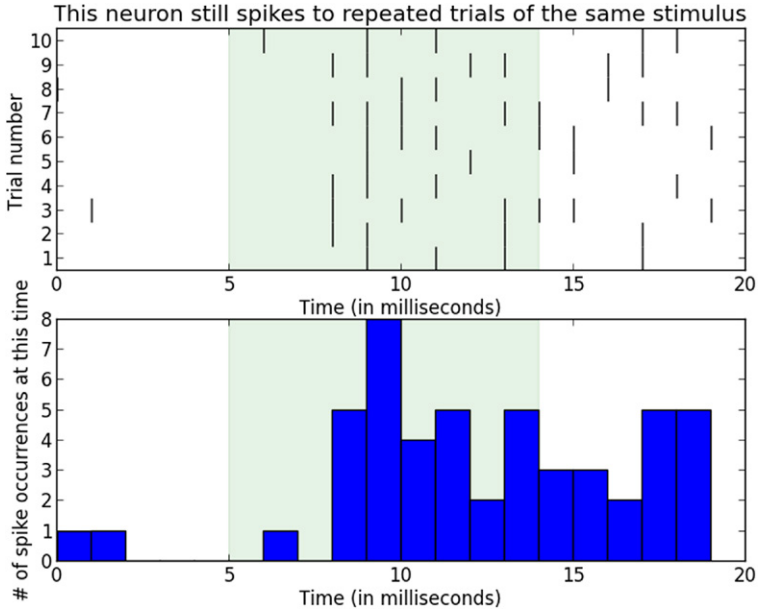

FIGURE 3.7 Raster plot (top panel) and PSTH (bottom panel) of neural response to visual stimulation in 10 trials.

Let's revisit our hypothesis, that the neuron fires *tonically at a rate of 500 spikes per second* to green light stimuli. We calculated that this would mean 4.5 spikes on average for the duration of our particular stimulus. But how can we infer this spike rate from the PSTH above? First, let us plot a new figure in which we scale the number of spike occurrences at each time by the number of stimulus trials (setting aside the rasters for a moment). This will show us, on average, how often the neuron spikes at each time point, and give us an estimate of the *spike probability* for each point in time.

We start by creating the figure, and omit the subplot line, noting that you don't need it for figures with single plots. Since we wanted to adjust the number of spikes for all the trials to form an estimate of *spike probability*, we will plot the `mean` of the spikes as a bar plot, instead of a `sum`.

If the neuron spiked, on average, at a rate of 500 spikes every second (every 1000 ms), then we might expect that for every millisecond there will on average be 0.5 spikes. Of course, we could not have performed this estimation with just a single trial as one cannot count half a spike. By measuring repeated trials, we form a more robust estimate of the spike rate over time, with the prediction (from our hypothesis) that the neuron will maintain a spike probability of 0.5 during the time that the stimulus is presented.

Let's draw a horizontal dashed black line (note the `linestyle` which can make it dotted or dashed, among other line types, and the *color* = *"k"* to de note *black*, we could have used *"r"* for red, or *"g"* for *green*, or *"b"* for *blue*) at the 0.5 spike probability threshold. Had we wanted a vertical line, we could have used the Python function `plt.axvline`.

`Pseudocode`	Create the figure Plot bar graph of the mean spikes Add horizontal line as a spike threshold

`Python`	`fig=plt.figure()`
	`plt.bar(range(21), np.mean(tenSpikeTrains,0),1)`
	`plt.axhline(y=0.5,xmin=0,xmax=20,linestyle='--',color='k')`

`MATLAB`	`figure`
	`bar(0:20,sum(tenSpikeTrains)./size(tenSpikeTrains,1));`
	`line(xlim,[0.5 0.5],'linestyle','--','color','k')`

Also label the axes and title, save the figure, and show it. Putting it all together, and using the simplified bar plot:

Python	MATLAB

`# The Python way for Figure 3.8`
`fig=plt.figure()`
`plt.axvspan(5,14,alpha=0.1,color='g')`
`plt.bar(range(21), np.mean(tenSpikeTrains,0),1)`
`plt.axhline(y=0.5,xmin=0,xmax=20,linestyle='--', color='k')`
`plt.title('Spike probability given 10 stimulus trials')`
`plt.xlabel('time (in milliseconds)')`
`plt.ylabel('probability of spike occurrences at this time')`
`plt.savefig('Figure 3.8normalized PSTH with cutoff.png')`
`# End Python code for Figure 3.8`
	`%What does the analogous Matlab code look like?`
	`figure`
	`rectangle('Position',[5,0,9,11],'FaceColor', [0.7 1 0.7],'linestyle','none')`
	`xlim([-0.5 20])`
	`ylim([0 1])`
	`hold on`
	`bar(0:20,sum(tenSpikeTrains)./ size(tenSpikeTrains,1));`
	`line(xlim,[0.5 0.5],'linestyle','--','color','k')`
	`title('Spike probability given 10 stimulus trials')`
	`xlabel('time (in milliseconds)')`
	`ylabel('probability of spiking at this time')`
	`% End MATLAB code`

In MATLAB, we simply normalize by the number of spike trains. Again, we see the power of MATLAB when handling matrices. Thus, represent something as a matrix whenever possible, as it will allow to bring powerful tools to bear.

Pseudocode
§ Begin English explanation of code for Fig. 3.8
Create the figure plotting area
Shade the area green to indicate the time of light stimulation
Plot a bar plot of the mean of tenSpikeTrains
Plot a dashed black horizontal line at y = 0.5
Set the title to "spike probability given 10 stimulus trials"
Set the x-axis label to "time (in milliseconds)"
Set the y-axis label to "probability of spike occurrences at this time"
Save the figure to "Fig. 3.8 normalized PSTH with cutoff.png"
§ End English explanation of code for Fig. 3.8

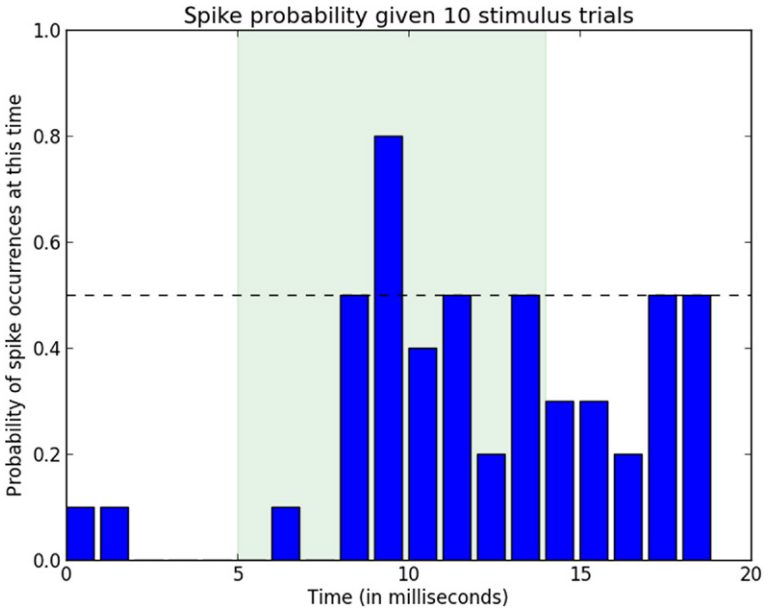

FIGURE 3.8 PSTH normalized by number of trials.

At first glance, it seems that our hypothesis of a consistent probability of 0.5 spikes over the stimulus interval does not hold and is false. We will discuss rigorous statistical tests at a later point. For now, we note that there are several phenomena that are not consistent with our hypothesis. For instance, there is a latency period before the spikes start, there is a phasic component of high spike probability around 4 ms and a tonically maintained probability around 0.4 or 0.5 for the remainder of the stimulus thereafter, even continuing for a while after the stimulus turns off. So it looks like things are more complicated than we initially expected, which (in biology) should be expected. To further illiminate what is going on, we could vary the intensity of the light, and hypothesize that the neuron fires more spikes with a shorter latency for brighter stimuli.

Let's load in the Python dictionary called `tenIntensities.pkl` (or in the case of MATLAB, the `.mat` file `tenIntensities.mat`), available for download via the companion website.

Python *dictionaries* are arranged in a manner where all of the *values* are assigned to a *key*. The *key* can be either a text string or a number, and the *value* can be almost anything: a number, string, list, array, or even another dictionary. To view the *keys* for this dictionary in Python , we type:

```
>>> tenIntensities.keys()
['4_intensity',
'2_intensity',
'8_intensity',
'0_intensity',
'7_intensity',
'5_intensity',
'9_intensity',
'6_intensity',
'3_intensity',
'1_intensity']
```

Each key corresponds to the intensity of the stimulus, ranging from 0 to 9. So to get the *values* for, say, the *key* `4_intensity`, we type:

```
>>> tenIntensities['4_intensity']
[[15.0, 15.0, 22.0, 25.0, 34.0, 23.0],
 [10.0, 32.0, 34.0, 22.0, 34.0],
 [13.0, 17.0],
 [9.0, 30.0, 36.0, 33.0],
 [8.0, 32.0, 31.0, 35.0, 19.0, 36.0, 19.0],
 [30.0, 13.0, 31.0, 36.0],
 [21.0, 31.0, 27.0, 30.0],
 [12.0, 15.0, 23.0, 39.0],
 [23.0, 30.0, 14.0, 23.0, 20.0, 23.0],
 [9.0, 16.0, 13.0, 27.0]]
```

We observe that each list within this value is another list of spike times. We use these spike times now to visualize the PSTHs over all stimuli.

Our raster and PSTH plotting techniques here are the same as before, with two main differences. The first, and most obvious from the output figure, is that we now have 20 subplots—10 rows and 2 columns. In the Python package `matplotlib`, the first subplot, referenced as *1*, is always at the top left. As we increase this index, our plot moves across each row to the right, to the end of the row, before moving down to the next column.

In the Python code, we make use of the `numpy` function `histogram`. It calculates the number of occurrences of values over a given range. It then returns the count of how many values occurred within each bin, and also returns the bins used, assigning these to the variables preceding the equals sign (and it doesn't plot anything). We use the variables obtained with `np.histogram` to make our histogram below with the function `bar`. In the code below we also give `numpy` a nickname, "`np`" which lets us refer to packages in a shorthand manner, e.g.: `np.histogram()`:

Python

```
# The Python way for Figure 3.9
import pickle
with open('tenIntensities.pkl', 'rb') as handle:
  tenIntensities = pickle.load(handle)
fig = plt.figure()
numIntensities = len(tenIntensities)
nbar={}
for key in tenIntensities.keys():
  ax=plt.subplot(numIntensities,2,float(key[0])*2+1)
  for trial in range(10): # this relies on there being 10
trials per stimulus intensity
plt.vlines(tenIntensities[key][trial],trial,trial+1)
  plt.xlim([0,20]);plt.ylim([0,10])
  plt.ylabel('intensity: '+str(key[0])+'\ntrial # ',style='it
alic',fontsize=5)
  plt.yticks(fontsize=5)
  plt.axvspan(5,14,alpha=0.1*float(key[0]),color='g')
  if float(key[0]) < 9:
    plt.xlabel('');plt.xticks([])
  else:
    plt.xlabel('time in milliseconds')
  if float(key[0]) == 0:
    plt.title('raster plot of spiking for each
intensity',fontsize=10)
  ax=plt.subplot(numIntensities,2,float(key[0])*2+2)
  plt.axvspan(5,14,alpha=0.1*float(key[0]),color='g')
  spikeTimes = [a for b in tenIntensities[key] for a in b] #1
  nOut,bins=np.histogram(spikeTimes,bins=range(20))
  nbar[float(key[0])] = nOut/10.
  plt.bar(bins[:-1],nOut/10.)
  plt.xlim([0,20]); plt.ylim([0,1])
  plt.yticks(fontsize=5)
  plt.ylabel('spike prob',style='italic',fontsize = 6)
  if float(key[0]) == 0:
    plt.title('PSTH for each intensity',fontsize=10)
  if float(key[0]) < numIntensities-1:
    plt.xlabel(''); plt.xticks([])
  else:
    plt.xlabel('time in milliseconds')
plt.savefig('Figure subplot 10 intensity rasters and psths.
png')
# End Python code for Figure 3.9
```

MATLAB

```
load('tenIntensities.mat')
figure
a= [1:2:20];
b =[2:2:20];
for ii = 1:size(A2,1)
subplot(10,2,a(ii))
if ii == 1
    title('raster plot for each intensity')
  end
rectangle('Position',[5,0,9,11],'FaceColor',
[1-(0.1.*ii) 1 1-(0.1.*ii)],'linestyle','none')
for jj = 1:10
spikeTimes = find(A2{ii,1}(jj,:)==1)-1
line(repmat(spikeTimes,2,1),repmat([jj-0.5;
jj+0.5],1,length(spikeTimes)),'color','k')
xlim([0 20])
ylim([0.5 10.5])
set(gca,'xtick',[])
end
end
%xlabel('time (in milliseconds)')
%ylabel('Trial number')
set(gca,'Layer','top')
for ii = 1:size(A2,1)
subplot(10,2,b(ii))
  if ii == 1
    title('PSTH for each intensity')
  end
rectangle('Position',[5,0,9,8],'FaceColor',
[0.7 1 0.7],'linestyle','none')
hold on
x=0:length(A2{ii,1})-1;
bar(x,sum(A2{ii,1}));
xlim([-0.5 20])
ylim([0 8])
set(gca,'xtick',[])
%xlabel('time (in milliseconds)')
%ylabel('# spikes')
end
```

Pseudocode

§ Begin English explanation of code for Fig. 3.9
Create new figure
Declare an empty dictionary *nbar*
For each key in the dictionary *tenIntensities*:

first column, raster plots for each intensity of the light stimulus. Plot in subplot corresponding to each intensity (an example *key* is: '*7_intensity*', so *key[0]* is the *0th* value of the string '*7_intensity*', which is '*7*', and *float('7')* equals 7.0, We take that value times two and add one since subplot indices count by row. For each trial this relies on there being 10 trials per stimulus intensity. Plot vertical lines corresponding to the spike times

Format the raster plots: Set the x- and y-axis limits. Set the y-axis label to the intensity, use '\n' as a carriage return, label the trial number, italic and fontsize, set the yticks fontsize. Add the green box, use the *alpha* value so that the transparency scales with intensity. If the intensity is less than 9, that is, if we are not plotting at the bottom. Do not label the x-axis.

Else, that is, if the intensity is 9. Label the x-axis as 'time in milliseconds'

If the intensity is 0, that is, if we are plotting at the top. Set the title to 'raster plot of spiking for each intensity' and font size to 10.

#1 perform list comprehension to unpack list

In the second column, plot the PSTHs for each intensity of the light stimulus. Plot the subplot in the second column, with each increasing intensity moving down a row. Plot the green box and set the *alpha* value to correspond to the intensity of the stimulus. Extract all the spike times for a stimulus intensity. Get *nOut*, a histogram array binned by the value *bins*. Add the values in *nOut/10*. to the dictionary *nbar* with key *float(key([0]))*. Plot the PSTH with *bar* function, calling all the bins except the last bin, and scaling *nOut* by the number of trials (10) .

Format the PSTHs: Set the x-axis and y-axis limits to [0,20] and [0, 1], respectively. Set the y-axis font size. Set the y-label to 'spike prob, make it *italic*, set the fontsize to 6. If we are in the first row (at the top). Set the title to 'PSTH for each intensity', with a fontsize of 10

If we are in any plot above the bottom plot, turn of the xlabel and xticks.

Else if we are at the bottom plot, set the xlabel to 'time in milliseconds'

Save the figure

§ End English explanation of code for Fig. 3.9

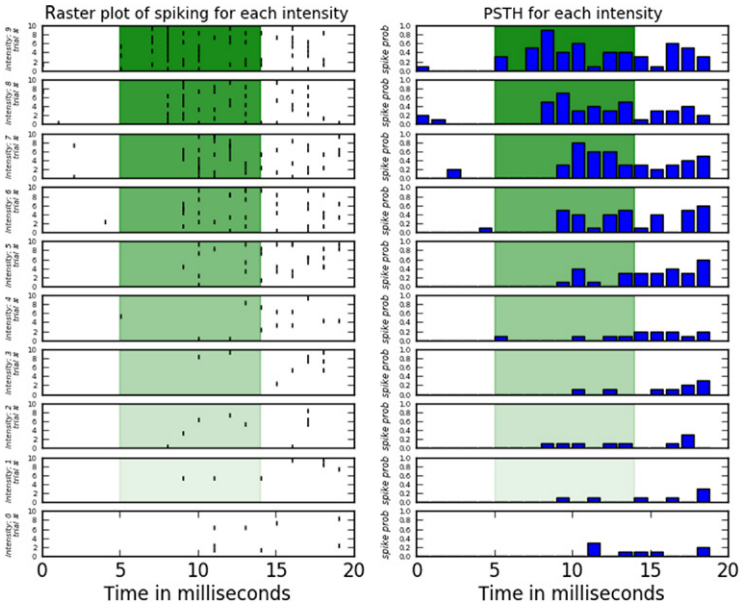

FIGURE 3.9 Raster plots (left panels) and normalized PSTHs (right panel) for stimuli of varying light intensities.

In the Python code, we slipped in the initialization of the *dictionary* `nbar`, which we did with curly braces `{}`. Later in the code, we assign *values* from `nOut,` which represent the number of occurrences of spikes for particular times, to *keys* of `nbar`. We access all the values of `nbar` below with `nbar.values()`.

Our ability to measure *latency to first spike* here becomes quite difficult. We can qualitatively say that higher-intensity stimuli cause shorter latency responses. We will leave any swings at statistical inference to later chapters, and relish for now in our ability to make a colorful plots from the spike data.

Python	MATLAB

`# Begin the Python way for Figure 3.10`
`fig = plt.figure()`
`ax = plt.subplot(111)`
`aa = ax.imshow(nbar.values(),cmap='hot',`
`interpolation='bilinear')`
`plt.yticks([x for x in range(10)],[str(x) for x in`
`range(10)[::-1]])`
`plt.ylabel('stimulus intensity')`
`plt.xlabel('time in milliseconds')`
`plt.title('heat map of mean spiking for various`
`intensity stimuli')`
`cb = fig.colorbar(aa,shrink=0.5)`
`cb.ax.set_ylabel('mean spikes per time bin')`
`# End Python way for Figure 3.10`
	`for ii = 1:size(A2,1)`
	`  A3(ii,:) = (sum(A2{ii,1})./10);`
	`end`
	`A3(:,22:100) = [];`
	`figure`
	`h = pcolor(A3);`
	`set(h,'Facecolor','interp')`
	`set(h,'Linestyle','none')`
	`set(gca,'YDir','reverse')`
	`colormap('hot')`
	`h = colorbar;`
	`ylabel(h, 'mean spikes per time bin')`
	`xlabel('time (in milliseconds)')`
	`ylabel('stimulus intensity')`
	`title('heat map of mean spiking for stimuli of varying`
	`intensity')`

Pseudocode

§ Begin English explanation of code for Fig. 3.10

Create plotting area

Specify that we're making one subplot

Plot the values of *nbar* as an image with a hot colormap and the colors bilinearly interpolated

Set where the y-ticks go and what their labels are

Set the y-*a*xis label to 'stimulus intensity'

Set the x-axis label to 'time in milliseconds'

Set the title to 'heat map of mean spiking for various intensity stimuli'

Create a colorbar, use the *shrink* command to customize its height

Set the colorbar label to 'mean spikes per time bin'

§ End English explanation of code for Fig. 3.10

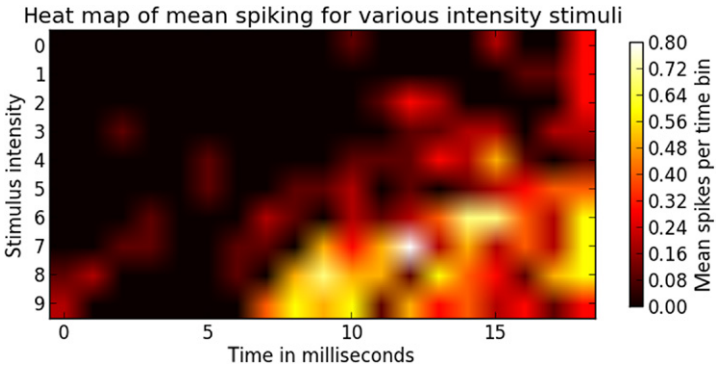

FIGURE 3.10 Heat map of mean spiking activity over time as a function of stimulus intensity.

`Pseudocode`	Create figure plotting area
	Specify two rows and two columns to plot, select the first one
	Plot the bar values as image with colormap hot, bilinear interpolation, and aspect 1.2
	Turn the y-axis and x-axis tick marks off
	Specify two rows and two columns to plot, select the second one
	Plot the bar values as image with colormap bone, nearest interpolation, and aspect 1.2
	Turn the y-axis and x-axis tick marks off
	Specify two rows and two columns to plot, select the third one
	Plot the bar values as image with colormap jet, bicubic interpolation, and aspect 1.2
	Turn the y-axis and x-axis tick marks off
	Specify two rows and two columns to plot, select the fourth one
	Plot the bar values as image with colormap cool, nearest interpolation, and aspect 1.2
	Turn the y-axis and x-axis tick marks off
	Save figure

(*Continued*)

Python

```
fig = plt.figure(); ax = plt.subplot(221)
aa = ax.imshow(nbar.values(),cmap='hot' ,interpolation='bilinear',aspect=1.2)
plt.yticks([]); plt.xticks([])
ax = plt.subplot(222)
aa = ax.imshow(nbar.values(),cmap='bone', interpolation='nearest',aspect=1.2)
plt.yticks([]); plt.xticks([])
ax = plt.subplot(223);
aa = ax.imshow(nbar.values(),cmap='jet', interpolation='bicubic',aspect=1.2)
plt.yticks([]); plt.xticks([])
ax = plt.subplot(224)
aa = ax.imshow(nbar.values(),cmap='cool',
interpolation='nearest',aspect=1.2)
plt.yticks([]); plt.xticks([])
plt.savefig('Figure 3.11- 4 heatmaps labels off.png')
```

MATLAB

```
figure
ax1 = subplot(2,2,1)
h = pcolor(A3);
set(h,'Facecolor','interp')
set(h,'Linestyle','none')
set(gca,'YDir','reverse')
colormap(ax1,'hot')
axis off
ax2 = subplot(2,2,2)
h = pcolor(A3);
set(h,'Linestyle','none')
set(gca,'YDir','reverse')
colormap(ax2,'bone')
axis off
ax3 = subplot(2,2,3)
h = pcolor(A3);
set(h,'Facecolor','interp')
set(h,'Linestyle','none')
set(gca,'YDir','reverse')
colormap(ax3,'jet')
axis off
ax4 = subplot(2,2,4)
h = pcolor(A3);
set(h,'Linestyle','none')
set(gca,'YDir','reverse')
colormap(ax4,'winter')
axis off
```

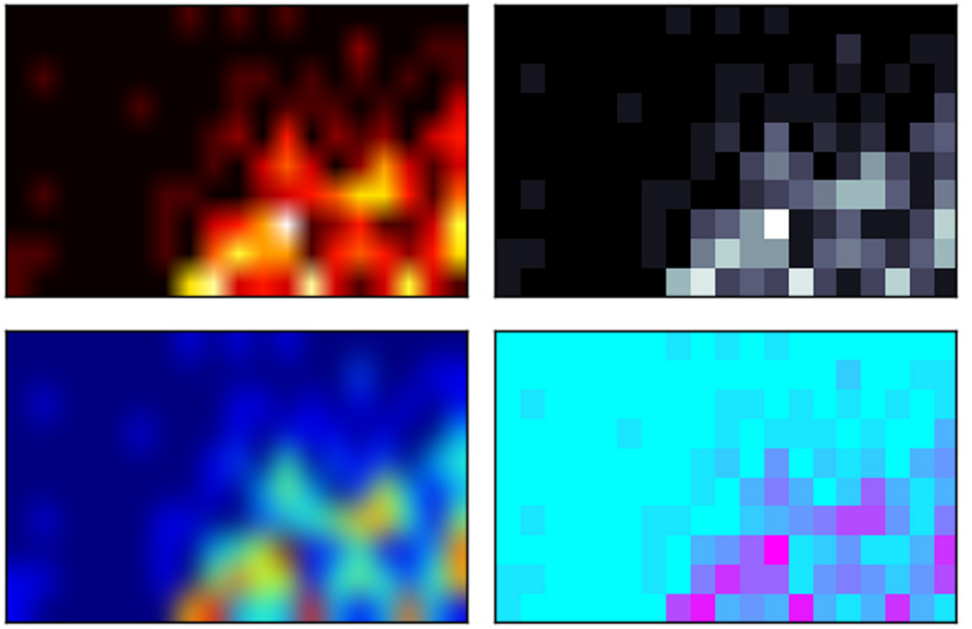

FIGURE 3.11 Heat maps with different color maps. Top left: "Hot", Top right: "Bone", Bottom left: "Jet", Bottom right: "Winter".

That's it for basic spike wrangling. Admittedly, this was somewhat of a toy example, but we have to get started somewhere. You'll find in your own dataset that you will need to employ a variety of wrangling methods to achieve your desired outcome—the goal here is to learn the various tools that help you to do so. In the very next chapter, Chapter 4, Correlating Spike Trains, we'll address a full-blown data analysis project, also involving spikes and their wrangling, but recorded from many neurons with many electrodes simultaneously.

QUESTIONS WE DID NOT ADDRESS

In the 10 intensity stimulus–response set, what order were the stimuli presented in? Did all of the bright intensity stimuli occur sequentially before moving to a lower intensity stimulus? If so, might the ordering of the stimuli influence the neuron's response characteristics? Might the neuron exhibit *adaptation*, where its firing probability adapts as a function of previous stimuli and its internal characteristics? How much time of no stimuli was between trials—what was the *inter-trial interval,* or ITI? How might changing the ITI influence the neuron's spiking?

What other optogenetic tools could we use to study the activity of single neurons? How many photons are required for the activation of ChR2? What sort of latency of activation patterns would we expect for other optogenetic tools? What sort of experimental preparation would be required to mimic the experiment in this chapter? What sort of experimental apparatus would be necessary to record from a single neuron?⌘

⌘. *Pensee on the proper unit of measurement for the spiking activity (firing rate) of single neurons*: Data result from the outcome of a measurement process. A unit of measurement is the fundamental unit which we use to express the quantity of a given quality. For instance, the currently agreed upon unit of measurement for length in the SI system is the meter, defined as "the length of the path travelled by light in vacuum during a time interval of 1/299792458 of a second." (Taylor & Thompson, 2001). Consequently, all lengths that we wish to measure are then expressed in multiples of this reference length, e.g. 2 meters or 0.5 meters. This raises the question what the appropriate unit of measurement for a firing rate is. The firing rate of a neuron in response to a given stimulus (or even in the absence of a stimulus) is a quality of the neuron. The implication is that the neuron in question discharges action potentials (or "spikes") a certain number of times in a given interval, usually a second. The choice of this interval is probably what led to the fundamental confusion that one often sees in neuroscience publications. Firing rates are frequently expressed in terms of "Hz," e.g. "the neuron fired at a rate of 30 Hz." The "Hz" in question refers to the unit of measurement of a periodic, oscillatory process, namely 1 cycle (or period) per second. Unfortunately, this fundamentally mischaracterizes the very nature of action potentials. In contrast to harmonic oscillators, e.g., the motion of guitar strings, action potentials are neither cyclical nor periodic. Most frequently, they are conceptualized as "spikes," or point processes, in which case only the time when they occurred and how often this happened in a given interval is meaningfully interpretable. Spiking activity of single neurons is notoriously aperiodic and highly irregular—interspike intervals in a spike train are close to what would be expected from a Poisson process (Softy & Koch, 1993) and the variance of spike counts upon repeated stimulation suggests overdispersion (Taouali et al., 2016). Finally, it makes practical sense to avoid expressing firing rates in Hz in order to simply avoid the potential confusion when plotting it simultaneously with quantities that are appropriately expressed in Hz, such as the temporal frequency of a stimulus or the power of an analog signal in a certain frequency bin, as we'll encounter in Chapter 5. The debate about the theoretical significance of neural oscillations is heated enough (Shadlen & Movshon, 1999) without implying that spike rates are inherently oscillatory as well. But if not Hz, what is the proper unit of firing rate? As spikes are typically defined by the voltage trace recorded from an electrode in the brain crossing a reasonable threshold (and recorded as the time at which this crossing happened) then counted, it does make sense to express in units of impulses

per second (ips) or spikes per second (sp/s) or simply events (threshold crossings) per second. All of these are conceptually sound and it is perhaps this range of equally suitable available options that prevented any of them from catching on as a consensus. In military contexts, the "rate of fire" (of rapid-firing guns) is typically expressed in rounds per second (rps), so by analogy, spikes per second (which is what we care about in a firing rate) is perhaps the most apt. Historically, there has been a movement to replace these with the eponym "Adrians," in honor of the pioneering Lord Edgar Douglas Adrian, the original discoverer of the rate code (Adrian, 1926) who won the Nobel Prize in 1932 and is the great-grandfather of many a neurophysiologist, if neurotree.org is to be believed. However, this unit did not catch on either—given the problematic nature of eponyms, this is perhaps just as well (Wallisch, 2011)—but almost anything would be better than expressing firing rates in Hz, which is fundamentally misleading. To repeat: a rate is not a frequency. For actual frequencies, the entire signal scales with it, if the frequency changes. In contrast, the individual action potentials remain invariant, regardless of spike rate. These are fast events, which have to be sampled frequently (or at high frequency) in order to be captured, even (or particularly) if the actual spike rate is very low. With this in mind, "spikes per second" really makes the most sense.

This page intentionally left blank