1 Introduction to MATLAB

Synopsis

Data Simple numerical examples.

Goal Introduce basic operations in MATLAB.

Tools Create vectors and arrays, load data, make plots.

1.1 Introduction

In this book, we use the software package MATLAB. The best way to learn new software (and probably most things) is when you are motivated by a particular problem. In subsequent chapters, we pursue specific questions driven by neural data, and use our desire to understand these data to motivate the development and application of computational methods.

This chapter focuses on basic coding techniques and principles in MATLAB. In examples, you are asked to execute code in MATLAB. If your MATLAB experience is limited, you should actually *do* this, not just read the text.

This chapter follows in spirit and sometimes in detail chapter 2 of *MATLAB for Neuroscientists*, an excellent reference for learning to use MATLAB with many additional examples [1]. If you have never used MATLAB before, you might examine the *MATLAB Primer* (available on the MathWorks website).

1.2 Starting MATLAB

To begin, gain access to MATLAB. To do so, visit the MathWorks website, http://www.mathworks.com/, and check for licensing options from your local institution, if relevant. We assume in this book that you have access to MATLAB, the ability to launch the software, and a basic familiarity with the programming environment.

1.3 MATLAB Is a Calculator

Enter the following command at the MATLAB prompt:

```
4+3
```

Q: What does MATLAB return? Does it make sense?

You may notice that the font for `4+3` differs from the rest of the text. This font indicates MATLAB code. In this case, you are being asked to enter the command `4+3` at the MATLAB prompt, execute the command, and evaluate the output.

1.4 MATLAB Can Compute Complicated Quantities

Enter the following command at the MATLAB prompt:

```
4/10^2
```

Q: What does MATLAB return? Does it make sense? You may use parentheses to alter the result. How does

```
(4/10)^2
```

compare to your previous answer?

1.5 Built-in Functions

A *function* is a program that operates on inputs. In the following,

```
sin(2*pi)
```

`sin` is a function that computes the sine of its input; in this case, the input is `2*pi`. Here are three more examples of functions that operate on inputs:

```
cos(2*pi + 1/10)
exp(-2)
atan(2*pi)
```

Q: What is the function `atan`?

A: Try using MATLAB Help. For example, at the MATLAB prompt, type

```
help atan
```

to read a brief description of the function `atan`.

The MATLAB `help` function is extremely useful. It's always a good place to look when you have questions about a function.

1.6 Vectors

A *vector* is a list of numbers. Let's define a vector that consists of four numbers:

```
[1 2 3 4]
```

Q: What happens when you execute this above code in MATLAB?

1.7 Manipulating Vectors with Scalars

A *scalar* is a single number. Consider

```
[1 2 3 4] * 3
```

which consists of the product of a vector (`[1 2 3 4]`) and a scalar (`3`).

Q: Upon executing this statement, what do you find? Does it make sense?

1.8 Manipulating Vectors with Vectors

Consider

```
[1 2 3 4] .* [1 2 3 4]
```

Q: Does this operation return a vector or a scalar? Does the result make sense?

The notation `.*` is used to multiply. This causes the vectors to be multiplied element by element. Namely, the first element of the first vector is multiplied by the first element of the second vector (i.e., `1*1`); the second element of the first vector is multiplied by the second element of the second vector (i.e., `2*2`); the third element of the first vector is multiplied by the third element of the second vector (i.e., `3*3`); and so on. To perform this calculation, the vectors must have the same number of elements. Consider

```
[1 2 3 4] .* [1 2 3 4 5]
```

Q: What happens when you execute this expression?

A: The expression should fail to execute. To understand why, consider the number of elements in each vector. The second vector has one additional element (the scalar `5`) compared to the first vector. So, if we try to multiply the two vectors element by element, we cannot; there is no value in the first vector to multiply the `5` in the second vector.

The operator `.*` differs from the operator `*`. The latter operator performs matrix multiplication, which is very different from the element by element multiplication we just performed. If you've taken a class in linear algebra, then you've seen matrix multiplication. We do not use matrix multiplication extensively here, but it is an essential tool in more advanced data analysis. The important point is the following.

Alert! The operators `.*` and `*` are very different.

1.9 Defining Variables

In the previous examples, we typed `[1 2 3 4]` over and over again. A better approach is to assign a *variable* to this vector and then simply reference this variable to access the vector. Consider

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

Here we have defined two variables. The variable `a` is assigned the number `2` and is therefore a scalar. The variable `b` is assigned the vector `[1 2 3 4]` and is therefore

a vector. To work with this scalar and vector, we can now refer to the variables. Consider

```
c = a*b
d = b.*b
```

Q: What are the results in the new variables `c` and `d`?

1.10 Probing the Defined Variables

Once we define variables, we often want to check the size of these variables. A good place to start is the *workspace window* of MATLAB. Look for this window on your desktop.

Q: What variables do you see? What information do you learn about the variables by examining the workspace?

To see a list of the variables you've defined, type

```
who
```

To determine the size of a variable, for example, the variable `c` defined in the previous section, type

```
size(c)
```

1.11 Summing Elements in a Vector

Sometimes we need to add up all the elements in a vector. In simple cases, we can perform this operation by hand. For example, consider the vector `c` used in the previous example. To recall the values in `c`, type `c` at the MATLAB command prompt and evaluate it:

```
c
```

To sum the elements of `c`, we may examine the output of the previous command, which lists all the values in `c`, and add these values by hand. Doing so is not difficult, but it is tedious. You might imagine the difficulty of this approach when the vector to sum contains many elements. For example, manually computing the sum of a vector containing 100 elements is not especially difficult but highly error prone. Fortunately, there's an easier way: we can use the MATLAB command `sum`:

```
sum(c)
```

Q: Notice that we've used a new function: `sum`. What is the input to this function? What is the output of this function? You might consider `help sum`.

Q: Compare the result of the MATLAB command `sum(c)` and your by-hand sum. Are the two the same? *Hint*: They should be.

1.12 Clearing All Variables

To clear all variables from the workspace, enter

```
clear
```

Q: Are all the variables gone? How can you check? *Hint*: Consider the command `who`.

1.13 Matrices

A more complicated list of numbers contains multiple rows or columns. This is called a *matrix*. We can think of a matrix as a group of vectors. Consider the following:

```
p = [1 2 3; 4 5 6]
```

This operation creates a matrix with two rows and three columns.

Q: How can you verify the size of `p` in MATLAB? *Hint*: Consider the command `size`.

We can manipulate matrices just as we manipulate vectors. Consider adding the scalar `2` to the matrix:

```
p + 2
```

Q: What happens?

Q: How would you create a matrix with three rows and four columns? Create this matrix, and verify the size using the command `size`.

1.14 Indexing Matrices and Vectors

Matrices and vectors are lists of numbers, and sometimes we want to access individual elements or small subsets of these lists. That's easy to do in MATLAB. Consider

```
a = [1 2 3 4 5]
b = [6 7 8 9 10]
```

To access the second element of `a` or `b`, enter

```
a(2)
b(2)
```

Q: Do the results make sense? How would you access the fourth element of each vector?

We can combine `a` and `b` to form a matrix.

```
c = [a; b]
```

To learn about `c`, enter

```
size(c)
```

The size of `c` is [2 5]. It has two rows and five columns. To access the individual element in the first row and fourth column of `c`, type

```
c(1,4)
```

We access matrices using (row, column) notation. So `c(1,4)` indicates the element in row 1, column 4, of `c`.

To access all columns in the first row of `c`, enter

```
c(1,:)
```

The notation `:` means "all indices". Here we use this notation to indicate all columns of `c`. To access the second through fourth columns of row 1 of `c`, type

```
c(1,2:4)
```

The notation `2:4` means "all integers from 2 to 4," which in this case is 2, 3, 4.

Q: How could you access all rows in the second column of `c`?

1.15 Finding Subsets of Elements in Matrices and Vectors

Sometimes we're interested in locating particular values within a matrix or vector. For example, let's first define a vector:

```
b = [1,2,3,4,5,6,7,8,9,10]*2
```

Q: What is the size of the variable `b`? What are the values in the variable `b`?

There's an easier way to define the same list of numbers. Consider the command

```
a = (1:1:10)*2
```

Q: What is the size of `a`? What are the values in `a`? How do the values in `a` compare to the values in `b`?

Now let's find the indices for all values in `a` that exceed 10. We could, of course, do so manually.

Q: Determine by inspection the indices of `a` that exceed 10. What are they?

There's a simple way to perform this same operation in MATLAB,

```
indices = find(a > 10)
```

Q: `find` is a function built into MATLAB. What does this function do? *Hint*: Consider `help find`.

Q: What is now stored in the variable `indices`?

A: The variable `indices` indicates all indices of the vector `a` with values greater than `10`. To verify this, examine the value of `a` at each index in `indices`; it should be greater than `10` for each index.

We can examine the values of `a` at the `indices`:

```
a(indices)
```

Q: What do you find? Do all the values exceed `10`?

Beyond simple examinations, we can also manipulate the values of `a` at the `indices`. For example,

```
a(indices) = 0
```

Q: After performing this operation, what do you find is the largest value stored in the vector `a`? What has happened to all values of `a` that exceed `10`?

1.16 Plotting Data

It's usually not easy to look at lists of numbers and gain an intuitive feeling for their behavior, especially when the lists contain many elements. In these cases, it's much better to visualize the lists of numbers in a plot. Consider

```
x = (0:1:10)
```

This above line constructs a vector that starts at 0, ends at 10, and takes steps of size 1 from 0 to 10. Then define

```
y = sin(x)
```

We have now applied the function `sin` to the list of numbers in `x`.

Q: Look at the values of `y` printed as output upon evaluating the line of code defining `y`. Can you deduce the behavior of `y`?

A: A list of the values in `y` is not particularly informative. Let's examine a printout of `y`:

```
0 0.8415 0.9093 0.1411 -0.7568 -0.9589 -0.2794 0.6570
0.9894 0.4121 -0.5440
```

Examining this list, we get a sense for the values of `y`. We find an interval of some positive values, followed by an interval of negative values, and so on. However, it's difficult to get a deep intuitive sense for the behavior of `y` through inspection of this printout alone.

To visualize y versus x, execute

```
plot(x,y)
```

This command produces a plot of x versus y (see figure 1.1a). Visual inspection reveals the curve is a bit jagged, not smooth like a sinusoid. To make the curve smoother, let's redefine x as

```
x = (0:0.1:10);
```

Q: Compare this definition of x to the previous definition. How do these two definitions of x differ?

Q: What is the size of x now? Does this make sense?

You might notice that we've added a semicolon (;) to the end of the definition of the vector x. This semicolon tells MATLAB not to print out the results of this command in the command window. We don't want to write out x because it contains lots of elements, and it's not useful for us to look at them right now. Now recompute y as

```
y = sin(x);
```

We again include the semicolon at the end to prevent a printout of many not currently informative values. Now, let's plot the two new variables:

```
plot(x,y)
```

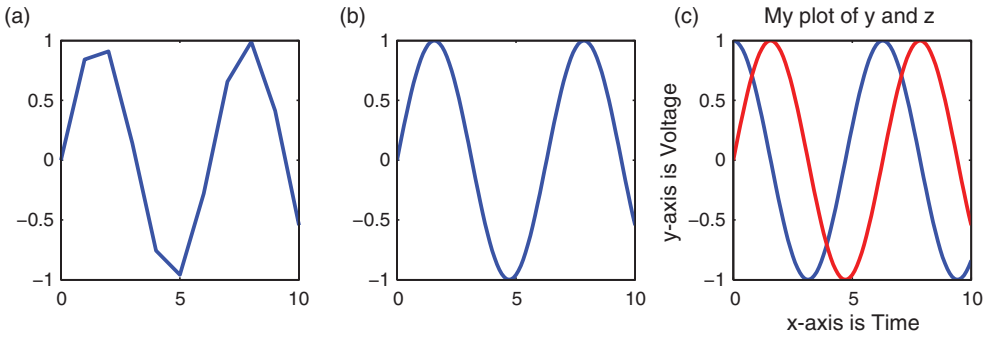

Figure 1.1
Plots of x versus y for (*a*) low density of x, and (*b*) high density of x. (*c*) a Plot of x, y, and z.

Q: Compare the newly created plot (figure 1.1b) with the original plot (figure 1.1a). How do the two differ?

1.17 Multiple Plots, One atop the Other

Continuing the example in the previous section, let's define a second vector

```
z = cos(x);
```

and plot it:

```
plot(x,z)
```

We'd now like to compare the two variables `y` and `z`. To do this, let's plot both vectors on the same figure. First, open a new figure:

```
figure
```

and plot `x` versus `z`:

```
plot(x,z)
```

Now, tell MATLAB to freeze, or "hold," the graphics on this figure:

```
hold on
```

With the graphics now frozen, let's also plot on this figure `x` versus `y`:

```
plot(x,y, 'r')
```

Notice that we've included a third input to the function `plot`. Here the third input tells MATLAB to draw the curve in a particular color: `'r'` for red. There are many options we can use to plot; to see more, check out `help plot`. When we're done *holding* the figure, it's polite to turn "hold" off:

```
hold off
```

We can also label the axes and give the figure a title:

```
xlabel('x-axis is Time')
ylabel('y-axis is Voltage')
title('My plot of y and z')
```

1.18 Random Numbers

To generate a single random number in MATLAB, type

```
randn(1)
```

Q: Execute this command a few times. What values do you get?

Sometimes, we would like to generate a list of random numbers. To do so, we can again use the `randn` command but this time with different inputs. Consider the following line of code:

```
r = randn(10,1)
```

In this line, we again call the function `randn`. But this time we call the function with two inputs, `(10,1)`, and store the results in the variable `r`.

Q: What is the size of the variable `r`? What numbers do you find in `r`?

Q: What happens to the size of `r` as you change the values of the inputs? Consider, for example,

```
r = randn(100,1);
r = randn(10,10);
```

What is the size of `r` in each case? What are the types of values you observe in `r` in each case?

We sometimes need to generate long lists of random numbers. We can do so using the `randn` command as follows:

```
r = randn(1000,1);
```

Q: How many elements are in the vector `r`?

To see how these random numbers are distributed, we can visualize the results. Let's plot the variable `r`:

```
plot(r)
```

Q: Examine this plot. What values does `r` take on? What is the biggest value of `r`? the smallest? the most common?

1.19 Histograms

There are many ways to display the data in the variable `r`. Another way is to construct a *histogram*. A histogram displays the number of times we observe a value in a list. Consider the following list of numbers:

```
a = [0,0,0, 1,1,1,1, 3,3];
```

It's easy to construct a histogram of the values in `a` using the built-in MATLAB function `hist`:

```
hist(a, (0:1:5))
```

Q: The result of this command is to produce the plot in figure 1.2. Examine this plot. What do you find?

The histogram (figure 1.2) indicates the number of times we observe each value in the variable `a`. In this case, we observe the values

0, observed three times;

1, observed four times;

3, observed two times.

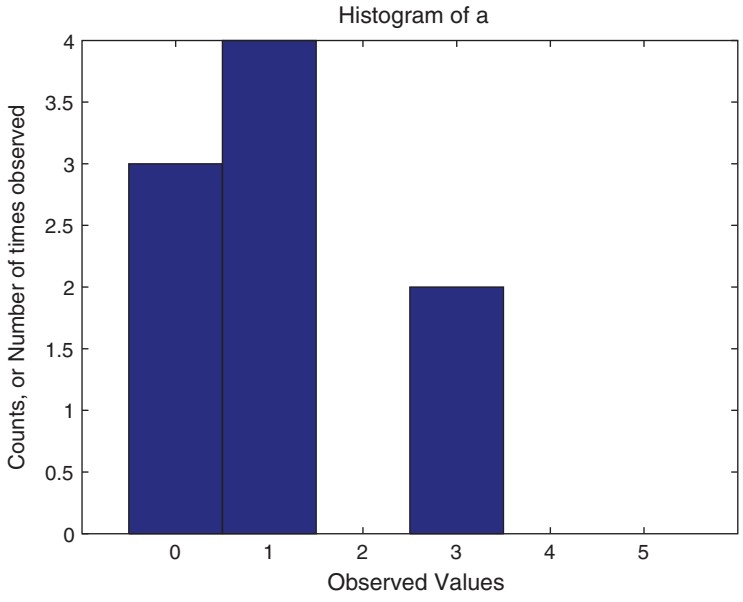

Figure 1.2

Histogram of variable `a`. Horizontal axis indicates values in `a`; vertical axis, number of times each value appears.

In the histogram plot, there are three bars. The first bar has an x-coordinate of 0 and a y-coordinate of 3. This tells us that in the variable `a`, we observe the value 0 three times. In the `hist` command, we specify two inputs:

first input: `a`, the variable of interest;

second input: the histogram *bins*, which specify the center value of each bin.

In this example, for the second input, we specify six bins: the bins start at a center value of 0, and step forward in values of 1, up to a center value of 5. For completeness, let's label the histogram:

```
xlabel('Observed Values')
ylabel('Counts, or Number of times observed')
title('Histogram of a')
```

Q: Consider the x-coordinate of 1 in the plot. Do you observe a bar at this x-coordinate? If so, what is the height? How does this height correspond to the number of 1s you observe in your list `a`? What about the x-coordinate of 2? the x-coordinate of 3?

Q: Consider how the histogram plot changes as the second input to `hist` (the bins) is varied. More specifically, consider `hist(a, (0:2:5))` and `hist(a, (0:0.1:5))`. Compare the histograms you create in each case. Do they make sense?

Finally, let's return to the list of random numbers (generated in section 1.18) that we stored in the variable `r`, and plot a histogram:

```
hist(r, (-5:0.5:5))
```

Q: Consider the histogram plot of `r`. What is the most common value, that is, which value has the most counts?

Q: How does the histogram of `r` change as you vary the bins (the second input to the `hist` command)?

See `help hist` to learn about the function `hist`.

1.20 Repeating Commands

Sometimes we want to repeat the same command over and over again. For instance, if we want to plot `sin(x + k*pi/4)`, where `k` varies from 1 to 5 in steps of 1, how do we do it? Consider the following (in which the % symbols indicate comments; comments are not evaluated by MATLAB, but help explain the code):

```
x = (0:0.1:10);        %Define x from 0 to 10 with step 0.1.
k=1;                   %Fix k=1,
y = sin(x + k*pi/4);   %... and define y at this k.

figure                 %Now, make a new figure window,
plot(x,y)              %... and plot y versus x.

k=2;                   %Let's repeat this, for k=2,
y = sin(x + k*pi/4);   %... and redefine y at this k,
hold on                %... and plot it.
plot(x,y)
hold off

k=3;                   %Let's repeat this, for k=3,
y = sin(x + k*pi/4);   %... and redefine y at this k,
hold on                %... and plot it.
plot(x,y)
hold off

k=4;                   %Let's repeat this, for k=4,
y = sin(x + k*pi/4);   %... and redefine y at this k,
hold on                %... and plot it.
plot(x,y)
hold off

k=5;                   %Let's repeat this, for k=5,
y = sin(x + k*pi/4);   %... and redefine y at this k,
hold on                %... and plot it.
plot(x,y)
hold off
```

This code is correct but horrible. To execute our objective, we cut and paste the same lines of code over and over again. As a general rule, if you're repeatedly cutting and pasting

code, you're doing something inefficient and typically error prone. There's a much more elegant way to achieve the objective, and it involves making a *for-loop*:

```
x = (0:0.1:10);           %First, define the vector x,
figure                    %... and open a new figure.
for k=1:1:5               %For k from 1 to 5 in steps of 1,
    y = sin(x + k*pi/4);  %... define y (note 'k' in sin),
    hold on               %... hold the figure,
    plot(x,y)             %... plot x versus y,
    hold off              %... and release the figure.
end
```

This small section of code replaces the code involving repeated cutting and pasting. Here, we update the definition of `y` with different values of `k` and plot it within a for-loop.

Q: Spend some time studying this for-loop. Does it make sense?

Some of the subsequent code we develop includes for-loops. So, it's useful to be familiar with this type of operation.

1.21 Defining a New Function: The m-File

So far, we haven't explicitly saved any code. Instead, we entered all commands directly into the MATLAB command line. Sometimes, especially when developing a series of complex commands, we want to save the code so as not to have to reenter it. In other cases, we want to define and save new MATLAB functions for specific purposes.

A powerful feature of MATLAB allows us to create our own functions to achieve our goals. As an example, let's write a new MATLAB function: the function will take as input a vector and scalar, and return as output each vector element squared plus the scalar. Ideally, this function would be called in MATLAB as follows:

```
v = (0:1:10);                  %Define a vector.
b = 2.5;                       %... and a scalar,
v2 = my_square_function(v,b);  %... as inputs to the function.
```

However, these lines of code do not yet work. We first need to define the new function `my_square_function`. To do so, we create a new file in the MATLAB editor. Let's begin by entering at the MATLAB command prompt,

```
edit
```

MATLAB will open a new file called `Untitled` in the editor. This file is initially blank (i.e., it contains no text). We use this blank template to create the new function. Type into the `Untitled` file the following three lines of code:

```
function output = my_square_function(input1, input2)
  output = input1.^2 + input2;
end
```

In the first line, we define labels for the input and output variables to the our function. The keyword `function` lets MATLAB know we're defining a new function. We define the function name (`my_square_function`) and specify this function to have two inputs (`input1` and `input2`). If we desired more inputs, we would simply add additional variables in the parentheses that follow the function name. We also specify a single output for this function (`output`). The syntax of this first line of code makes the relation of inputs and outputs clear; the output is equal to the function evaluated at the inputs.

Q: Consider the second line in this code. It defines the body of the new function and performs the actual work of the desired computation. Does this line achieve the stated goal of this function—to return as output the vector elements squared plus the scalar?

The last line of code, `end`, serves to end the function definition.

Q: Consider changing the variable names of the inputs from `input1` and `input2` to `a` and `b`, respectively. How must we change the function to perform the same computation?

A: The input variables are called dummy variables. We're free to chose these names to be (almost) anything. It's good practice to choose informative names, not confusing names. For example, a confusing name for an input variable would be `output`. That choice of name is not wrong but aesthetically unpleasant. Such a choice may make it difficult for a colleague (or your future self) to understand the function's behavior. Changing the variable names requires us to change the function as follows:

```
function output = my_square_function(a, b)
  output = a.^2 + b;
end
```

Notice that we've simply replaced each occurrence of `input1` with `a`, and each occurrence of `input2` with `b`. The resulting function behaves exactly the same as the original function. We have only changed some of the internal labels to the function, which has no impact on the desired computation.

Having input these three lines of code, we save this m-file with the name `my_square_function.m`. Notice the `.m` file extension, which indicates a MATLAB function or script. You might perform this save through key strokes (e.g., Command-S or Control-S, depending on your operating system) or by using the mouse.

The file name and function name must match. In this case, the file name (`my_square_function.m`) matches the function name (`my_square_function`) that we call at the MATLAB command line prompt.

To call this function, we first need to navigate MATLAB to the directory that holds the function file. Otherwise, when we attempt to execute

```
v2 = my_square_function(v,b);
```

we get an error because MATLAB can't locate the file `my_square_function.m`. So, first navigate MATLAB to the directory where this file lives.

We have created a new function. This function takes two inputs, and returns one output. Having created the function and navigated MATLAB to the appropriate directory, we may finally execute this function:

```
v = (0:1:10);
b = 2.5;
v2 = my_square_function(v,b);
```

Notice that when the function is called, the variables `v` and `b` act as the input arguments. Within the function, variable `v` is assigned a different label (e.g., `input1`) and the variable `b` is assigned a different label (e.g., `input2`). However, this relabeling doesn't matter when we call the function; it happens behind the scenes, within the function. In the same way, within the function, the output variable is labeled `output`. But in our call to the function, we relabel the output `v2`.

Q: Examine the output variable `v2`. Does it match your expectations?

Q: Consider this call to the new function,

```
v2 = my_square_function(b,v);
```

We have swapped the order of the inputs. Does the function still perform the desired computation? If not, how could we fix the function? *Hint:* This question is rather advanced.

1.22 Saving Your Work

The file `my_square_function.m` defines a MATLAB function that we can execute. In addition, we can also create MATLAB files called scripts. Scripts are useful for executing a series of commands and saving the work when calling functions. To create a new MATLAB script, enter at the MATLAB command prompt

```
edit
```

This creates a blank file named `Untitled` in the MATLAB editor, into which we may enter any commands we like. For example, type into this blank file the following commands:

```
x = (0:0.1:10);           %First, define the vector x,
figure                    %... and open a figure.
for k=1:1:5               %For k from 1 to 5 in steps of 1,
    y = sin(x + k*pi/4);  %... define y (note 'k' in sin),
    hold on               %... hold the figure,
    plot(x,y)             %... and plot x versus y,
    hold off              %... and release the figure.
end
```

It is then easy to execute the code in this script. You may do so by copying the code from the script and pasting it into the command line, or (more elegantly) by using the mouse to select Run on the MATLAB editor tab. This will execute all the commands within the script. You can then save this script (with the `.m` file extension) and return to it later. For example, you might redefine `x = (0:0.1:25);` and reevaluate your script. To do so requires only changing one line of the script (the first line) and reexecuting all the code in the script. By saving the list of commands as a script, you do not need to retype all the other commands when you make a change and reevaluate the results. Saving a list of commands in a script is convenient and less prone to error.

1.23 Loading Data

Sometimes we need to load data (collected in an experiment, say) into MATLAB. Fortunately, that's easy to do. For an example of this procedure, visit,

`http://github.com/Mark-Kramer/Case-Studies-Kramer-Eden`

and download the file `Ch1-example-data.mat`. We now would like to open this dataset. The first step is to direct MATLAB to the same directory that contains the downloaded file. Once in the correct directory, type

```
clear                       %First clear the workspace,
load Ch1-example-data.mat   %... then load the data.
```

The first command, `clear`, clears the MATLAB workspace. This command is not necessary, but we perform it here so that any new variables we subsequently load are obvious. The second line of code, `load Ch1-example-data.mat`, reads in the data stored in the file `Ch1-example-data.mat`.

Q: What variables now exist in your workspace? What is the size of variables `t1` and `v1`? Plot `v1` and `t1` to get a sense for how the two variables behave.

Note that the file extension `.mat` denotes a MATLAB save file. Typically, a `.mat` file is created in MATLAB (using the `save` command; see MATLAB Help) and then loaded into MATLAB with the `load` command. Other file types can also be loaded into MATLAB; see MATLAB Help.

1.24 Loading Additional Functionality

A powerful feature of MATLAB (or any programming language) allows us to use sophisticated functions developed by others. Throughout this book, we make use of the open source software package *Chronux*, developed for the analysis of neural data [2]. The Chronux software is a MATLAB library, and to acquire its functionality we must download and install it. To do so, visit the Chronux website,

`http://chronux.org/`

On this website, you will find detailed information about the Chronux software package and its functionality. By navigating through this website, you will find the latest version of the Chronux software for MATLAB. Download and install this software following the setup instructions included in the downloaded software manual. The software consists of a directory with many MATLAB m-files that contain (sophisticated) functions.

To utilize these files, you must add the Chronux directory and all its subdirectories to the MATLAB path. To do so, check out the functions `genpath` and `addpath` in MATLAB Help. Once you have successfully added the Chronux directory and all subdirectories to the path, then Chronux should be accessible in MATLAB.

Q: To check that Chronux has been successfully installed, type

```
help chronux
```

This should display on the MATLAB command line detailed information about Chronux. If it does, you have successfully installed Chronux.

We perform many interesting tasks with Chronux in subsequent chapters.