# An Electrophysiologist's Introduction to Matlab

## — with Examples and Exercises —

**Daniel A. Wagenaar**

**First presented at: Neural Systems and Behavior Course, Marine Biological Laboratory, Woods Hole, MA, June 2008. Revised annually for the 2009–2011 courses.**

# Introduction

Matlab is an interpreted[1] computer language intended for data analysis. At first, you don't really have to know that it is a full-fledged computer language, because it can be used like a basic calculator that can also generate graphs. Over time, you will appreciate how well Matlab handles large data sets, and how easy it is to add your own data analysis functionality to what comes standard. An introduction to Matlab could easily fill a semester-long course at a university, because the capabilities of the system are very wide-ranging. We don't have that much time, so we'll just cover the very basics, and then proceed to some applications you will need for analysis of electrophysiological data. The style of this introduction is rather concise; it is my hope that this will allow you to quickly learn the essentials without getting bogged down in details. Several more extensive introductory texts are available on the internet (as well as in bookstores). The following are still available as of May 2011:

- "Numerical Computing with MATLAB": A several-hundred-pages-long text book by Cleve Moler, the president of Mathworks (the company which produces Matlab):

  http://www.mathworks.com/moler/chapters.html.

- "An Introduction to Matlab": A shorter introduction by David F. Griffiths of the University of Dundee; similar in style to the text you are reading, but quite a bit longer:

  http://www.maths.dundee.ac.uk/∼ftp/na-reports/MatlabNotes.pdf.

- "Introduction to Matlab": Another fairly short introduction, by Anthony J. O'Connor of Griffith University, Brisbane, Australia:

  http://maxwell.me.gu.edu.au/spl/matlab-page/matlab_tony.pdf.

You may also look into the book "Matlab for Neuroscientists," by Wallisch et al., Academic Press, 2008.

As you work your way through this tutorial, I strongly encourage you to experiment with each of the examples: try variations on them; find out what works and what doesn't. You really can't damage anything by typing nonsense at a computer (unless you try hard: the "**`delete`**" command, for instance, is predictably dangerous). If you get stuck, try typing "**`help`**". To learn more about Matlab's online help system, type "**`help help`**". You get the picture. There is also an extensive "help browser" built into Matlab; this you can access from the top menu bar.

In the following, text you type at the keyboard is typeset **`like this`**; answers given by Matlab are typeset `like this`.

[1] "Interpreted" in this context means that you can type commands and get immediate responses. This is in opposition to "compiled" languages, which execute faster, but make program development much harder.

# Chapter 1

# Introduction to the basics of the language

This chapter teaches you how to do use Matlab to perform simple calculations and to draw graphs. Not much in this chapter is specifically geared to handling electrophysiological data, but even if you are eager to start analyzing your recently acquired data, working your way through the examples and exercises in this chapter will hopefully pay off: They will provide you with a working knowledge of Matlab, which will not only greatly facilitate understanding the later chapters, but will also allow you to tailor the examples presented there to your specific needs.

## 1.1 Basic mathematics

### 1.1.1 Arithmetic

Addition, subtraction, multiplacation, division, and exponentation are writen as you would expect: "**+**", "**-**", "*****", "**/**", "**^**". Multiplication and division take precedence over addition and subtraction, and exponentiation takes precedence over all. Parentheses can be used to enforce interpretation. Large and small numbers can be written in exponential notation, e.g., "**5e-3**" means 0.005.

**Example 1**

```
>> 1+2*5

ans =

    11

>> (1+2)*5

ans =

    15
```

```
>> 7*3^2

ans =

    63

>> 5e-2/7.0e3

ans =

    7.1429e-06
```

You see that Matlab announces that it is ready for your input by showing its "`>>`" prompt, and that it prefixes its answers by the text "`ans =`". (If you are using the student version of Matlab, the prompt is "`EDU>>`" instead of plain "`>>`".)

**Exercise 1**

How would you ask Matlab to compute

$$\frac{4 \times 2}{3 + 5} \quad ?$$

### 1.1.2 Mathematical functions

Matlab defines a long list of mathematical functions (plus you can define your own; see below).

**Example 2**

```
>> cos(60*pi/180)

ans =

    0.5000
```

```
>> log(10)

ans =

    2.3026
```

**Exercise 2**

Can you make Matlab confirm that $\sqrt{2}$ is about 1.41? *Hint:* There is a function called "**sqrt**". Find out about it by typing "**help sqrt**".

### 1.1.3 Logic and comparisons

Comparison between numbers in Matlab is expressed using "**==**", "**>**", "**<**", "**>=**", and "**<=**". Note that equality is tested using two equals signs. A single equals sign expresses variable assignment; see below. Matlab represents truth as "**1**" and falsehood as "**0**". Logical "and" is written as "**&**"; logical "or" as "**|**".

**Example 3**

```
>> 2 > 1

ans =

    1

>> 3 == 7

ans =

    0
```

```
>> 5 > 3 & 5 < 8

ans =

    1

>> 5 = 3

??? 5=3
      |
Error: Missing operator, comma,
or semicolon.
```

**Exercise 3**

Which is bigger, $3^7$ or $7^4$? *Hint:* You could just ask Matlab to compute the two numbers separately, and do the comparison yourself, but that's not the point of the excercise.

## 1.2 Creating and using variables

Names of variables in Matlab can have any length, and may contain letters, numbers, and underscores. They must start with a letter. Assignment is written as "**=**".

**Example 4**

```
>> x = 5

x =

    5
```

```
>> y = 3 * x^2

y =

    75
```

**Exercise 4**

What happens if you redefine a variable? Do other variables automatically change their values to match (as in *Neuron*), or do they remain unaffected (as in *Java* or *C*)?

If you don't want to see intermediate results, simply place a semicolon at the end of a line. To show the value of a variable, simply type its name.

**Example 5**

```
>> spd = 8;
>> dur = 7;
>> dist = spd*dur;
```

```
>> dist

dist =

    56
```

**Exercise 5**

Can you think of a reason why this is useful? (*Hint:* Try example 13 below without semicolons.)

### 1.2.1 Structured variables

It is possible to store variables inside variables. This can greatly help to preserve clarity when using a large number of variables.

**Example 6**

```
>> x.a = 3;
>> x.b = 7;
>> y = x
y =

    a: 3
    b: 7
```

**Exercise 6**

Can you apply this technique recursively? In other words, what happens if you try to store a structured variable inside another variable? (Try it by typing "`rr.xx = x`" following the previous example.)

## 1.3 Using vectors and matrices

Matlab derives its name from the fact that it is very good at handling matrices. Variables can represent matrices just as easily as they can represent simple numbers (sometimes called "scalars" when contrasting them against vectors or matrices). Arithmetic and mathematical operations can be performed just as easily on matrices as on numbers.

### 1.3.1 Defining matrices

Matrices are defined by putting a series of numbers (or variables) in square brackets. Rows of the matrix are separated by semicolons.

**Example 7**

```
>> x = [ 1 2 3 ]

x =

    1 2 3

>> y = [ 4; 5; 6]

y =

    4
    5
    6

>> xxx = [x; 7 8 9]

xxx =

    1 2 3
    7 8 9
```

As the example suggests, more often than true matrices (which are two-dimensional arrays of numbers), we will use one-dimensional arrays, or *vectors*. Matlab treats them all the same; a vector of length $N$ is just a matrix of size 1x$N$ (as in the case of $x$ in the example) or $N$x1 (as in the case of $y$).

**Exercise 7**

In example 7, we concatenated the vector "`[7 8 9]`" to the vector "`x`" to generate "`xxx`". Can you concatenate "`[7 8 9]`" to the vector "`y`" from the example? Why (not)? How about concatenating "`[7; 8; 9]`" to "`y`"?

### 1.3.2 Arithmetic on matrices

Matrices may be added to each other or multiplied by scalars using the same operators used for simple numbers. This only works for matrices that have the same size.

**Example 8** (Continued from previous example.)

```
>> z = 2 * x

z =

    2 4 6

>> w = x + z + 1

w =

    4 7 10
```

**Exercise 8**

Can you apply mathematical functions to matrices? Create a matrix that contains the numbers 0, $\pi/6$, $\pi/4$, $\pi/3$ and $\pi/2$ (which in radians correspond to 0°, 30°, 45°, 60°, and 90°), and calculate their sines and cosines. Does the answer make sense?

Confusingly, to multiply matrices element-by-element, you cannot use the operator "`*`", since that represents matrix multiplication. Instead, you must write "`.*`". Similarly, "`/`" represents multiplication by the matrix inverse, and "`^`" represents matrix exponentiation (which you almost certainly will never use). Instead, write "`./`" and "`.^`". If you accidentally write "`*`" (or "`/`", etc.) where "`.*`" (or "`./`", etc.) would be appropriate, you may get unexpected results or error messages. Fortunately, it is never an error to write "`.*`", even when multiplying simple numbers.

**Example 9**

```
>> [4 6 15] ./ [2 3 5]

ans =

   2 2 3

>> [1 2] * [1; 2]

ans =

   5
>> [1; 2] * [1 2]

ans =

   1 2
   2 4

>> [1 2] * [ 1 2]
??? Error using ==> *
Inner matrix dimensions must
agree.

>> 5 .* 7

ans =

   35
```

**Note**: If you do not know what "matrix multiplication" means, don't worry; you won't be needing it in this course. A simple working strategy is to write "`.*`" and "`./`" instead of "`*`" and "`/`" unless you have a specific reason not too.

**Exercise 9**

Construct a matrix named "`x2`" the elements of which are the squares of the elements of the matrix "`xxx`" in Example 7.

### 1.3.3 Accessing the contents of matrices

The examples above show how Matlab can perform arithmetic on all the elements of a matrix at once. What if you want to work with specific elements? The examples below show how to do that.

**Example 10**

```
>> x = [2 4 6 8 10];
>> x(4)

ans =

   8

>> k = 3;
>> x(k) = 0

y =

   2 4 0 8 10

>> y = [2 4; 6 8]

y =

   2 4
   6 8

>> y(2,1)

ans =

   6
```

**Exercise 10**

What happens if you try to access an element that does not exist, such as "`x(7)`" in the previous example? What if you accidentially treat a matrix as a vector or vice versa, e.g., by asking for "`y(2)`" or "`x(2,2)`" in the previous example? Do you understand how Matlab treats those illformed inputs?

### 1.3.4 Matrices inside structured variables

The ability to hold multiple variables inside another variable is especially useful with matrices. For instance, you might like the variable "**spk**" to represent a series of action potentials, and you want to have it store the times of the action potentials as well as their heights and the electrode numbers on which they were recorded.

**Example 11**

```
>> spk.tms = [1.5 2.3 7.8];
>> spk.hei = [13 17 14.3];
>> spk.elc = [6 4 4];
```

```
>> spk

spk =

    tms: [1.5000 2.3000 7.8000]
    hei: [13 17 14.3000]
    elc: [6 4 4]
```

**Exercise 11**

Do all matrices inside a structured need to have the same shape? Can you include a scalar in the structure "**spk**" of the previous example? A line of text? *Hint:* Matlab wants text ("string literals") in single quotes, like "**'this'**". Why might that be useful?

### 1.3.5 Locating elements that satisfy some criterion

Continuing the previous example, let's try to isolate those spikes that were recorded on electrode number four, and find their times.

**Example 12** (Continued from previous)

```
>> idx = find(spk.elc==4)

idx =

    2 3
```

```
>> spk.tms(idx)

ans =

    2.3000 7.8000
```

The function "**find**" returns the indices of the elements that satisfy the condition; these may then be used to access the contents of related variables.

**Exercise 12**

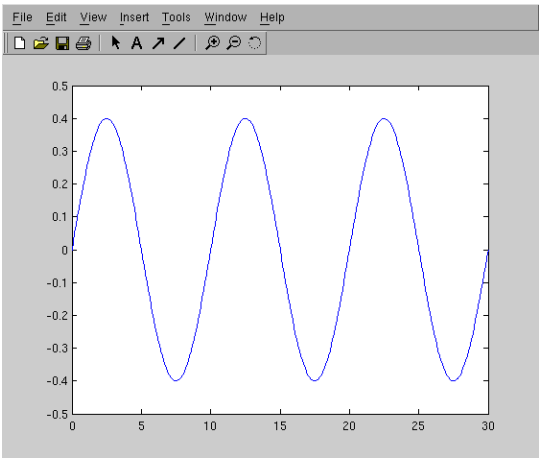
Can you find the heights of all spikes that happened within the first 5 seconds of the recording? (Assume that "**spk.tms**" is measured in seconds.)

## 1.4 Plotting results

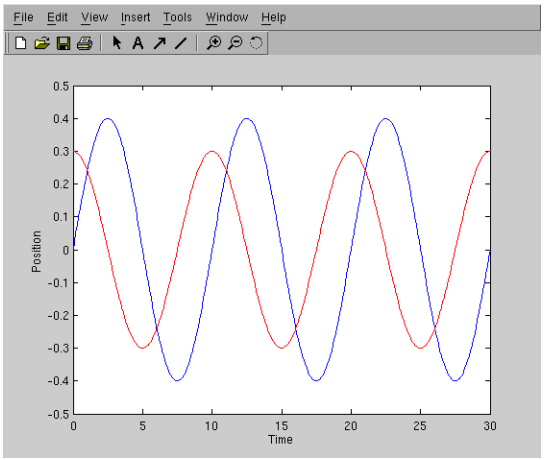
One of the things that make Matlab such a powerful tool is that you can quickly create graphical representations of your data. Here are a few simple examples; later, we will see how plotting intermediate results can enormously facilitate data analysis.

**Example 13**

```
>> tt = [0:0.1:30];
>> yys = .4*sin(2*pi*tt/10);
>> plot(tt, yys);
```




```
>> yyc = .3*cos(2*pi*tt/10);
>> plot(tt, yys, 'b', ...
        tt, yyc, 'r');
>> xlabel 'Time'
>> ylabel 'Position'
>> title 'Two waves'
```




There are several things to note about this example:

- The syntax "`[a1:da:a2]`" generates a vector that hold a range of numbers; in this case, the range 0, 0.1, 0.2, 0.3, . . . , 30.
- Just as you can do arithmetic on vectors, you can pass vectors to mathematical functions, which are then applied element-by-element. (You already knew that, if you tried exercise 8.)
- The function "`plot`" generates the actual graphics. You can generate a plot of $y$ vs $x$ by writing "`plot(x,y)`". Even simpler, you can plot $y$ vs the vector [1, 2, 3, . . . , length($y$)] by writing "`plot(y)`". In the second part of the example, two curves are plotted at once, and the text in single quotes specifies colors for each curve ("`'b'`" for blue, "`'r'`" for red).
- If a statement does not fit comfortably on a line, you can split it by typing "`...`" at the end of the first bit.
- There are many ways to adorn a plot; "`xlabel`", "`ylabel`", and "`title`" are among the most useful.

**Exercise 13**

Create a plot of your favorite mathematical function.

## 1.5 Saving your work (and loading it again)

To save all your variables to a file, simply write "`save myfile`". To load a previously saved file, write "`load myfile`". To see what variables you have loaded, type "`whos`". Often, you'll want to save only a few selected variables to a file; for this, see the example below. The command "`clear`" cleans Matlab's slate: the value of all variables are instantly and irretrievably forgotten. Except, of course, if they have been saved.

**Example 14**

```
>> x=3;
>> y=[7 4];
>> z=x+y;
>> zzz=z.^2;
>> whos
  Name Size Bytes Class

  x    1x1    8 double array
  y    1x2   16 double array
  z    1x2   16 double array
  zzz  1x2   16 double array

  Grand total is 7 elements
using 56 bytes

>> save sleepy.mat z zzz
```

```
>> clear
>> whos
>> load sleepy.mat
>> whos
  Name Size Bytes Class

  z    1x2   16 double array
  zzz  1x2   16 double array

  Grand total is 4 elements
using 32 bytes

>> zzz
zzz =

   100 49
```

**Exercise 14**

What happens if you load two files, each of which contains a variable with the name "**x**"?

## 1.6 Creating your own functions

If you find yourself applying the same sequence of commands to different sets of data over and over again, it is time to store those commands in a user-defined function. That way, you can execute the whole sequence with the same ease as calling any of Matlab's builtin functions. In Matlab, each user-defined function must be stored in its own file, the name of which corresponds to the name of the function. For instance, let's create a function called "triple," which multiplies numbers by three. The first step is to create a new text file named "triple.m." You can do this from the Finder (Mac), the Explorer (Windows), or the command line (Linux), but it is most easily done from within Matlab:

```
>> edit triple
```

This will open a new window in which you can edit the new function. Here are the contents for the new file:

```
function y = triple(x)
% TRIPLE - Multiply numbers by three
%   y = TRIPLE(x) sets Y to 3 times X.
%   This works for scalar as well as matrix-valued X.

y = 3*x;
```

Note that lines beginning with "**%**" are *comments*, i.e., explanatory text that Matlab will ignore. By convention, the first comment line in a function gives a brief summary of what the function does, and the following several lines give more details. Typing such comments is well worth your time, because otherwise you will soon not remember what any of your beautifully crafted functions do. If you take the time to write comments, you can always ask Matlab for a reminder:

```
>> help triple

  TRIPLE - Multiply numbers by three
     y = TRIPLE(x) sets Y to 3 times X.
     This works for scalar as well as matrix-valued X.
```

Using your new function is easy:

**Example 15**

```
>> triple(5)

ans =

    15
```

```
>> a=[3 7];
>> b=triple(a)+1

b =

    10 22
```

**Exercise 15**

Does "**triple**" work on a matrix (such as "**[1 2; 3 4]**")? Does it work on a structured variable (such as "**spk**" from example 11) ?

Let's consider a slightly more involved example. (Type this into a new empty file and save it as "fibo.m".)

```
function ff = fibo(n)
% FIBO - Generate Fibonacci numbers
%   ff = FIBO(n) returns a vector containing the first N
%   Fibonacci numbers, that is, the first part of the
%   sequence (1, 1, 2, 3, 5, 8, ...).

ff = zeros(n, 1);
ff(1) = 1;
if n>1
  ff(2) = 1;
end
for k=3:n
  ff(k) = ff(k-2) + ff(k-1);
end
```

This example introduces several new ideas:

- The function "**zeros**" initializes a matrix to a given size. This is not strictly necessary, but it will make your code run more efficiently and—in my opinion—easier to read.
- The construct "**if *cond* ... end**" makes the code inside execute only if the condition *cond* is satisfied (i.e., if $n > 1$ in the example).
- The construct "**for *var* = *range* ... end**" makes the code inside execute repeatedly; in the example, once for each value of $k$ from 3 up to (and including) $n$. (If $n < 3$, the code will not execute at all.)

**Example 16**

```
>> fibo(10)

ans =

    1 1 2 3 5 8 13 21 34 55
```

**Exercise 16**

Create a function named "**even**" that returns 2 if its argument (i.e., the number fed into the function) is even, and 1 if its argument is odd. *Hint:* You could use either the builtin function "**mod**", or the builtin function "**floor**" as part of your implementation. Type "**help mod**" and "**help floor**" to learn more.

If you're ambitious, you can try to extend this function to work nicely on matrices as well as on single numbers, so that for instance "**even([5 8 13 21 34])**" would yield "`1 2 1 1 2`".

## 1.7 Conclusion

Here ends the lightning-fast overview of the main features of Matlab. Nobody would expect you to be proficient at writing your own functions for tensor algebra in curved spaces at this point (and that's OK, because we won't be doing theoretical physics here). However, I hope you have gained an appreciation for the versatility of Matlab as a tool for analysis, and, most importantly, are starting to be unafraid of it: despite the fact that Matlab is a full-blown computer language, I hope to have shown that it is easy to use for simple things.

The next chapters will introduce some specific techniques useful for analysis of electrophysiological data, in particular spike detection and spike train analysis. For a more in-depth general introduction, I recommend trying the texts mentioned earlier.

# Chapter 2

# Spike detection and spike sorting

Spike detection (the process of identifying action potentials in extracellular traces) and spike sorting (the process of assigning those action potentials to individual cells) are topics of a substantial scientific literature, and are generally outside of the scope of this course. We will, however, need to analyze our recordings, so this chapter introduces a basic spike detector and a manual spike sorter which I have found useful for leech recordings.

## 2.1 Converting traces to spike trains

Spike detection is essentially the process of converting traces recorded from an extracellular electrode to a series of spike times. The first step is to load the electrophysiology data into Matlab. For this, I wrote the function "**loadephys**". You can use it to load your own data, but for now, let's load some data I recorded earlier.[1] This is from a ganglion 3–8 preparation bathed in conopressin, with suction electrodes bilaterally on the DP1 nervers of ganglia 5 and 6.

```
>> [vlt,tms] = loadephys('conoextra.daq');
```

(Don't forget the semicolon, or you'll see a very long stream of numbers scroll by.) You don't need to know how "**loadephys**" works internally; just that it loads electrophysiology data, and returns the recorded voltages as well as the times at which they were recorded.

Let's take a look at what we loaded:

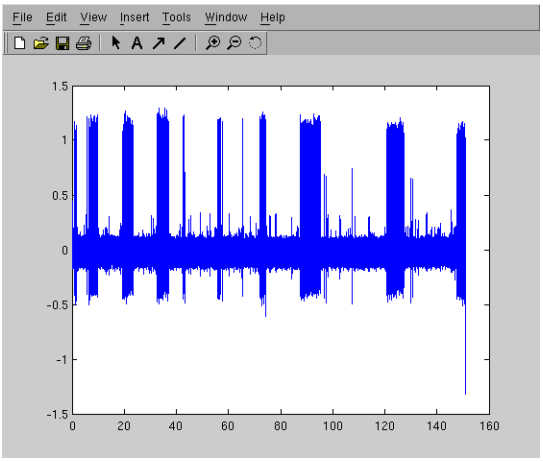
```
>> whos
  Name    Size          Bytes     Class

  tms     1510000x1     12080000  double array
  vlt     1510000x8     96640000  double array
```

As you can see, the variable "**vlt**" contains data from 8 electrodes. Evidently, the records are rather long (1,510,000 samples). The variable "**tms**" contains the time stamps at which each sample was recorded, measured in seconds. These times are the same for each of the 8 electrodes, so "**tms**" is just a vector, not an Nx8 array. If you type "**tms(1)**" and "**tms(end)**", you will see that time starts at $t = 0$ s and ends at $t = 151$ s. If you type "**tms(2)-tms(1)**", you will find that the interval between successive samples is 0.0001 s, i.e., the sampling frequency was 10 kHz.

[1] These data, as well as some other data files used later in this chapter, are available in a zip-file called "WagenaarMBL-data.zip", which you may download from http://www.danielwagenaar.net/teach.html if it's not already on your computer.

The first thing to do now is probably to plot one of the traces, just to get a better sense of the data. My notebook says that channel 8 corresponds to the right DP1 nerve of ganglion 6:

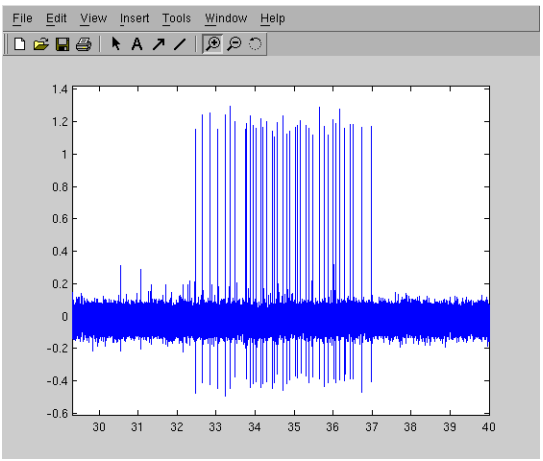
```
>> plot(tms, vlt(:,8));
```




If that took forever, you can opt to plot only every 10th data point, with only minimal loss of precision:

```
>> plot(tms(1:10:end), vlt(1:10:end,8));
```

On the x-axis of the graph is time, in seconds; on the y-axis is voltage, in volts. You can use the looking class button to enter zoom mode, then drag a rectangle over the graph to zoom in on a detail. That way, you can confirm that the thickets are actually bursts of individual spikes:




If you zoom back out (by double clicking on the graph), you will see that there are spikes of various sizes. From other experiments we know that the largest ones are from the contralateral dorsal exciter cell 3 (motor neuron DE-3) and that the intermediate ones are likely from the annulus erector. I don't know which cell produced the smallest spikes.

We ultimately want to extract the times of the DE-3 spikes. This recording is exceptionally clean, so it would be easy to just select spikes with amplitude greater than 1 millivolt. Often, however, the amplitudes of spikes from different neurons will not be quite as distinct, so it is useful to start by

first picking up every blip that could possibly be a spike, and select the relevant spikes in a second step.

The first step is accomplished using the function "**detectspike**":

```
>> spk = detectspike(vlt(:,8), tms);
```

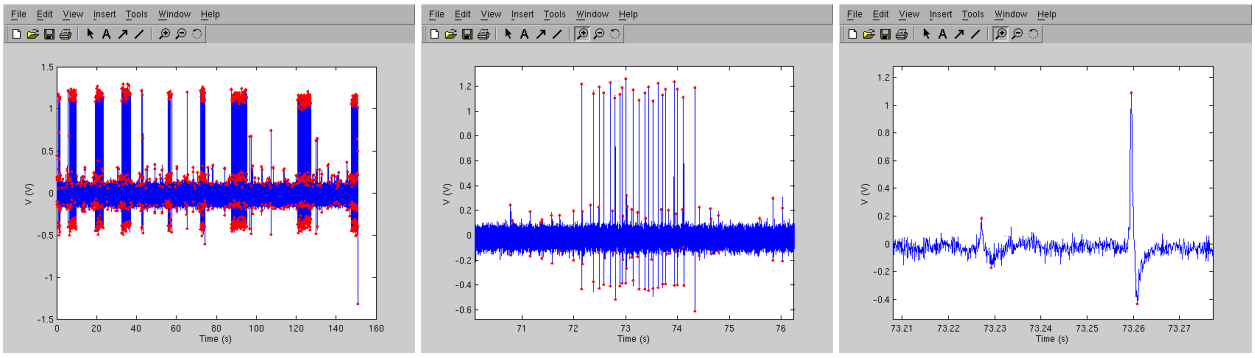
This returns a structured variable with information about each of the spikes:

```
>> spk

spk =

    tms: [1418x1 double]
    amp: [1418x1 double]
```

Let's see what got detected as spikes:

```
>> clf
>> plot(tms(1:end), vlt(1:end,8));
>> hold on
>> plot(spk.tms, spk.amp, 'r.');
>> xlabel 'Time (s)'
>> ylabel 'V (V)'
```

(The command "**clf**" clears the current graph; the command "**hold on**" instructs Matlab to add subsequent plot commands to the graph rather than to replace what was previously plotted.)




*(The second and third graphs are zooms into the first one.)*

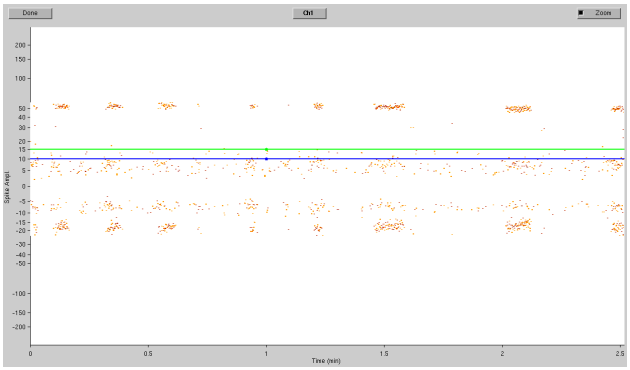
As you can tell, this detected far more "spikes" than are real action potentials from cell DE-3, and it often detected the negative as well as the positive phase from one action potential. No problem, our goal was to find everything that could possibly be an action potential. We'll sort the wheat from the chaff in the next step.

## 2.2 Selecting the spikes that truly are DE-3's action potentials

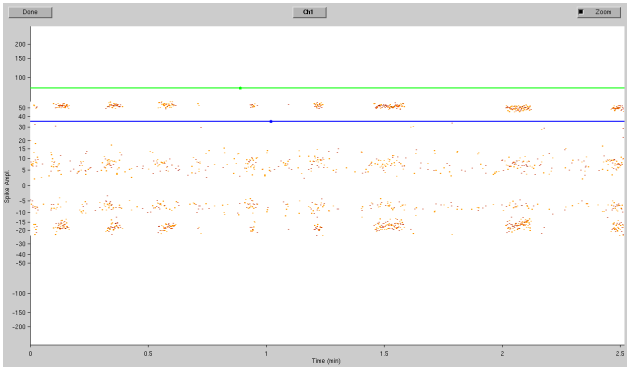
As I mentioned, automatic spike sorting is a topic of scientific research, and no current methods are entirely satisfactory. Therefore, we will use a semi-automated technique. If you type:

```
>> gdspk = selectspike(spk);
```

a new window will pop up showing the amplitudes of all the spikes in the recording and threshold lines. You can use the threshold lines to demarcate the spikes you care about. Notice the little "handles" on each line. If you drag those, the lines move. By grabbing a line not near a handle, you can make extra handles appear, so you can deal with situations where the amplitudes of spikes from a given cell change over time. To make a handle disappear, simply drag it past the next (or previous) handle. By dragging a rectangle over the graph (start away from the threshold lines), you can zoom in to see details.



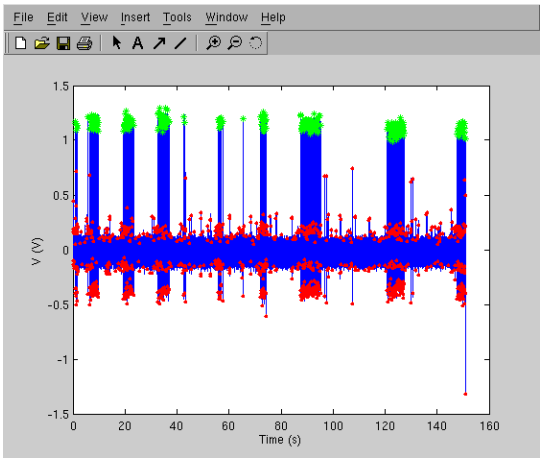

*The window created by "*`selectspike`*".*

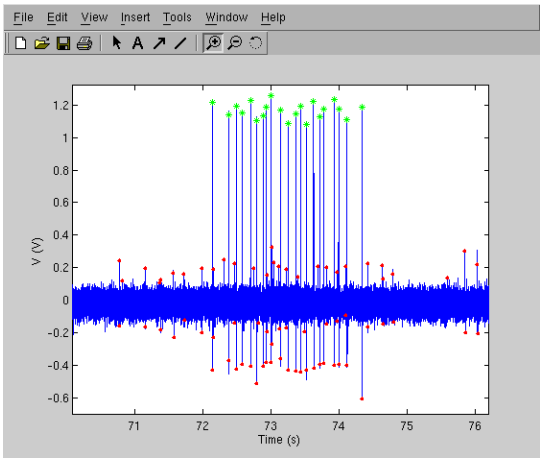



*After moving the threshold lines to appropriate positions.*

When you are satisfied, click "Done." The window will disappear, and the selected spikes will be stored in the new variable "`gdspk`". You can plot them on top of the previous graph to see if the selection criteria were good:

```
>> plot(gdspk.tms, gdspk.amp, 'g*');
```




As you can see, this successfully selected all the real DE-3 spikes, and no false positives.

This would be a good time to save your work:

```
>> save selectedspikes6R.mat gdspk
```

**Exercise 17**

Extract the spikes that belong to DE-3 on the DP-1 nerve on the left of ganglian 6. They are stored in electrode channel 6 of the same file. *Hint:* You will notice that these spikes have greater amplitude in the negative direction. That's just the result of how the electrode was wired. You may find it easier to select the negative-going peaks in "**selectspike**" instead of the positive-going ones. Your choice shouldn't significantly affect the interpretation of your data. The next chapter will teach you techniques to confirm that claim.

# Chapter 3

# Analyzing spike trains

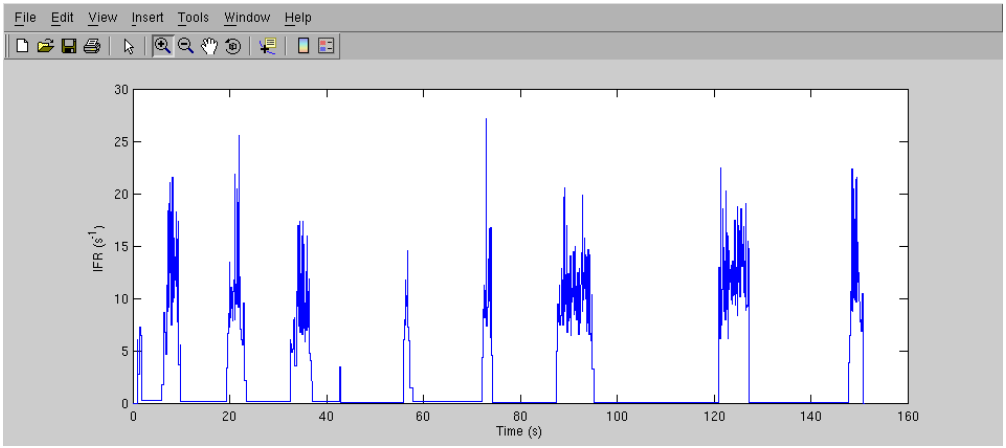
In the previous chapter, we learned how to extract the times of action potentials (also known as a spike train) from a nerve recording. However, extracting spike trains is just the beginning of analyzing a recording. In this chapter, we will consider a few useful characteristics that one can extract from a spike train.

## 3.1 Instantaneous firing rate

The *instantaneous firing rate* or *IFR* at some time point $t$ in a spike train is defined as the reciprocal of the interval between the two spikes surrounding $t$. (Thus, if a spike train consists of spikes at times 0.3, 0.7, 1.3, 2.1, and 3.8 s, the IFR at time $t = 0.6$ s is 2.5 $s^{-1}$. The IFR is not defined before $t = 0.3$ s or after $t = 3.8$ s.) Calculating instantaneous firing rates using builtin Matlab functions is a little involved, so I have created the function "**instantfr**", which does the hard work for you.

**Example 18**

```
>> load selectedspikes6R.mat
>> t_samp = 0.01; % Let's sample the IFR every 0.01 s.
>> tt = [0:t_samp:151]; % Recall that length of the original
>> % recording was 151 s.
>> ifr = instantfr(gdspk.tms, tt);
>> plot(tt, ifr);
>> xlabel 'Time (s)'
>> ylabel 'IFR (s^{-1})'
```



(This example assumes that you saved the results of spike sorting as suggested at the end of chapter 2.)

**Exercise 18**

What happens if you sample IFR at 0.1-s intervals or 1-s intervals? Do you still get a fair representation of the data?

## 3.2 Smoothing the firing rate

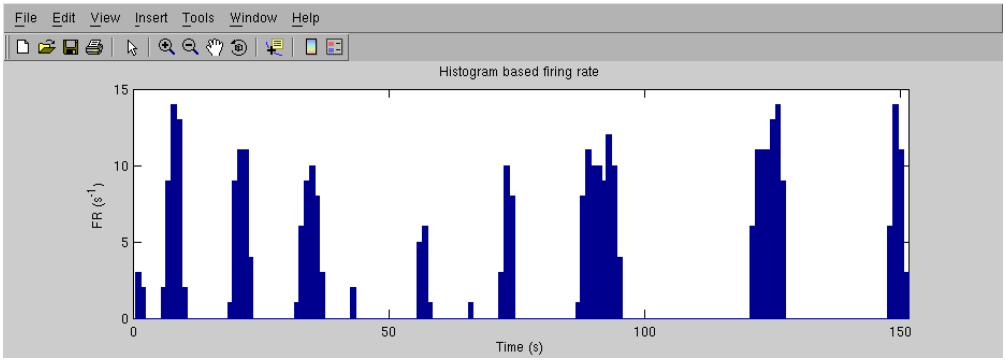
In some neural codes, such as in the auditory system of owls, the timing of individual spikes is of great importance. In many other cases—and to our knowledge in all leech neurons—only the rate of spikes in some time interval matters. One way to calculate this rate is to simply compute a histogram of spikes in time bins of some reasonable size, e.g., 1 s:

```
>> hist(gdspk.tms, [0:1:151]);
```

This plots the results; if you want to store the results in a variable instead, you'd write:

```
>> [fr, tt] = hist(gdspk.tms, [0:1:151]);
```

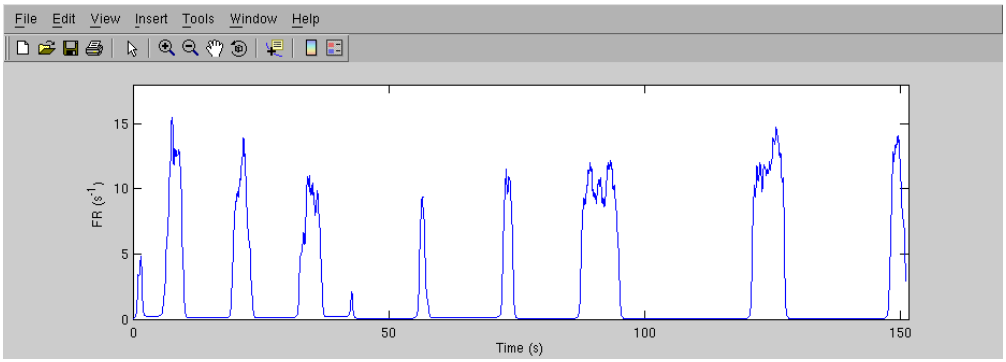
Then you could plot the results by typing "`plot(tt, fr)`" or—to get a bar graph just like the one produced by the "`hist`" function—by typing "`bar(tt, fr)`".




A visually more pleasing result which also doesn't suffer from the random effects of spikes falling just to the left or to the right of the edge between two bins can be obtained by using the previously computed instantaneous firing rate, and applying a low-pass filter.

**Example 19** (Continued from example 18.)

```
>> tau = 1; % Low-pass to τ = 1 s.
>> [b, a] = butterlow1(t_samp / tau); % Prepare the filter.
>> ffr = filtfilt(b, a, ifr); % Calculate smoothed firing rate.
>> plot(tt, ffr);
```




(Note for the curious: "**`butterlow1(f_0/f_s)`**" prepares a first-order low-pass Butterworth filter with cut-off frequency $f_0/f_s$, and "**`filtfilt`**" applies that filter twice, once in the forward direction, and once backwards. The net result is a second-order low-pass filter with zero phase shift.)

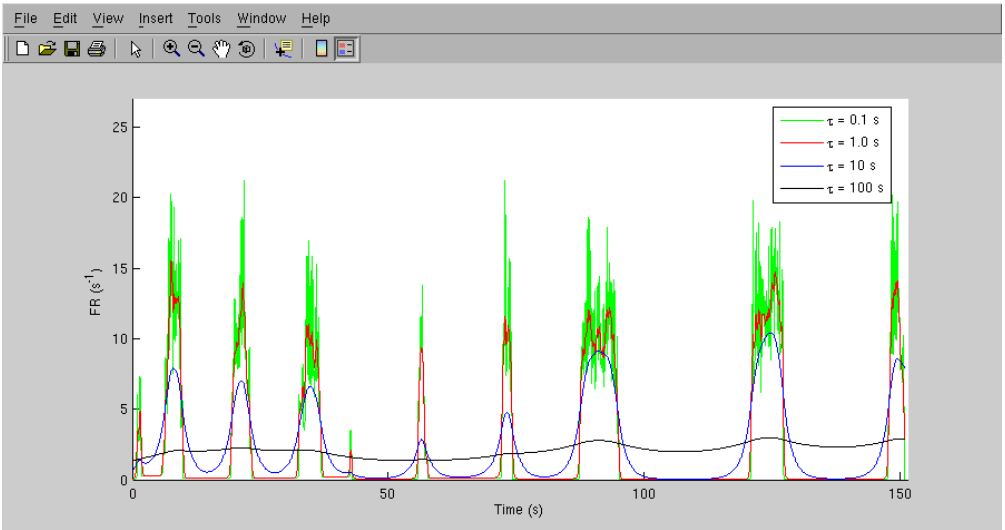
**Exercise 19**

> Create a figure that contains both the IFR from example 18 (in blue) and the smoothed firing rate from example 19 (in red). Zoom in on a burst. Do you understand the differences between the two curves?

Depending on your application, various different timescales can be used for the calculation of firing rates. In the example below, we use $\tau$ = 0.1 s, 1 s, 10 s, and 100 s.

**Example 20** (Continued from example 18.)

```
>> taus=[0.1 1.0 10 100];
>> clf; hold on
>> clr='grbk';
>> for k=1:4
>>   [b, a]=butterlow1(t_samp / taus(k));
>>   ffr = filtfilt(b, a, ifr);
>>   plot(tt, ffr, clr(k));
>> end
>> legend('\tau = 0.1 s', '\tau = 1.0 s', ...
      '\tau = 10 s', '\tau = 100 s');
```




**Exercise 20**

Can you add a line to the figure created in the above example that represents the average firing rate? In calculating the average, does it matter (much) whether you first apply a low pass filter? Why (not)?

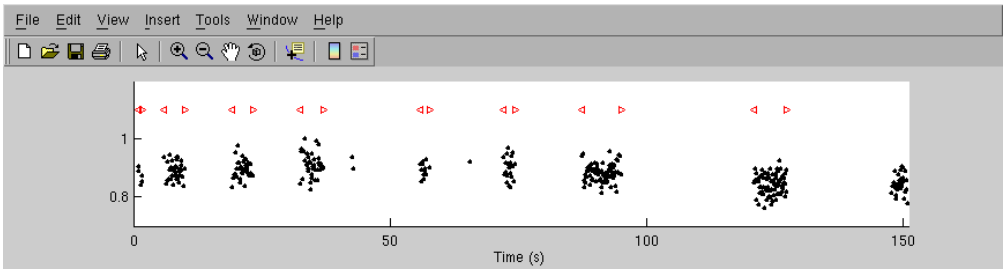
## 3.3 Detecting and characterizing bursts

As should be obvious from any of the preceding graphs, cell DE-3 does not fire tonically. Rather, it fires bursts of many action potentials separated by several seconds without much activity. Understanding these bursts could be key to understanding the neural code.

If you look back to the graph of the instantaneous firing rate produced by example 18, you notice that the IFR is never less than 1 $s^{-1}$ inside the bursts, and never more than that outside the bursts. Thus, we can detect bursts by looking for intervals of 1 s or more without spikes. I have created the function "**burstinfo**" to perform this detection and to extract some useful parameters about each burst. In the example below, just a few of these parameters are used; type "**help burstinfo**" to learn what else is available.

**Example 21**

```
>> load selectedspikes6R.mat
>> bi = burstinfo(gdspk.tms, 1.0, 151); % The last parameter ...
          % specifies the length of the recording.
>> clf; hold on
>> plot(gdspk.tms, gdspk.amp / max(gdspk.amp), 'k.');
>> plot(bi.t_start, 0*bi.t_start+1.1, 'r<');
>> plot(bi.t_end, 0*bi.t_end+1.1, 'r>');
>> axis([0 151 0.7 1.2])
>> xlabel 'Time (s)'
```




In the graph, the beginnings of detected bursts are marked by left-pointing triangles, and the ends by right-pointing triangles. Note that the final burst in the recording is not marked. This is because it was still in progress at the time the recording was ended ($t = 151$ s). Thus, reporting its end time as the time of the last recorded spike would be inaccurate, and "**`burstinfo`**" chooses to not report on it at all. If you do want to know about it, you can pretend that the recording went on forever by not specifying an end time: "**`bi = burstinfo(gdspk.tms, 1.0)`**". Note also that the pair of spikes around $t = 43$ s was not detected as a burst; this is because "**`burstinfo`**" only considers sequences of at least three spikes to be bursts.

**Exercise 21**

What is the average duration of bursts in the above example? What is the maximum interval between two consecutive bursts? Is the number of spikes in each burst correlated with the duration of those bursts? With the preceding interval? *Hint:* "**`burstinfo`**" returns all the parameters you need for this exercise. To investigate correlation, you can simply plot the relevant parameters against each other, or use the "**`corrcoef`**" function.

# Intermezzo

Congratulations! You have covered ground to analyze the data from today's lab exercise. The remaining chapters of this tutorial introduce several other techniques that you may find useful for analyzing electrophysiological data, but they are not critical for today's work. I suggest you read them at your leasure at a later time.

I hope the whirlwind pace of chapters two and three hasn't overwhelmed you. It is no problem at all if you have not understood all of the fine points. I just hope you will take home the lesson that Matlab is a very flexible tool, and that you are now not afraid to further explore it on your own. I have used Matlab for over a decade now, and I am still getting better at it, so please do not be discouraged if this tutorial felt a little (or very) overwhelming; we covered a lot of material in a very short time. Do feel free to come to me with any questions you may have, even if you're not sure how to phrase them.

On a final note, although this tutorial has spoken exclusively about Matlab, I do not want to leave you with the impression that Matlab is the only game in town. In fact, there are several other software packages that allow you to do this sort of analyis. I would like to mention two by name, because they are free software (unlike Matlab which is closed-source and expensive).

The first, "Octave", is almost a drop-in replacement for Matlab, except that it is free software. All of the examples from chapter one work just fine in Octave. The current version of Octave (http://www.gnu.org/software/octave) still misses a few functions so that "selectspike" doesn't run, but development appears to be rapid, so that may be fixed soon.

The second, "Pylab", is also similar to Matlab in intent, but quite different in implementation: it is a collection of extensions to the Python language. Pylab is very much worth exploring, not only if your institute doesn't subscribe to Matlab. A good starting point to learn about Pylab is http://matplotlib.sourceforge.net/. Most Linux distributions have precompiled versions available, and Windows downloads are available through the aforementioned SourceForge page. If you like python, you'll love Pylab. Otherwise, Octave may be easier to learn.

Good luck with analyzing your data!

Woods Hole, June 2011

# Chapter 4

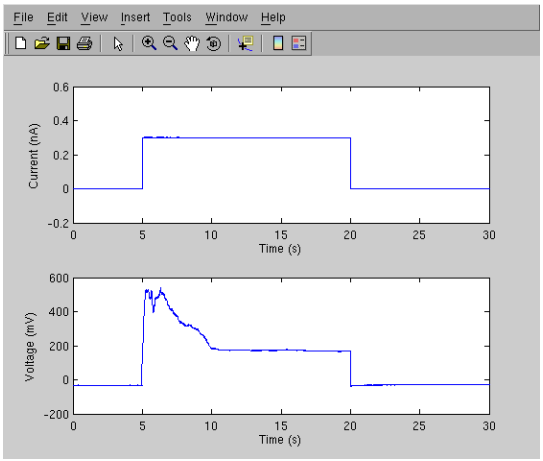
# Detecting intracellular spikes

In the previous two chapters, we have analyzed extracellularly recorded spikes. What if you have an intracellular recording? After the initial step of detection, analyzing intracellularly recorded spike trains is no different from extracellularly recorded ones: in either case, one works with a series of timestamps. However, the process of detecting intracellular spikes is quite different from the one for extracellular spikes, because intracellular spikes have a different shape than extracellular spikes, and intracellular recordings are susceptible to different sources of noise. In this chapter, we will discuss how to handle these.

For your reference, I have stored all of the following code in a script file called "intracel.m".

**Example 22**

```
>> load intracel.mat

>> figure(1); clf
>> subplot(2, 1, 1);
>> plot(tt, ii);
>> xlabel 'Time (s)'
>> ylabel 'Current (nA)'

>> subplot(2, 1, 2);
>> plot(tt, vv);
>> xlabel 'Time (s)'
>> ylabel 'Voltage (mV)'
```




**Exercise 22**

I surreptitiously introduced the function "**subplot**" in the above example. As you see, this splits a figure window so that it can contain multiple graphs. Type "**help subplot**" to learn more, and create a 2x2 layout of the raster plots and firing rate plots of the left and right DP1 nerve recordings in the data set we used in the previous two chapters. Save your code in a ".m" file for future reference.

The data in example 22 are from a recording by Krista Todd. The top trace shows injected current, the bottom trace recorded voltage from a DE-3 cell in a preparation where the ganglion was left attached to a portion of body wall so that muscle actions could be observed. Looking at the graphs, you will notice that the voltage trace is badly polluted by at least two artifacts: (1) bridge balance

was incomplete, resulting in a jump near the start and end of the stimulus; (2) during the beginning of the stimulus, the baseline voltage drifted considerably (due to motion artifacts). Please note that these problems should not be blamed on Krista; intracellular recording from this kind of preparation is quite a tour-de-force. Still, we need to remove them in order to reveal the action potentials.

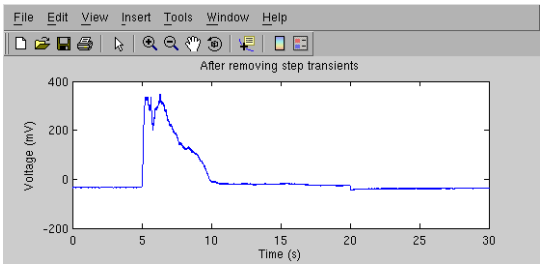
## 4.1 Removing the voltage steps caused by the bridge imbalance

Our first step will be to remove the big voltage steps at the beginning and end of the stimulus. To do this semi-automatically, the following Matlab code detects the transients in the current trace and subtracts the corresponding jumps from the voltage trace.

**Example 23**

```
>> L=length(vv);
>> DI=4;
>> yy=vv;
>> badidx=find([nan ...
      abs(diff(ii))]>.1);
>> for k=1:length(badidx)
     i0=max(badidx(k)-DI, 1);
     i1=min(badidx(k)+DI, L);
     v0 = yy(i0);
     v1 = yy(i1);
     yy(i1:end)=yy(i1:end) ...
      + v0-v1;
     yy(i0+1:i1-1)=v0;
   end
```

```
>> figure(2); clf
>> plot(tt, yy);
>> xlabel 'Time (s)'
>> ylabel 'Voltage (mV)'
```




Note for the curious: we measured the recorded voltage $DI = 4$ samples before and after the step, and subtracted the difference.

**Exercise 23**

Why did I write "`max(badidx(k)-DI, 1)`" in the example? Could I simply have written "`badidxk-DI`"? (What if one of the current steps was very close to the beginning of the recording?)

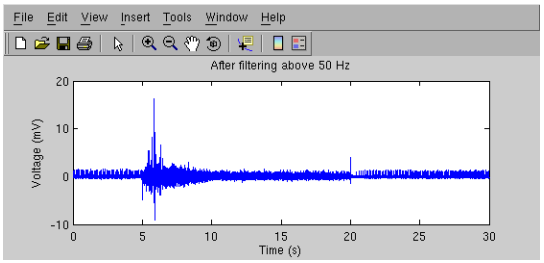
## 4.2 Removing the motion-induced drift

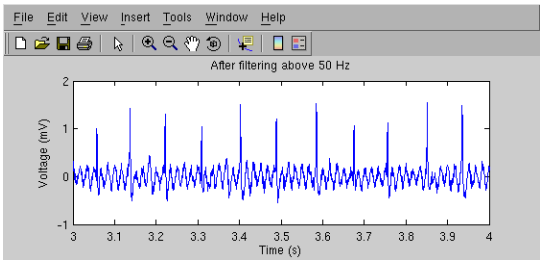
Looking at the graph in example 23, it is obvious that this procedure only solved part of the problems, so let's keep working. To remove the baseline drift induced by motion, we simply filter away any signal with frequencies below 50 Hz:

**Example 24**

```
>> dt_s = mean(diff(tt));
>> f0_hz = 1/dt_s;
>> fcut_hz = 50;
>> [b, a]=butterhigh1...
      (fcut_hz/f0_hz);
>> zz=filtfilt(b, a, yy);
```

```
>> figure(3); clf
>> plot(tt, zz);
>> xlabel 'Time (s)'
>> ylabel 'Voltage (mV)'
```




**Exercise 24**

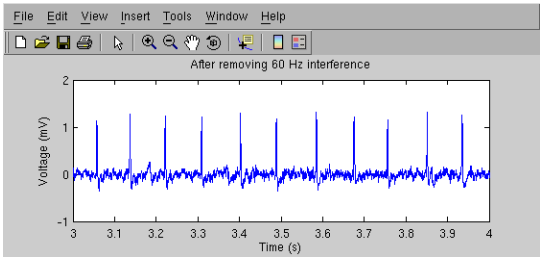
Experiment with different values of the cutoff frequency in the above example. Might 25 Hz or 100 Hz be better than 50 Hz? How would you judge?

## 4.3 Removing the 60 Hz pickup

The zoom on the right below example 24 shows that action potentials are now clearly distinguishable, but a fair bit of 60 Hz pickup, previously obscured by the larger artifacts, has also become evident. The most effective way I know to remove such pickup, is to use a template filter. The details of the method are beyond the present tutorial; if you are interested, feel free to look at the source of "**templatefilter**", or to ask me for an explanation.

**Example 25**

```
>> ww = templatefilter(zz, ...
      f0_hz/60, 200/f0_hz,
50);
>> figure(4); clf
>> plot(tt, ww);
>> xlabel 'Time (s)'
>> ylabel 'Voltage (mV)'
>> axis([3 4 -1 2])
```




**Exercise 25** (For die-hards)

Experiment with the parameters of "**templatefilter**".

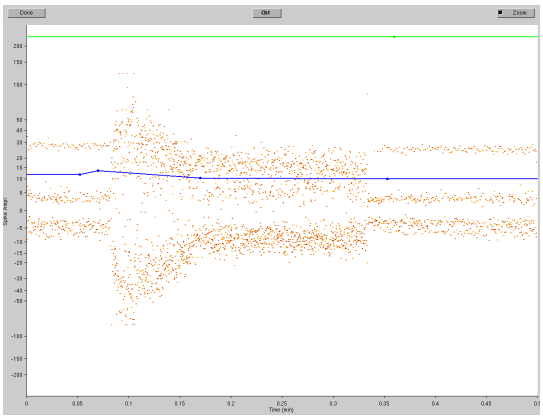
After removing all these artifacts, the trace looks reasonably clean; time to detect action potentials.

## 4.4 Detecting the actual action potentials

If you zoom around in the trace, you will notice that the amplitude of action potentials is quite variable, especially during the stimulus. In cases like this, it may take a bit of trial and error to find the best threshold. We can use the same basic technique as for extracellular spikes, but the firing rate in this example is so high that we must use a shorter window than before to estimate the RMS noise, and also a lower initial threshold:

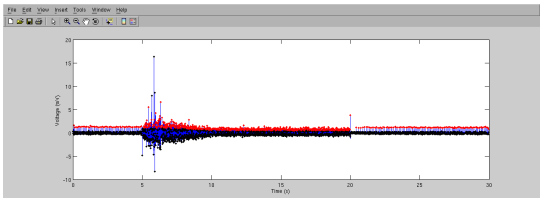
**Example 26**

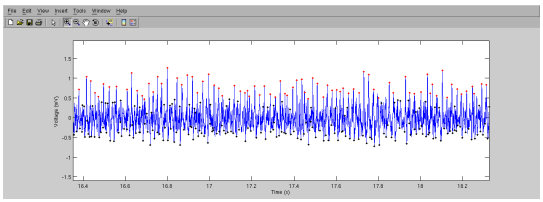
```
>> spk=detectspike(ww, tt, 2, 2);
>> gdspk=selectspike(spk);
```



*The "`selectspike`" window after dragging the threshold lines to my best estimate.*

```
>> figure(5); clf
>> plot(tt, ww);
>> hold on
>> plot(spk.tms, spk.amp, 'k.');
>> plot(gdspk.tms, gdspk.amp, 'r.');
>> xlabel 'Time (s)'
>> ylabel 'Voltage (mV)'
```





**Exercise 26**

Do you agree with where I drew the threshold line in the above example? Can you do better?

## 4.5 Conclusion

If you did the last exercise, you will have realized that, in cases like this where the signal-to-noise ratio is very low, setting a threshold will always be somewhat arbitrary. Still, this method probably yields a reasonably good estimate of the firing rate. Of course, if you are recording from a Retzius or sensory cell—with much larger action potentials—your signal-to-noise ratio will be a lot better, and your confidence in spike detection will be correspondingly greater.

# Chapter 5

# Analysis of multiple spike trains

If you have simultaneously recorded data from multiple intracellular or extracellular electrodes, you can of course use the techniques from the previous chapters to analyze the recordings separately. Frequently, however, much can be learned from the relationship between two recording channels. In this chapter, we will consider the example of a simultaneous intracellular and extracellular recording from the same neuron. In particular, we shall be concerned with the delays between recorded spikes. For the examples below, you may either use your own data, or the test data set provided.

## 5.1 Loading the raw data and extracting spike times

The obvious first step is to load the data into matlab:

```
>> [vlt, tms] = loadephys('intraextra.xml');
```

(These data were stored in a special file format defined by acquisition software created in my lab, but you don't need to worry about the details: "**loadephys**" knows how to load the relevant data.)

Let's take a look at what we loaded:

```
>> whos
  Name    Size          Bytes     Class

  tms     110000x1     880000  double array
  vlt     110000x6    5280000  double array
```

Unlike in the example in Chapter 2, there are only three channels of interest here: Intracellular voltage is stored in channel 2, intracellular current in channel 4, and extracellular voltage in channel 6. (Data from a second extracellular electrode is in channel 5, but those data are not of interest here.)

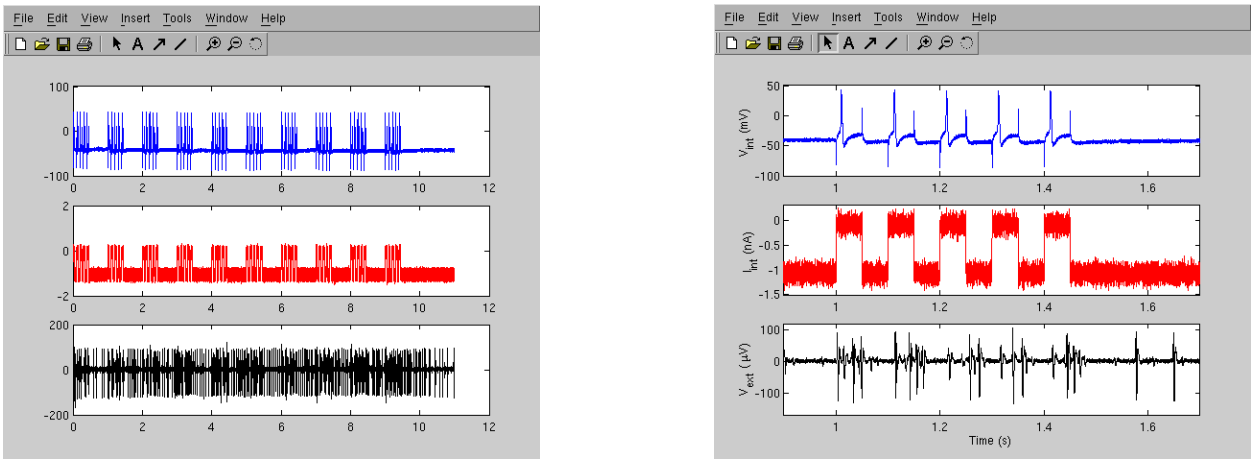
Scale factors have not been properly preserved (because of a bug in the software I used to record these data), so let's avoid confusion by giving better names to the various signals:

```
>> Vint_mV = vlt(:,2);
>> Iint_nA = vlt(:,4)/10;
>> Vext_uV = vlt(:,6)/10;
```

With that, let's start by looking at the various signals. We'll plot them in three "subplots," and use a clever Matlab trick to allow zooming into all three at the same time:

```
>> figure(1); clf
>> subplot(3, 1, 1); plot(tms, Vint_mV, 'b');
>> subplot(3, 1, 2); plot(tms, Iint_nA, 'r');
>> subplot(3, 1, 3); plot(tms, Vext_uV, 'k');
>> linkaxes(get(gcf, 'children'), 'x');
```

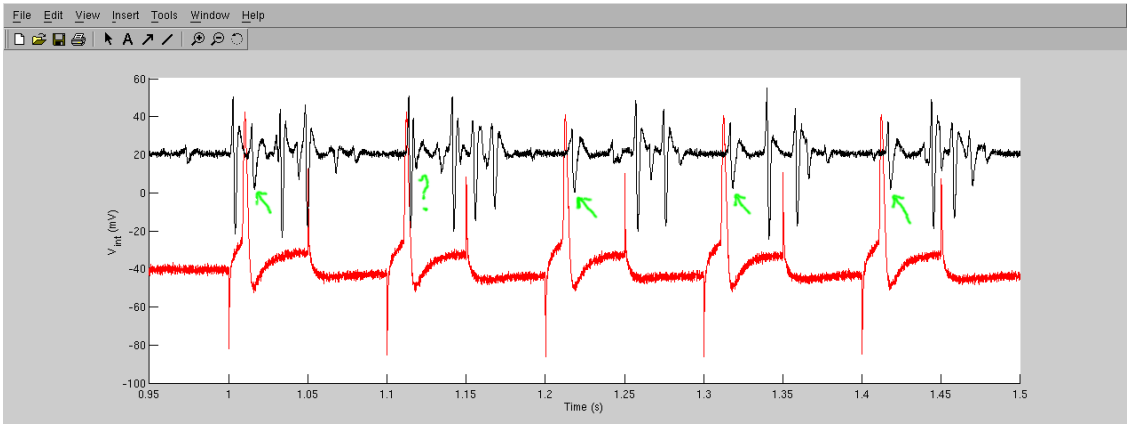
(If you are using an older version of Matlab at home, as I do, you won't have the "**linkaxes**" function available, so you might be better off plotting everything in one plot.)




*The results of the previous plot commands (left), and after labeling the axes and zooming in (right).*

In the top panel, the "rabbit ears" resulting from the current steps are clearly visible, as are the evoked action potentials. In the bottom panel, spikes from several different cells can be seen, and it is hard to tell which ones belong to the neuron with the intracellular electrode in it. Perhaps it might help to overlay the intracellular and extracellular traces?

```
>> figure(2); clf; hold on;
>> plot(tms, Vint_mV, 'r');
>> plot(tms, Vext_uV/3+20, 'k');
>> xlabel 'Time (s)'
>> ylabel 'V_int (mV)'
>> axis([.95 1.5 -100 60]);
```



(I hand-drew those green annotations.) Now, we can clearly see that extracellular spikes of a specific size tend to follow intracellular spikes.

**Exercise 27** (Not Matlab-related.)

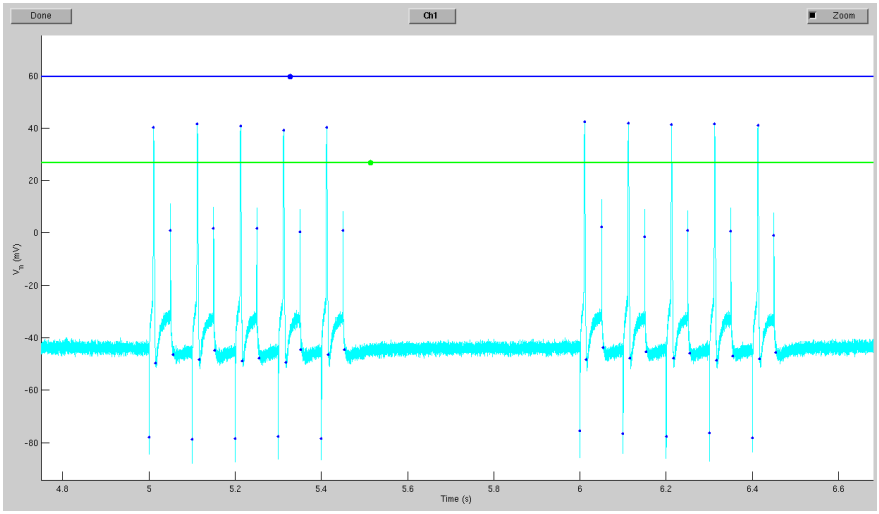
What do you think happened during the second stimulus?

*Hint:* What extracellular event occurred just before the expected spike?

We could now proceed to extract the extracellular spikes as in Chapter 2, and to extract the intracellular spikes as in Chapter 4, but picking out the right extracellular spikes might be difficult because of the presence of other spikes of similar amplitude. Let's therefore try something else.

Detecting spikes in very clean intracellular recordings like this one should be easy, and it is. I've written a convenient Matlab function that let's you pick them out:

```
>> intspk = selspktrace(Vint_mV, tms);
```




*The "`selspktrace`" window after zooming and dragging the limit lines to appropriate locations.*

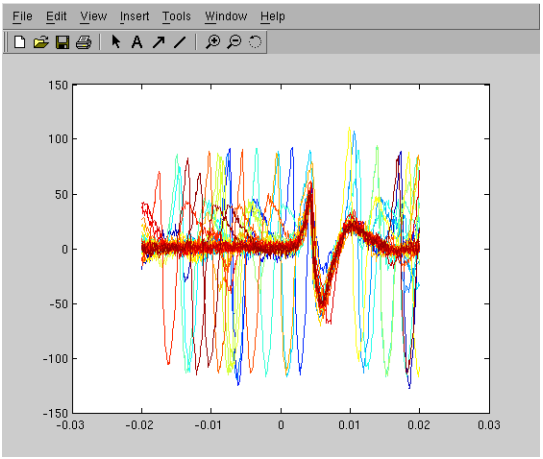
**Exercise 28** (For hardy souls.)

In this case, the rabbit ears are easily distinguished from the action potentials by amplitude. If the rabbit ears in your recording are taller, their widths should still be quite short, and thus distinctly not action potential–like. Write a Matlab function to exploit this characteristic and toss out the rabbit ears from the list of spikes in "`intspk`".

Now we know when each of the intracellular spikes happened. Thus, we could look at the extracellular trace right around these events. The most direct way to do that might be this:

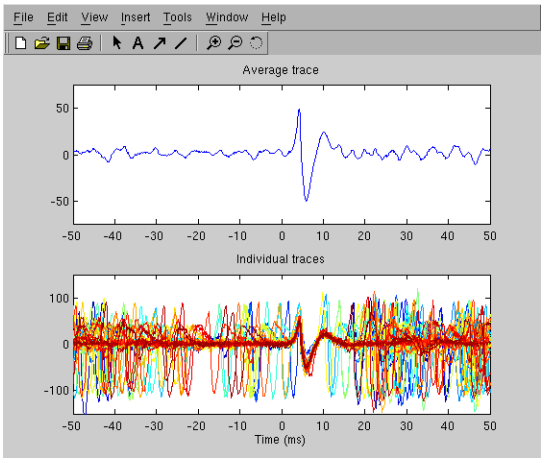
```
>> figure(1); clf; hold on
>> N = length(intspk.tms);
>> clr = jet(N);
>> for k=1:N
     idx=find(tms==intspk.tms(k));
     if idx>200 & idx<length(tms)-200
       plot(tms(idx-200:idx+200)-tms(idx), ...
         Vext_uV(idx-200:idx+200), 'color', clr(k,:));
     end
   end
```

The result looks like this:




To take the tedium out of that, I've created a a function that automatically extracts data around events. As a bonus, it also calculates the "event-triggered average": the average of the data collected around each of the events. You use it like this:

```
>> [avg, dt, indiv] = trigavg(Vext_uV, tms, intspk.tms);
>> subplot(2, 1, 1); plot(dt, avg);
>> subplot(2, 1, 2); plot(dt, indiv);
```




*(I've labeled the axes, recolored the traces, and zoomed in a little before taking that screenshot.)*

**Exercise 29**

Can you overlay the average intracellular trace on the top panel of the above graph?

Based on the output of "**trigavg**", calculating things like the average propagation delay is straightforward:

```
>> [mx, idx] = min(avg);
>> avgdelay = dt(idx)
avgdelay =

    5.9
```

**Exercise 30** (Not Matlab-related).

Why did I consider the time of the extracellular spike to be the time of its negative peak?

**Exercise 31**

How can you tell whether any intracellular events did not cause a corresponding extracellular spike? Could you encode your answer as a Matlab function that automatically finds out which action potentials failed to propagate?

One rather elaborate solution to that last exercise are the functions "**trigavgmax**" and "**trigavgfail**". Especially "**trigavgmax**" is more general and more ambitious in its scope than would be strictly necessary for this application, but you may be interested to read the source code. Regardless, they work like this:

```
>> [avg, dt, indiv] = trigavg(Vext_uV, tms, intspk.tms);
>> [mxx, dtt] = trigavgmax(-indiv, dt);
>> failures = find(trigavgfail(mxx));
```

# Chapter 6

# Some notes on further analysis

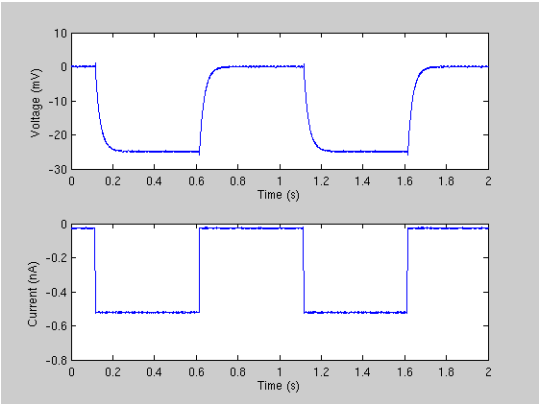
This chapter contains some notes on further analyzing your data. It is a work in progress, but hopefully already somewhat useful.

## 6.1 Measuring time constants

In this section, we will use Matlab to more accurately measure time constants than the standard method of eyeballing the delay until a signal has decayed to one third of its original size. While that method is often good enough, greater accuracy is called for if you are going to publish data. Moreover, the method described here, which consists of fitting an exponential curve to the data, provides an example of the more general problem of *function fitting*, which you will encounter in many other guises.

Let's start by loading a small data set:

```
>> [dat, tms] = loadephys('timecst.abf');'
>> figure(1); clf; subplot(2, 1, 1)
>> plot(tms, dat(:, 1);
>> subplot(2, 1, 2);
>> plot(tms, dat(:,3);
```



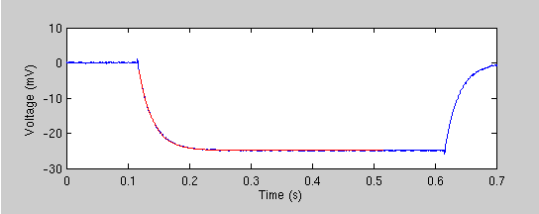

The bottom trace shows the current applied to an RC circuit; the top trace shows the resulting voltage. We are going to determine the time constant of each of the transitions. By zooming into the graphs, it is easy to determine that the transitions occur at 0.12 s, 0.62 s, 1.12 s and 1.62 s. Alternatively, you can use Matlab to find the transitions:

```
>> [iup, idn] = schmitt(dat(:,3), -0.2, -0.4, 1);
>> iup=iup(iup>1);
>> ttrans = tms(sort([iup; idn]))'

ttrans =

    [ 0.1160    0.6160    1.1160    1.6160 ]
```

(The function "**schmitt**" reports an upward transition at the beginning of the trace because the initial voltage is above its threshold; the second line above drops this transition from further consideration.)

Let's fit an exponential to the first transition:

```
>> t0 = ttrans(1)

t0 =

    0.1160

>> idx = find(tms>t0+0.002 & tms<t0+0.40);
>> [fit, yy] = physfit('expc', tms(idx), dat(idx,1));
>> subplot(2, 1, 1); hold on; plot(tms(idx), yy1, 'r');
>> fit.p

ans =

    1.0e+03 *
     3.5716   -0.0428   -0.0249
```




What do those numbers mean? Let's ask Matlab!

```
>> fit.form

ans =

    A*exp(B*x) + C
```

So, Matlab is telling us that this part of the trace is fit by

$$3571.6\exp(-42.8t) - 24.9.$$

In other words, $\tau = 1/42.8 = 0.0233$ s and the trace asymptotes to $-24.9$ mV. The red trace overlaid on the data is the fitted curve; evidently, the fit quality is pretty good.

**Exercise 32**

Based on those numbers, can you calculate the values of $R$ and $C$? Note that the current injected was:

```
>> mean(dat(idx, 3))
ans =
   -0.5204
```

i.e., 0.52 nA.

Let's step back for a moment, and consider what we just did. The main magic was the "**physfit**" call. This function is a general function fitter. It can fit straight lines, sinusoids, polynomials, exponentials, and several other predefined functional forms, as well as arbitrary user-defined functions. Beside returning the fit parameters (in "**fit.p**"), it also tells you the uncertainty in those parameters (in "**fit.s**"). If you tell it the uncertainty in your data, it will also report the quality of the fit. For instance, you could measure the noise inherent in your data:

```
>> noise = std(dat(tms<t0-0.05), 1)
noise =
   0.0783
```

and feed that to "**physfit**":

```
>> [fit, yy] = physfit('expc', tms(idx), dat(idx,1),noise);
>> fit(2).chi2
ans =
   1.0711
```

Since $\chi^2$ is close to one, we conclude that the fit is indeed very good. (If you call "**physfit**" in this way, "**fit(1)**" returns a fit of the data that disregards the uncertainties, and "**fit(2)**" returns a fit that uses the uncertainties.)

**Exercise 33**

Fit curves to the other transitions. Do the numbers agree? *Hint:* The elegant solution uses a "**for**" loop.

Obviously, we have only scratched the surface of what "**physfit**" can do, but hopefully you now have a basis for further explorations on your own.