Analyzing Neural Time Series Data: Theory and Practice

Mike X Cohen

The MIT Press

This book was set in ITC Stone Serif Std by Toppan Best-set Premedia Limited, Hong Kong. Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Cohen, Mike X, 1979–
Analyzing neural time series data : theory and practice / Mike X Cohen.
pages cm.—(Issues in clinical and cognitive neuropsychology)
Includes bibliographical references and index.
ISBN 978-0-262-01987-3 (hardcover : alk. paper) 1. Neural networks (Neurobiology) 2. Neural networks (Computer science) 3. Computational neuroscience. 4. Artificial intelligence—Biological applications. I. Title.
QP363.3.C633 2014
612.8'2339—dc23
2013016461

10 9 8 7 6 5 4 3 2 1

4 Introduction to Matlab Programming

This chapter is not intended to be a complete guide to Matlab programming. Rather, the code accompanying this chapter introduces you to most of the commands used in this book, and the text provides some general advice and code-writing tips. There are many general Matlab introduction tutorials online. There are also several books from which to learn Matlab, including those written specifically for neuroscientists (Wallisch 2009) and behavioral scientists (Ramsay, Hooker, and Graves 2009; Rosenbaum 2007). However, it is a good idea even for readers with modest Matlab experience to go through the code accompanying this chapter: the code starts simple but gets increasingly complex. You should go through each of the three scripts accompanying this chapter on your own in Matlab, line by line and in order.

4.1 Write Clean and Efficient Code

Perhaps you think that as long as your code works, it does not matter how it looks or whether it is efficient. To some extent, this is true: cognitive electrophysiology studies are and should be evaluated based on the quality of the theories, hypotheses, experiment design, analyses, and interpretations; they should not be evaluated on the aesthetics of the programming code used to analyze the data.

That said, there are three reasons to try to write clean and efficient code. First and foremost, clean code is easy to read and understand. Having clean and easy-to-read code will help you prevent making programming errors, and when there are programming errors, it will help you identify and remedy those errors. Having clean code will also help you adapt existing scripts to new analyses and new datasets. If you write confusing and sloppily written code, you can probably read and interpret it when you first write it, but after weeks, months, or even years without looking at that code, you may get lost and not understand what you wrote. On the other hand, if your code is clean and properly commented, you will be able to return to the code after months or years of not looking at it and still understand what the code does and how to work with it. This is even more important if you share your code with other people and if you want other people to be able to understand and adapt your code. Again, clean and efficient code will help make their lives and data analyses easier.

There are three strategies to keep in mind that will help you write clean code. (1) Use comments. Write comments before a line of code or collection of lines of code to indicate succinctly what those lines do. Also use comments to specify the size of large matrices and the order of the dimensions in those matrices (e.g., time, frequency, condition, electrodes). Keep comments brief and to the point; do not use comments excessively, otherwise the code may become difficult to find from within a sea of comments. (2) Group lines of code by their common purpose. This is analogous to how you would group sentences into a paragraph. Think of a line of code as a sentence; several lines of code that have one purpose, or one message, should be grouped into a paragraph, separated by one or several blank lines. Relatedly, if there are many functions or commands on one line, consider breaking it up into two or more lines to be easier to read. (3) Use sensible and interpretable variable names (more on this point in section 4.2).

The second reason to write clean and efficient code is that efficient code runs faster. If your code runs faster, you will spend less time waiting for analyses to finish and more time looking at results and working on new analyses. To write efficient code, avoid redundancies (performing the same computations multiple times, e.g., evaluating an equation inside a loop when it could be evaluated outside the loop), pieces of code that are never called or that do nothing, separating into multiple files what would better be done in one file, or keeping in one file what could better be done in multiple files. Whenever possible, perform matrix manipulations instead of loops. Loops should be used only when necessary. Using informative comments will help you figure out ways to make the code more efficient.

The third reason to write clean and efficient code is that clean and efficient code promotes clear and organized thinking. Programming is problem solving. To program, you must first conceptualize the problem, then break it down into subcomponents, and then break those subcomponents further down into individual digestible computer commands. Programming involves taking an abstract idea for an analysis or figure you would like to show and turning that idea into a series of logical and concrete statements. These are also important skills for science in general: science involves taking an abstract idea about how some aspect of the brain or behavior works and turning that idea into a series of logical and concrete experiments, statistical analyses, and theory-relevant interpretations. There are overlapping skills that are useful both for programming and for being a scientist, and it is likely that working on one set of skills will transfer to the other set of skills.

Most of the hard work in programming should be done in your head and on a piece of paper, not in Matlab. Particularly if you are new to programming, start writing your script on a piece of paper with a pencil. Write down what the script should do and in what order. After you have a plan for how to write the script, then turn to your computer, open Matlab, and start programming.

After you consider the advice in the previous five paragraphs, take this one additional piece of advice: do not obsess over having the cleanest and most efficient code possible. It is not the most important part of being a cognitive electrophysiologist, and you can certainly be a great scientist without ever writing a single line of Matlab code. But as you write code, whether you are programming all of your own analyses from scratch or writing a script to call eeglab or fieldtrip commands, try to keep your code efficient and clear. Your future self will thank you.

4.2 Use Meaningful File and Variable Names

Give files useful names. For example, name a file "Flankers_task_TF_analyses.m" instead of "Untitled84.m." There is no shame in having long file names. Alternatively (or additionally), write commented notes at the top of each script that explain what the purpose of that script is. Note that some versions of Matlab on some operating systems will give errors when calling files that have spaces in the file names; you can use underscores instead.

It is even more important to give meaningful names to variables. Variables should have names that will allow you to identify and disambiguate the purpose of those variables from other variables. For example, if you have EEG data to store in a variable, it is better to call the variable something like `EEG_data` than something like `variable_name`. Avoid using variable names like `a, b, c, d` except if they are used only in small loops or as temporary variables that will be used only for a few lines of code. You can also develop your own style for naming variables. For example, I always put "i" at the end of counting variables in loops ("i" for index). This is particularly useful when using multiple nested loops over subjects (looping variable `subi`), channels (looping variable `chani`), frequencies (looping variable `freqi`), and trials (looping variable `triali`). In Matlab, variable names cannot start with numbers but may contain numbers, cannot have many nonalphanumeric characters (e.g., &, *, %, $, #), and cannot have spaces (underlines can be used instead).

4.3 Make Regular Backups of Your Code and Keep Original Copies of Modified Code

Spending hours on an analysis script only to accidentally delete or overwrite the file is not an enjoyable experience. Fortunately, Matlab automatically creates backup files (.asv or .m~ on Windows/Mac), which should help minimize accidental script loss, although these backups might be tens of minutes behind the original version. Matlab .m files used for scripts and functions contain only text and therefore occupy very little disk drive space, so there is little need to delete a file. You can, for example, email scripts to yourself to maintain time-stamped backups.

Avoid working on multiple copies of the same analysis script. For example, if you keep a time-frequency decomposition script for an experiment on the server computer, and then you back up that script on your portable USB drive, make sure you are modifying only one of those scripts. Otherwise you might end up making different changes to different versions of the script.

If you modify functions that come with Matlab or a third-party toolbox such as eeglab or fieldtrip, it is a good idea to save the original function with a different file name. For example, you can save the file as "filename.m.orig" and then modify "filename.m." The other option is to modify the original file and comment each line you modify or add. This latter option is suboptimal when you make many modifications because keeping track of every change you make may get cumbersome.

4.4 Initialize Variables

Initializing variables means that you reserve space in the Matlab buffer for that variable by creating the variable before populating it with data. Typically, the variable is set to contain all zeros, or all ones, or all NaNs (not-a-number). You do not need to initialize all variables, particularly smaller variables or variables that you use for only a short period of time. However, larger or more important variables, such as those that contain data you intend to analyze or save, should be initialized before use. In Matlab, unlike in some other programming languages, it is permitted to add elements or dimensions to variables without initializing them first (this is demonstrated in the online Matlab code [script "c"] that accompanies this chapter), but this behavior should be avoided when possible. There are three reasons why you should initialize variables.

First, initializing variables, particularly for large matrices, helps avoid memory crashes. Second, initializing variables that will be populated inside a loop helps prevent data from previous iterations of the loop contaminating current iterations. For example, imagine you have a script that imports and processes behavioral data, and the script contains a loop over subjects. There is a variable called `trialdata` that stores data from each trial within a subject before computing cross-trial averages. If the first subject has 500 trials and the second subject has 400 trials, without your initializing `trialdata` at each iteration of the loop over subjects, the data for the second subject will contain 500 trials, the last 100 of which were left over from the first subject. Obviously, this is a situation you want to avoid. Programming errors like this can be difficult to become aware of because they are unlikely to produce any Matlab errors or warnings. In some cases, you may not know how big a variable will be and thus cannot initialize it to the final size. If there is any danger of cross-loop-iteration contamination, you can clear the variable in an appropriate place (typically, the beginning of the loop), or you can initialize the variable to be bigger than necessary and then remove unused parts of the matrix afterwards. In this case it might be better to initialize the matrix to NaNs instead of zeros; if you accidentally forget to remove the unused part of the matrix, you do not want to average zeros into the data (the function `nanmean` will take the average of all non-NaN values in a matrix).

Another potential mistake that may be difficult to find because it will not produce a Matlab error or warning is the location of the initialization. Going back to the example from the previous paragraph, imagine further that you have another variable that contains trial-averaged data from each subject, called `subjectdata`. You should initialize `subjectdata` before the loop over subjects, and you should initialize `trialdata` within the loop over subjects. If you initialize `subjectdata` within the subject-loop, you will end up with a matrix of all zeros except for the last subject because data from previous subjects will be reinitialized at each iteration.

The third reason to initialize variables is that it will help you to think in advance about the sizes, dimensions, and contents of large and important variables. As noted at the end of section 4.1, the more thinking you do before and during programming, the cleaner, more efficient, and less error-prone your scripts are likely to be.

4.5 Help!

Even the most experienced and savvy programmers get stuck sometimes. There will certainly come a time when you need help, at least with Matlab programming. Matlab programming issues generally fall into one of three categories.

You know the function name but don't understand how it works. Start by typing `help <function name>` in the Matlab command. In some cases you can type `doc <function name>` to get a more detailed help file. Many functions have help files that contain examples; try running the example lines of code. Most functions are simply text files; you can open the function with `edit <function name>` and look through the code to try to understand how it works. This option is more useful when you develop some experience with programming and reading other people's code. Not all functions are viewable; some functions are compiled for speed. Try running the code with simpler inputs for which you can better understand the output. For example, if you have a large four-dimensional matrix and are unsure which dimension is being averaged in the mean function, try creating smaller matrices of only a few numbers, for which you can easily compute the means, and compare these against the function outputs. You can also plot the data before and after calling the function to see what effect the function had on the data. You can search the Internet to see if there are additional discussions of that function or additional examples of how to use the function. You can also ask colleagues for help. However, try to figure it out on your own before asking someone else. It may initially seem like a waste of time to spend 30 min understanding a function when a colleague could explain it to you in 30 s, but if you figure it out on your own, you are likely to learn more from that experience, and therefore, you are likely to avoid making that kind of error in the future. This is how you become a better programmer.

You know what you want Matlab to do, but you can't figure out the command or function for it. This is a frustrating problem. The three ways to solve this issue are by reading the help file of similar functions (in particular, look for the "see also" part at the end of the help file), searching on the Internet for what you want Matlab to do, and asking a colleague.

You know what you want Matlab to do and you know the command to do it, but there are errors when you run the code. Newcomers to Matlab seem to spend most of their time and frustration on this kind of Matlab issue. If your Matlab command window contains more red than black, don't give up hope; errors become less frequent over time as you learn from mistakes. Here are some tips for resolving this kind of Matlab error.

First, find the function or command that produces the error. This may sound trivial but can be tricky if there are multiple functions called in one line. For example, if you get an error with the following line of code:

```
abs(mean(exp(1i*angledata(:,trials4analysis)),2)),
```

you need to determine whether the error resulted from one of the three functions called (`abs`, `mean`, `exp`) or from one of the two variables (`angledata` or `trials4analysis`). To locate the error, start from the innermost (or most deeply embedded; most likely in the middle of the most parentheses or brackets) variable or function, and evaluate this in the Matlab command. In this case, start by evaluating `trials4analysis` and see if that produces an error. If not, move to the next function or variable—in this example, `angledata(:,trials4analysis)`. Eventually, you will find where the error occurs.

Now that you have located the error, read the error message (the red text). Sometimes, error messages seem to be written in a foreign language. If you do not understand the error, look for keywords. For example, if the error message contains "matrix size" or "matrix dimensions," use the size function to examine the dimensions of the variable that produced the error. If the error message contains "subscript indices must be real positive integers" or "index exceeds matrix dimensions," then probably your index variable (in the above example, `trials4analysis`) has zeros, negative numbers, fractions, or numbers greater than the size of the matrix being indexed. Some error messages are more self-explanatory, such as "incorrect number of input arguments."

If you still cannot solve the error, try plotting all of the inputs and outputs of the functions that precede the error; perhaps you will notice something strange or obvious in the plots. Missing data points in plots are likely to be NaNs or Infs (infinity)—these can cause errors in some functions or when used as indexing variables. You can also use the step-in option, which will halt the function that produced the error at the offending line. This is beyond the scope of this chapter, but you can read more about stepping-in on the Internet or in Matlab tutorials.

Other possible causes of an error include that the function is not in Matlab's path (and thus Matlab does not know the function exists), that the function is contained in a toolbox that you do not have, or that the function is compiled for or relies on libraries that are specific to a version of Matlab (e.g., 32-bit vs. 64-bit) or operating system. Errors can also occur if you use a variable name that is the same as a function name. For example, if you write `mean=my_data_part`; Matlab will recognize mean as a variable instead of the function. Using variable names that are existing functions is bad practice. If you want to know whether a name is already used by a function or existing variable, type `which` <name>.

4.6 Be Patient and Embrace the Learning Experience

Debugging Matlab code can be an infuriating and humiliating experience that makes you want to quit science and sell flowers on the street. But don't give up hope—it gets better. Embrace your mistakes and learn from them. Remember: no one is born a programmer. The difference between a good programmer and a bad programmer is that a good programmer spends years learning from his or her mistakes, and a bad programmer thinks that good programmers never make mistakes. I get annoyed when people think I am a good programmer because I can find and fix their bug in 30 s when they could not do it in 2 h. What they do *not* know is that I spent much more than 2 h finding and fixing that exact same bug in my own code, probably several times in the past. Eventually I learned to recognize what that bug is, where in the code it is likely to be found, and how to fix it.

Remember—time spent locating and fixing programming errors is *not* time lost; it is time invested.

4.7 Exercises

4.7.1 Exercises for Script A

1. Create a 4×8 matrix of randomly generated numbers.

2. Loop through all rows and columns, and test whether each element is greater than 0.5.

3. Report the results of the test along with the value of the matrix element and its row-column position. For example, your Matlab script should print `The 3rd row and 8th column has a value of 0.42345 and is not bigger than 0.5.`

4. Make sure to add exceptions to print out 1st, 2nd, and 3rd, instead of 1th, 2th, and 3th.

5. Put this code into a separate function that you can call from the command line with two inputs, corresponding to the number of rows and the number of columns of the matrix.

4.7.2 Exercises for Script B

6. Import and plot the picture of Amsterdam that comes with the online Matlab code.

7. On top of the picture, plot a thick red line from "Nieuwmarkt" (near the center of the picture) to "Station Amsterdam Centraal" (near the top of the picture).

8. Plot a magenta star over the Waterlooplein metro station (a bit South of Nieuwmarkt).

9. Find the maximum value on each color dimension (red, green, or blue) and plot a circle using that color. There may be more than one pixel with a maximum value; if so, pick one pixel at random.

4.7.3 Exercises for Script C

10. From the function you wrote for exercise 5, generate a 32×3 number matrix in which the three numbers in each row correspond to the row, column, and result of the test (1 for bigger than 0.5; 0 for smaller than 0.5).

11. Write this 32×3 matrix to a text file that contains this matrix along with appropriate variable labels in the first row. Make sure this file is tab-delimited and readable by a spreadsheet software such as Microsoft Excel or Open Office Calc.