NENS 230: Data Analysis Techniques for Neuroscience
Intro to MATLAB Debugging[1]

Daniel O'Shea, Sergey Stavisky
{djoshea, sstavisk}@stanford.edu

When writing code, it's often useful to be able to test small pieces of the code to make sure things are working correctly before worrying about whether the entire function or script is doing its job. MATLAB is an interpreted language, meaning that it looks at each line of code one line at a time and then executes it. This allows you to type code into the command line, work with data on the fly, try things out, and prototype very rapidly. Compare this to a compiled language like C or Java, where generally your code needs to be converted (compiled) into machine readable instructions. While excellent debugging tools for these languages exist, you can't really just run a line of code to see what happens without jumping through some hoops.

Assignments 2 and 3 were provided to you as incompletely written functions with ??? showing you the places where you needed to make edits. The rationale behind this was to help you through the assignment by explaining what needed to be done and in what order, so that you could focus on the new things you've learned (e.g. indexing last week, structures and data importing this week) rather than designing a solution to the whole problem from scratch. On the other hand, typically when you're creating a function from scratch you get to write the code however you want, and you build it up little bits at a time, testing and playing around with intermediate results often, rather than writing everything at once and testing at the very end.

The point of this document is to show you a few different techniques for writing and testing code that's either incomplete or buggy, which should hopefully help you complete the assignment with fewer headaches.

Technique 1: Write everything as a script, then convert to a function when it's working

As we discussed in lecture, functions nicely encapsulate functionality and isolate the variables used inside the function from the variables in your base workspace, i.e. the ones you create and manipulate from the command line. While this isolation works well when the function is already working and you just want to utilize it, it can make debugging more challenging. If the function you are working on crashes midway through execution, you return to the command window and none of the variables defined inside the function exist anymore, because you're back in the base workspace. This makes it hard to figure out what values variables had right before things stopped working.

On the other hand, scripts (.m files without a function signature at the top) work exactly as if you had copied and pasted the contents onto the command line. All variables created in the script will exist in the workspace it was called in (in this case, the base workspace assuming you called it from the command line). So if a script crashes mid-run, it stops executing, but you can poke around at the

variables it created up to that point, which could help you figure out why the line of code that failed didn't work correctly (e.g. a certain variable is a row vector but it needs to be a column vector).

Therefore what I will often do is convert the function to a script temporarily while writing it or fixing problems with it. Say you had the following function:

```
function out = sqrtNewton(in)
% sqrtNewton Finds the square root of a number using Newton's Method
% out = sqrtNewton(in)
%
% INPUTS
% in - the number to take the square root of
%
% OUTPUTS
% out - the square root of in

out = 1;
for i = 1:100
	out = (out+in/out)/2;
end

end
```

You could turn this into a script by commenting out the function signature and the tailing end keyword. Then you'll want to define the inputs explicitly, since you'll no longer be providing them as arguments. You can either do this on the first few lines of the script (as is done below), or just assign a value into them at the command line.

```
%function out = sqrtNewton(in)
% sqrtNewton Finds the square root of a number using Newton's Method
% out = sqrtNewton(in)
%
% INPUTS
% in - the number to take the square root of
%
% OUTPUTS
% out - the square root of in

in = 5; % temporarily define input arguments!

out = 1;
for i = 1:100
	out = (out+in/out)/2;
end

% comment this out since the function keyword it corresponds to is gone!
% end
```

Then when you're satisfied that everything's working, simply un-comment the function signature, un-comment the tailing end keyword, and delete or comment out the lines where you define the input arguments. If you look in the Text menu of the Editor window, you'll see options (and useful keyboard shortcuts) for commenting and uncommenting multiple lines of text at once.

Technique 2: Setting breakpoints

MATLAB has a built-in framework for debugging scripts and functions. If you're in the editor window, you'll notice that the left margin has horizontal dashes next to each line of the file that has executable code on it. If you left-click on one of these dashes, it will turn into a red dot, which represents a break point. If you left-click on the red dot, it will disappear and revert to the dash. When MATLAB is running a function or script and comes across one of these breakpoints, it will pause execution of the code and bring your cursor and focus to that line of that file in the editor. The current line will be indicated with a green arrow (Figure 1).

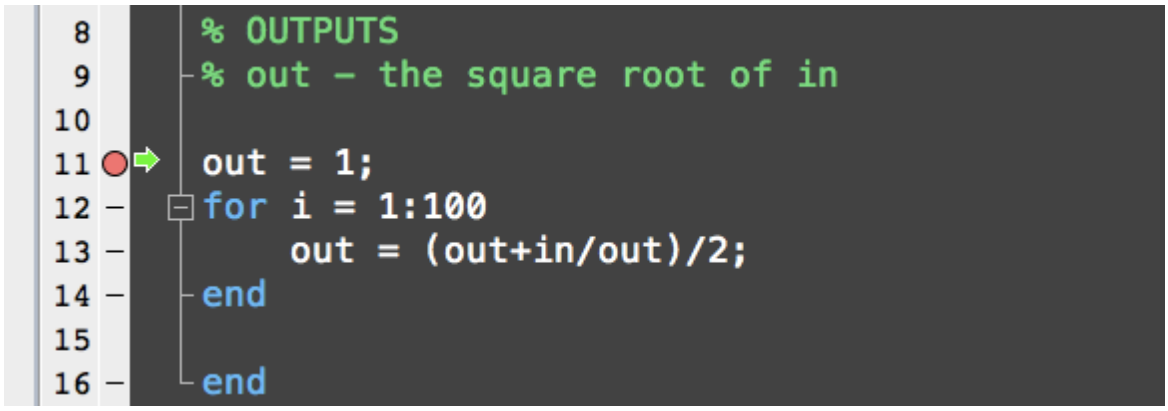

Figure 1: Execution paused at a breakpoint set on line 11.

You've now entered debug mode. You are now "inside" the function, meaning that any variables that were defined in that function are now accessible from the command window, whereas any variables that existed outside the function (in the base workspace) are not accessible. You can now hover over variables in the function, and you'll see a popup tooltip that shows you information about that variable and possibly the contents of that variable if it doesn't take up too much space. You could also see the contents of the variable by typing its name into the command line, or call other functions like `plot()` or `size()` to learn more about what is stored in a variable of interest. You can even alter the values stored in specific variables if you like, and these changes will be reflected in code that runs after this point. But keep in mind that anything you change is only temporary, and it won't be changed like that the next time you run this code. Temporarily changing the value of a variable by hand is useful if you notice something is awry and want to see if you can fix it quickly before continuing execution. You'd then want to alter the code to fix the underlying problem.

When you're satisfied and want to keep executing the m-file, you can use the debugging toolbar (Figure 2) to continue execution. If you don't see this toolbar, click inside the editor window to give it focus, select Desktop->Toolbars in the main menu, and then make sure Editor is checked in the menu. If you still don't see these buttons on the toolbar, they might be hidden because the window is too small (try clicking the >> at the edge of the toolbar to see any hidden buttons), or you can right click the toolbar and click Customize... to add these buttons to the Editor toolbar, though they should be there by default.

Here's what each of the buttons does. The set/clear breakpoint button will toggle whether a breakpoint is defined on the current line (where the cursor is). The clear all breakpoints in all files will do exactly that, which is useful if you've figured out where the problem is and now don't want the code to pause in debug mode everywhere you just added breakpoints. The "Step" button will execute the next line of code and then pause again. The "Step Into" button will execute the next line of code, but if that code calls another function, you'll pause inside that new function instead of executing it as a whole. Watch out, as you can find yourself inside MATLAB's own built-in functions if you do this sometimes. The "Step Out" button instructs MATLAB to finish executing the current function and then return back to the caller. If you pressed "Step Into" and found yourself inside a function that you're not interested in, say inside plot.m, then you can use "Step Out" to get back to where you were. The "Continue" button simply resumes execution until you hit another break point later on. The "Exit Debug Mode" aborts execution (this happens automatically if you make changes and then save).

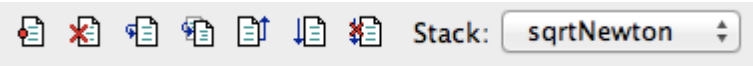

Figure 2: The debugging toolbar. From left to right: set/clear breakpoint, clear all breakpoints in all files, step, step into, step out, continue, exit debug mode, select workspace drop-down.

At the far right, you'll notice a drop-down which allows you to select which workspace on the stack you wish to work in. Remember that functions operate in their own isolated workspace, so if you call functionOuter, which calls sqrtNewton, and you set a breakpoint in sqrtNewton, you'll be debugging in sqrtNewton's workspace, which means you'll only see variables created in or passed in as inputs to sqrtNewton. If for some reason you'd like to see what variables existed in the calling function, functionOuter, you can select its workspace in this dropdown. This will automatically take you to the line of code in functionOuter.m which called sqrtNewton, and you'll now be able to access variables in functionOuter at the command line (but not variables that exist inside sqrtNewton).

There are also command window functions you can call that do the same thing as the toolbar buttons if you prefer to work from the command line:

Toolbar button function	**Command line function**
Set/clear breakpoint	`dbstop in` *functionName* `at` *lineNumber* `dbclear in` *functionName* `at` *lineNumber*
Clear all breakpoints in all files	`dbclear all`
Step	`dbstep`
Step In	`dbstep in`
Step Out	`dbstep out`
Continue	`dbcont`
Exit debug mode	`dbquit`
Select a different workspace on the stack	`dbup` `dbdown`

One additional command worth knowing is `keyboard`. If you call `keyboard` from inside a function or script, it's effectively the same as if you'd set a breakpoint on the line after the keyboard command. The main difference is that since `keyboard` is actually part of the function's code (saved to disk) as opposed to marked with a red dot in MATLAB's interface, you're guaranteed that that "keyboard" command will still exist the next time you open MATLAB, even on a different machine. Sometimes this is useful, though often just setting a breakpoint in the Editor is less tedious.

Technique 3: Debug automatically when an error occurs

Inevitably, in the course of writing code you'll run into an error that is unexpected and you're not entirely sure what caused it. Usually, when an error is encountered, MATLAB prints out the error message to the screen, beeps, and then aborts execution, leaving you back in the base workspace with none of the variables that were defined inside your function still existing.

When you encounter this, you can simply set a break point on the line where the error happened, and you'll automatically go into debug mode before executing this line, so you can take a look at what each variable is set to and try executing the offending line of code (or a portion of it) at the command line to see what's going on. However, what if the error only occurs on the 5000th iteration of a given loop? MATLAB respects the breakpoint *every time* that line of code is executed; thus you'd have to hit continue 4,999 times before you encounter the situation which lead to the error.

A preferable approach is to tell MATLAB that whenever it encounters an error, to pause right there and enter debug mode as if it had stopped just before running the line of code which caused the error. Then you can see exactly what each variable is set to, use the command line, and figure out what's wrong. Of course, you can't use step or continue to keep executing; once the error happens, MATLAB will abort execution. But this approach allows you to pause right before aborting to see what's going on, and then fix it.

To turn on this mode, run:
```
dbstop if error
```

To turn off this mode, run:
```
dbclear if error
```

Technique 4: Use encapsulation, build small functions that do one thing well

One nice thing about functions is that you can take some small bit of functionality and wrap them into a function. You can then test this function thoroughly, and not have to worry about breaking this functionality when writing the rest of your code. It's a lot easier for you to write your assignments without having to worry about whether Forrest's `abfload` function is doing its job correctly, or whether MATLAB's `textscan` or `plot` functions are implemented correctly according to their documentation. Similarly, it's much easier to get one part of your code working well, put this in a function and test it, then move onto more challenging things. This also helps you to avoid duplicating your efforts, because once you write this function, you can use it repeatedly rather than having to re-implement that functionality in place multiple times.

We hope this helps! We also encourage you check out the introduction to debugging in MATLAB's documentation at http://www.mathworks.com/help/techdoc/matlab_prog/f10-60570.html

Good luck coding!