

Starting thinking about final project

- Can be (but need not be) related to your own research. For instance, write a program to analyze data, visualize data, automate your workflow, create a software tool, model some system
- An brief proposal is due on 11/17. 1-2 paragraphs (1 page *max*) about what you want to do, and how you might approach it. Feel free to bounce ideas off of us!
- Final project due 12/9
- Ambitious projects need not be complete, but should demonstrate progress

Lecture 5: Best Practices

Lecture 5 Outline

Modular code: functions

Debugging tools

Measuring Performance

Improving Efficiency

Code Style

Assignment 5 Overview

Lecture 5 Outline

Modular code: functions

Debugging tools

Measuring Performance

Improving Efficiency

Code Style

Assignment 5 Overview

Functions

Functions allow you to **encapsulate** a set of operations and calculations into a callable module with inputs and outputs

Functions are saved as .m files (like scripts)

Functions begin with a *signature*, which contains the `function` keyword and lists the *outputs* and *inputs* by their internal name.

Writing functions lets you re-use code to do a particular job.



Function Example

```
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
```

Function Example

Function signature: names the inputs and outputs

function keyword tells MATLAB this is a function being declared

```
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
```

Function Example

Outputs: whatever value you store in the variables you name here will be returned to the caller (i.e. script or function that called sqrtNewton 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
```


Function Example

Inputs: whatever arguments the caller passes in will be assigned into these variables *inside* the 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
```

Function Example

Function Name: your function .m file should be named this on disk, e.g. sqrtNewton.m. MATLAB cares more about its filename than the name here, but they should match to avoid confusion.

```
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
```

Function Example

Documentation: tells the user how to use this function. Good practice to include a quick summary and what the inputs and outputs mean. Displayed if `help sqrtNewton` is called.

```
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
```

Function Example

Function code: runs when you call the 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
```

Function Example

Closing `end` keyword: not strictly necessary, but generally good practice

```
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
```

Scope: where variables exist

When you **encapsulate** code in a function, that code executes in an **isolated workspace**.

- The function's code only sees the values of variables that are passed in as inputs
- Only the values you return as outputs make it back to the caller, and they're assigned into the variables that the caller specifies

Scope: where variables exist

```
function foo = exampleFunction(x)  
→   x = 2*x;  
   foo = 3*x;  
end
```

Inside exampleFunction:

x == 1

```
x = 1;  
z = exampleFunction(x);
```

outside the function:

x == 1

Scope: where variables exist

```
function foo = exampleFunction(x)
    x = 2*x;
    foo = 3*x;
end
```

Inside exampleFunction:

```
x == 2
foo == 6
```

```
x = 1;
z = exampleFunction(x);
```

outside the function:

```
x == 1
```


Scope: where variables exist

```
function foo = exampleFunction(x)
    x = 2*x;
    foo = 3*x;
end
```

```
x = 1;
z = exampleFunction(x);

display(x)
display(z)
display(foo)
```

Inside exampleFunction:

– nothing –

outside the function:

x == 1

z == 6

error: foo
doesn't exist

Scope: where variables exist

```
function foo = exampleFunction(x)
→   x = 2*x;
    foo = 3*x;
end
```

```
x = 1;
z = exampleFunction(x);
```

```
display(x)
display(z)
display(foo)
```

```
display(exampleFunction(z))
```

Inside exampleFunction:

```
x == 6
x == 12
foo == 36
```

outside the function:

```
x == 1
z == 6
```

Scope: where variables exist

```
function foo = exampleFunction(x)
    x = 2*x;
    foo = 3*x;
end
```

Inside exampleFunction:

– nothing –

```
x = 1;
z = exampleFunction(x);
```

```
display(x)
display(z)
display(foo)
```

outside the function:

x == 1

z == 6

ans == 36

```
display(exampleFunction(z))
```

Multiple Input Arguments

A function can be called with **multiple input arguments**:

```
function y = changeOfBase(x,newBase)
% returns y = log_newBase(x)
    y = log(x)/log(newBase);
end
```

`y = changeOfBase(8, 2)` returns 3

Can have as many input arguments as you define in the function signature

... but if `changeOfBase` is called without the defined number of arguments...

`y = changeOfBase(8)` **Error using changeOfBase (line 3)**
 Not enough input arguments.

What if we want to have `newBase` default to 2 if we don't specify it?

Optional Input Arguments

Option 1: set a default argument:

```
function y = changeOfBase(x,newBase)
% returns y = log_newBase(x)
    if isempty( newBase )
        newBase = 2;
    end
    y = log(x)/log(newBase);
end
```

```
y = changeOfBase( 8, [] )
```

```
y == 3
```

Optional Input Arguments

Option 2 (better): allow variable number of input arguments

`nargin` is a special variable that returns how many arguments were passed in

```
function y = changeOfBase(x,newBase)
% returns y = log_newBase(x)
    if nargin < 2 || isempty( newBase )
        newBase = 2;
    end
    y = log(x)/log(newBase);
end
```

```
y = changeOfBase( 8 )
```

```
y == 3
```

```
y = changeOfBase( 8, 4 )
```

```
y == 1.5
```

Multiple Output Arguments

A function can have **multiple outputs** by calling it with the following syntax:

```
[out1, out2, out3] = myFunction( )
```

To make a function have multiple outputs, define them in the function definition:

```
function [out1, out2, out3] = flexibleFunction( )
```

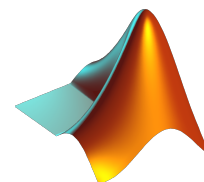
You **must create all output** variables somewhere in the body of the function

If you don't want all of the outputs when calling a function, put a tilde in the place of unneeded outputs:

```
[~, out2, ~] = myFunction( )
```

e.g. [~, maxTime] = max(valueOverTime) **% don't need maxValue**

(advanced) `varargin` combined with `nargin` and `varargout` combined with `nargout` lets you have variable number of input and output arguments (similar to variable number of input arguments)



flexibleFunction.m

Functions and Modularity

Choosing what to make its own function is a matter of personal style and experience

Rules of thumb:

- If you'll do it again in your program, make it its own function
- Make functions flexible by adding optional arguments rather than creating a very similar function
- Separate data importing, data processing, and data visualization into their own functions

Lecture 5 Outline

Modular code: functions

Debugging tools

Measuring Performance

Improving Efficiency

Code Style

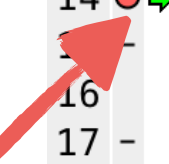
Assignment 5 Overview

Debugging Tools: breakpoints

Let's you "pause" code execution and look around to figure out what's going on. Useful not just to find error-generating bugs, but also to "step through" working code and better understand it.

Technique 1: Setting a "breakpoint"

```
1  function out = sqrtNewton(in)
2  % sqrtNewton Finds the square root of a number
3  % using Newton's Method
4  % out = sqrtNewton(in)
5  %
6  % INPUTS
7  % in - the number to take the square root of
8  %
9  % OUTPUTS
10 % out - the square root of in
11
12 -   out = 1;
13 -   for i = 1:100
14 -       out = (out+in./out)/2;
15 -   end
16
17 - end
```



Click on a line next to executable code to set and remove breakpoints


Debugging Tools: keyboard

Let's you "pause" code execution and look around to figure out what's going on. Useful not just to find error-generating bugs, but also to "step through" working code and better understand it.

Technique 2: Add keyboard to code to set a breakpoint

```
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
    keyboard
    out = (out+in./out)/2;
end
end
```

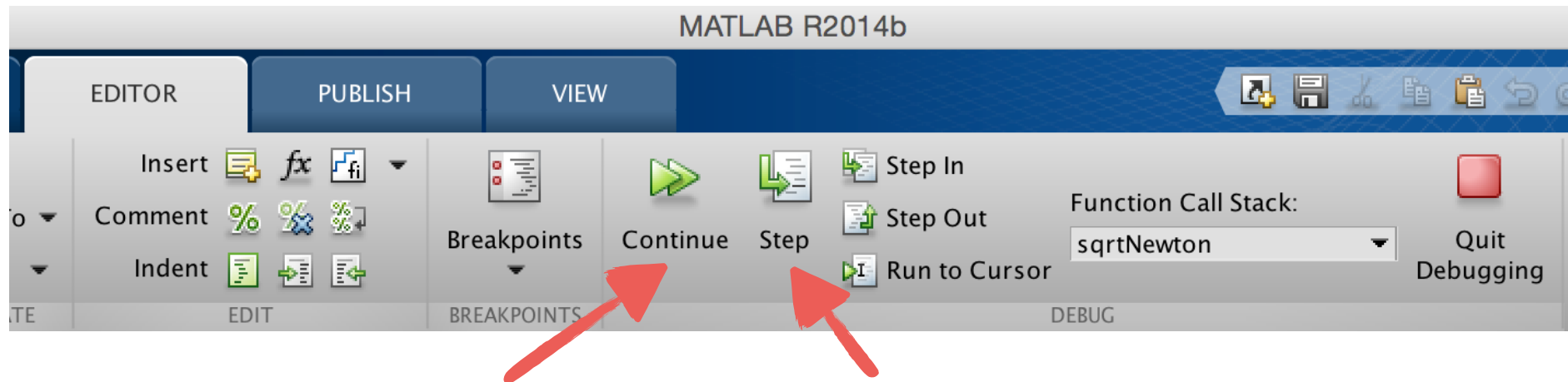


Don't forget to delete it
when you're done debugging

Debug Technique 3: Step through code

Can go line by line and see what happens, or skip to next breakpoint

Can do this from command line with `dbstep`, `dbcont`



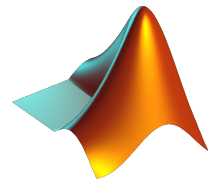
Debug Technique 4: dbstop if error

Super useful: type `dbstop if error` in the command line.

Then run your code.

It will automatically stop when an error is encountered.

`dbclear all` removes all breakpoints including `dbstop if error`

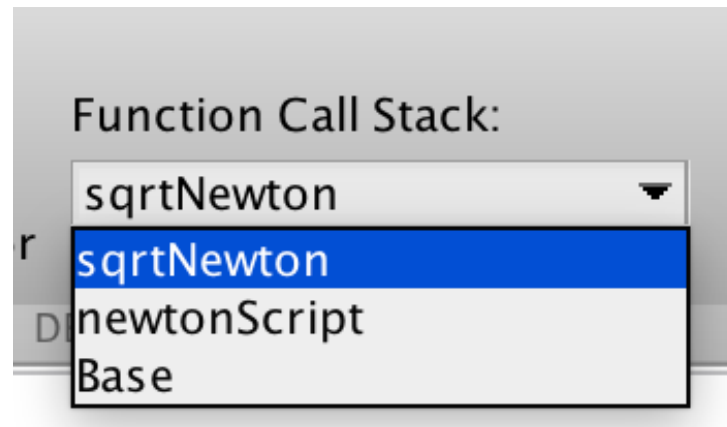


`dbstop if error` in `sqrtNewton.m`

You can move between scopes in debug mode

The Function Call Stack menu (under Editor) lets you see the variables visible in different functions' scope.

Can also access this from command line with `dbstack`, `dbup`, `dbdown`



Lecture 5 Outline

Modular code: functions

Debugging tools

Measuring Performance

Improving Efficiency

Code Style

Assignment 5 Overview

Why do we care about performance?

We sometimes care about code's **performance**, which has two parts:

Memory Usage - How much memory will your code need (peak, and post-execution)?

Computation Time - How quickly will your computation be completed?

Given today's computers, performance usually only matters when working with large datasets or doing computationally complex analyses (e.g. bootstrap statistics)

Also matters if you need the program to execute very quickly (e.g. it's controlling an experiment)

Measuring Memory Use

Variables (and objects) that exist consume **memory** (also called “RAM”; this is not the same as disk space)

How much memory MATLAB has available depends on your computer hardware, operating system, and what else is running

Typically, MATLAB will have access to 1-8 GB of memory on a standard laptop

If running MATLAB in Windows OS, you can query how much memory is available using `memory`

You can see what variables are in your workspace, and how large they are, with `whos`



You can store this in a structure using:

```
sInfo = whos( 'specificVar' )
```

where `'specificVar'` is an optional string argument that tells `whos` to return info about just the variable `specificVar`

When you delete a variable using `clear('varName')` or by a function terminating, the memory is returned to MATLAB for reuse

Measuring Computation Time With `tic`, `toc`

Every command that MATLAB executes takes a certain **compute time**

Some operations are much more expensive than others

Sometimes you can do the same thing orders of magnitude faster by changing your algorithm

The first step to speeding up your code is measuring how long various parts of it take

You can measure how fast a group of commands is executed by a “stopwatch”-like tool:

- use the command `tic` to start timing

- `toc` returns the elapsed time (in seconds) since `tic` was called

You can run multiple of these “stopwatches” at once as follows:

```
tic1 = tic;  
tic2 = tic;  
timeSinceTic1 = toc( tic1 );  
timeSinceTic2 = toc( tic2 );
```



Measuring Computation Time Using Profiler

The **profiler** gives you very detailed statistics about how various parts of your program contribute to its total compute time

Open it from **Home** —> **Code** —> **Run and Time**

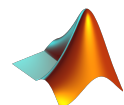
Calls is how many times a function was called

Total time is how long that function took to complete, *including subfunctions*

Self-Time is how long it took, *excluding subfunctions*

Click on a function name for a more detailed breakdown of its subfunctions, and visual depiction of what the slowest parts are

Running the profiler slows down execution, so look at relative compute time, not absolute times



Profiler Example

Lecture 5 Outline

Modular code: functions

Debugging tools

Measuring Performance

Improving Efficiency

Code Style

Assignment 5 Overview

Improving Memory Efficiency

Delete variables that you don't need anymore:

```
% Load raw data
rawDat = load( 'voltageTrace.abf' );
spikeTimes = extractSpikeTimes( rawDat ); % extracts spike times
clear( 'rawDat' ) % Don't need the raw data anymore, and it is large
```

Don't keep everything in memory at once. If raw data is very large, process it in pieces:

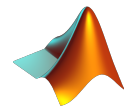
```
% Raw data comes in several large data files
spikeTimes = []; % will have list of all my spike times across recording
for iFile = 1 : numRawFiles
    thisRawDat = load( fileList{iFile} );
    spikeTimes = [ spikeTimes ; extractSpikeTimes( thisRawDat ) ];
    clear( 'thisRawDat' ) % Don't need the raw data anymore, and it is large
end
```

Use smallest precision data type that fully represents your data:

x = 1;	double	8 Bytes
x = single(1);	single	4 Bytes
x = int16(1);	int16	2 Bytes
x = int8(1);	int8	1 Byte
x = boolean(1);	logical	1 Byte [why not 1 <i>bit</i> ? It's a MATLAB peculiarity...]

Speeding Up Computation 1: Display Less Output

Display outputs such as `fprintf`, `display`, or “unsuppressed” output (commands with no semicolon after assignment) burn a lot of time

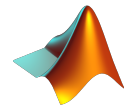


`fasterCodeDemo.m`

Speeding Up Computation 2: Preallocation

One of the easiest and most effective ways to speed up your code is to **preallocate variables** rather than growing them in a loop

Can preallocate with any value; typically the `zeros()` command is used



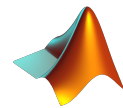
fasterCodeDemo.m

Speeding Up Computation 3: Vectorization

Vectorization refers to doing tasks as array operations rather than in loops

MATLAB has a very fast underlying linear algebra library

Performing one operation on an array of N elements (i.e. vectorized) is much faster than performing N operations, each on a scalar (for loop)



fasterCodeDemo.m

More Advanced Vectorization Using Repmat

`repmat` lets you replicate a matrix (thus, also a vector or scalar):

```
tilled = repmat( original, repeatRows, repeatCols )
```

Ex: `B = repmat(A, 3, 2)`

2 horizontal repetitions

3 vertical repetitions

23	3	23	23	3	23
15	15	4	15	15	4
23	3	23	23	3	23
15	15	4	15	15	4
23	3	23	23	3	23
15	15	4	15	15	4

Speeding Up Computation

- Less display to command line
- Pre-Allocate
- Vectorize when possible

Lecture 5 Outline

Modular code: functions

Debugging tools

Measuring Performance

Improving Efficiency

Code Style

Assignment 5 Overview

Code Style

Code that does the same exact same computations can be written differently

Good “**coding style**” means writing code that is easy to read and understand

No one “right way”: we’re presenting some useful conventions

Naming Functions

Give descriptive names to functions:

e.g. `normalizeByMax(...)`, `removeWonkyData(...)`, `loadMicroscopeData(...)`

Function definition inside a .m file should have same name as the .m file itself

Separate words with capital letters (e.g. `binSpikeTimes`) or underscore (e.g. `bin_spike_times`)

Sometimes useful to have the “main” or “entry” function or script start with capital letter or be all capitals, e.g. `GenerateFigure1.m` or `GENERATE_FIGURE_ONE.m`, which has helper functions `loadFig1Data(...)`, `analyzeFig1Data(...)`, `plotFig1Data(...)`, etc.

A good directory structure can help you stay organized. For example:



Avoid spaces or non-alphanumeric characters in any folders that MATLAB will be accessing as this can be occasionally problematic

E.g. don't name a folder `/sstavisk/Lab/My Questionable(?) Data Directory/`

Function Header

Long comment that goes either before or after the function definition line:

functionName.m

```
function [outvar1 outvar2] = functionName( arg1, arg2 )
% Here I'm going to describe what this function does, and maybe when it's
% used and what limitations it might have.
% NOTE: Put important messages here,
%       e.g. Watch out, if you enter the wrong arg2, the computer explodes!
%
% USAGE:
% [outvar1 outvar2] = functionName( arg1, arg2 )
%
% INPUTS:
%     arg1          This does blah
%     (arg2)        (optional) This specifies bleh, which is optional.
%
% OUTPUTS:
%     outvar1       This will be useful
%     outvar2       This might be too!
% Created by PI-WHEN-HE-WAS-A-STUDENT 1 on 25 December 1999
% Last Edited by GRAD_STUDENT on 4 October 2015

    code1 = hereBeCode( arg1 );
    code1 = moreCodeHappened(code1) arg2;
    ...
end
```

Naming Variables

Similarly, give descriptive names to variables. (e.g. `startTime`, `maxBrightness`)

Especially important to give descriptive names to looping **index variable**

Vague

```
for i = 1 : numChans
    for j = 1 : numTrials
        for k = 1: numSpike
            % Block of code to make rasters
        end
    end
end
```

Informative

```
for iChan = 1 : numChans
    for iTrial = 1 : numTrials
        for iSpike = 1: numSpike
            % Block of code to make rasters
        end
    end
end
```

Constants are often given all capital names

e.g. `SAMPLE_RATE = 60`; `BOLTZMANN = 1.38e-23`;

Some people come up with a convention for identifying types of variables, e.g. `goodTrialsIdx` for indices, or `keepGoingBool` for boolean/logical

Blank Lines

Put one blank line between “steps” within the same task, and two to mark a larger change of what you’re doing

```
% DATA IMPORT
% load data, preprocess
in = load('mydata.dat');
exons = in.stuff(:,1);
introns = in.stuff(:,2);
phenotype = in.appearance;

% DATA ANALYSIS
code block that does your analysis
some code to process exons and introns

Code processing phenotype
more phenotype stuff

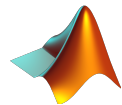
% PLOT THE DATA
figh = figure;
scatter( processedGenotype, processedPhenotype)
```


Indenting

You should **indent** insides of loops, conditional statements, and (optionally) functions

Nested blocks are indented one level deeper, analogous to indented bulleted list

Highlight code and Command-i (Mac/Linux) or Ctrl-i (Windows) to automatically do indenting



indentExample.m

Disambiguating Parentheses

MATLAB uses parentheses for both function **arguments** and variable **indexing**

One way to disambiguate is by putting an outside space around arguments:

e.g. myFunction(arg1, arg2) %calls function

versus myVariable(colIdx, rowIdx) %returns array element(s)

Comments help people read code

Comment either before a line of code, or directly after the code for a short comment :

```
% Now I'm going to compute my average stimulus using the previous  
% imported data and params that determine what smoothing to use.  
myAvgStim = computeStimulus( data, params );  
myAvgStim = abs( myAvgStim ); % don't care about sign
```

Can use symbols to make big, easy-to-read **section headers**:

```
% *****  
%                               MAJOR HEADER  
% *****  
  
% -----  
%                               Minor Header  
% -----
```

Double %% divides code into **cells**:

```
%% This makes cell 1  
code;  
  
%%  
% That made another cell  
  
morecode;
```

MATLAB lets you execute
one cell at a time

Good Code Style

Good names for functions, variables

Blank lines separate sections

Logical flow is shown with indents

Parentheses spacing cues functions vs. variables

Code is commented

Lecture 5 Outline

Modular code: functions

Debugging tools

Measuring Performance

Improving Efficiency

Code Style

Assignment 5 Overview

Assignment 5: Improving Bad Event-Triggered Average Function

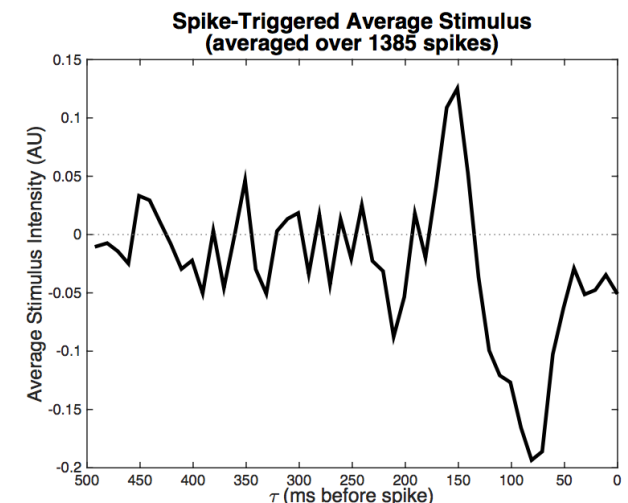
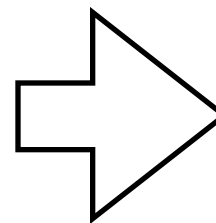
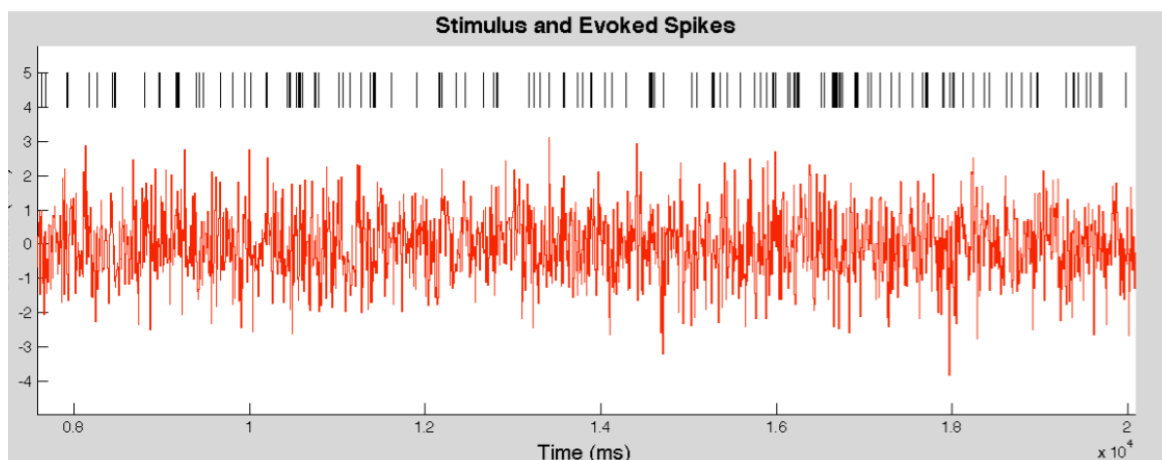
You will be provided with a program which almost works but is really awful:

- Has 1 small bug (use debugger!)
- Bad coding style
- Doesn't use functions for modularity
- Inefficient

Your job will be to debug it and then edit it to make it both more readable, and also run faster.

You will have to figure out what various parts of the program do. Working out other people's code is an important skill.

Breakpoints and/or stepping line by line can help you figure out what's going on



Assignment 5: Improving Bad Event-Triggered Average Function

Extra Credit: Whoever's improved code runs the fastest wins a prize!



Lecture 5 Review

Key Concepts

Functions take in arguments, do some operations, and return outputs

Breaking apart a long program into multiple functions makes your code easier to read and more **modular**

Generally, functions don't know about variables outside their “scope”

varargin allows you to have a **variable number of function inputs**

varargout allows you to have a **variable number of function outputs**

nargout and **nargin** can be called inside a function to report the caller’s expectations

The **debugger** lets you “pause” the code either at a specific line, or when an error happens

In debug mode, you can step through your code line by line

Both **memory** and **compute time** performance can be important

whos tells you about variables in the workspace, including their **memory size**

You can measure how long operations take using **tic** and **toc**

The **profiler** is a great way to spot bottlenecks

To **save memory**, delete variables after use, process in pieces, and use smaller data types

Preallocating a variable and then filling it up is much faster than expanding it in a loop

Vectorized operations work simultaneously on matrices instead of individual elements

Most computations can be done in a vectorized manner, and this is usually faster

repmat allows you to replicate any matrix, and helps vectorizing many operations

Naming of variables and functions should obey **coding style** best practices for clarity

Indenting of code doesn't affect execution but helps flow control readability

Blank lines helps separate conceptually related lines

Code should be **documented** with **function headers** and descriptive **comments**

Functions

function

return

varargout

nargout

varargin

nargin

keyboard

dbstop if error

whos

tic

toc

repmat

reshape