Cell Arrays, Struct Arrays, Debugging

NENS 230: Analysis Techniques in Neuroscience
Fall 2014

Outline

1. **Control flow review**
    1. `if`, `for`, `while`, `switch`
2. **Cell arrays**
    1. Creating and concatenating cell arrays
    2. `{}` Indexing vs. `()` Indexing
3. **Struct arrays**
    1. Associating related data
    2. Structs, struct arrays, and indexing
    3. Accessing fields, aggregating data across a struct array
4. **Functions and variable scope**
5. **Debugging tools**
    1. Breakpoints
    2. Stepping through code
6. **Start Assignment 4**

Outline

1. **Control flow review**
    1. `if`, `for`, `while`, `switch`
2. **Cell arrays**
    1. Creating and concatenating cell arrays
    2. `{}` Indexing vs. `()` Indexing
3. **Struct arrays**
    1. Associating related data
    2. Structs, struct arrays, and indexing
    3. Accessing fields, aggregating data across a struct array
4. **Functions and variable scope**
5. **Debugging tools**
    1. Breakpoints
    2. Stepping through code
6. **Start Assignment 4**

if statement

Runs a block of code if a certain condition is true

```
if condition
     % run this code if it’s true
     % i.e. evaluates to anything but 0
end

if errorCount > 0
    error(‘PC Load Letter’);
end
```

Branching: `if, else`

Runs a block of code if a certain condition is true

```
if condition
      % run this code if it’s true
else
      % run this code if it’s false (0)
end
```

Branching: `if, else`

Runs a block of code if a certain condition is true

```
if condition1 && condition2
    % run this code if both are true
else
    % run this code if either’s false
end
```

Branching: `if, else`

Runs a block of code if a certain condition is true.
Runs a different block of code if not.

```
if condition1 || condition2
    % run this code if either is true
else
    % run this code if both are false
end
```

Branching: `if, elseif, else`

Runs a block of code if a certain condition is true. If not, run another block of code if another condition is true. If not, ... (and so on). If none of the above is true, run the else block.

```
if condition1
    % run this code if it’s true
elseif condition2
    % run this code if condition1 is
    % false and condition2 is true
else
    % run this code if neither is true
end
```

Looping: `for` loops

Run a block of code once for every element in a list, assign into an index variable the current element in that list.

```
for i = 1:nIterations
     % run this code once with i == 1
     % then once with i == 2
     % ...
     % then with i == nIterations
end
```

Looping: `while` loops

Keep running a block of code as long as the condition is true.

```
while  some condition that evaluates to 0 or 1
     % keep this code while it’s true
end
```

Looping: `break` and `continue`

Valid only inside a loop (e.g. `for` or `while`)

`break` means stop right here, exit the loop, and begin executing the code after the end keyword

```
while true
     % code that does something
     if doneThisLoop
         % abort the loop and jump to below
         break;
     end
end
% below!
```

Looping: `break` and `continue`

Valid only inside a loop (e.g. `for` or `while`)

`continue` means stop right here, skip to the next iteration of the loop, and start on the first line of the loop

```
for iSweep = 1:nSweeps
     % code that does something
     if skipThisSweep
         % abort this iteration and continue on
         % the next iteration
         continue;
     end
end
```

Branching: `switch–case` block

When there's a long list of possible options, you can simplify the `if–elseif–elseif–elseif`… statements into a `switch–case` block.

```
switch value

     case 1

          % run this code if value == 1

     case 2

          % run this code if value == 2

     otherwise

          % run this code if value is none

          % of the above

end
```

Outline

1. **Control flow review**
    1. `if, for, while, switch`
2. **Cell arrays**
    1. Creating and concatenating cell arrays
    2. `{}` Indexing vs. `()` Indexing
3. **Struct arrays**
    1. Associating related data
    2. Structs, struct arrays, and indexing
    3. Accessing fields, aggregating data across a struct array
4. **Functions and variable scope**
5. **Debugging tools**
    1. Breakpoints
    2. Stepping through code
6. **Start Assignment 4**

Why are data types so important?

$$\begin{tikzcd}\boxed{\text{Raw Data}} \arrow[r] & \boxed{\text{Code}} \arrow[r] & \boxed{\text{Processed Data}}\end{tikzcd}$$

Examples:

- Voltage clamp traces
  - Current/signal, organized by channel, time
- Image file
  - Intensity, organized by channel, x pos, y pos, z pos

Processed form:

- Opsin tracking by frequency
  - Spikes evoked, organized by pulse frequency
- Cell positions
  - List of x,y,z coordinates for each cell detected

Other data structures

In addition to numeric arrays, there are:

- Character arrays or strings
- Cell arrays
- Structures

Fortunately, indexing works in pretty much the same way for all of them.

Lists of strings?

What if we had a list of several strings? How would we keep track of all of them?

```
channelName1 = ‘gfp’;
channelName2 = ‘dapi’;
channelName3 = ‘neun’;
              .
              .
              .
```

This is not a good idea, because any code that operates on each of the channels necessarily has to repeat itself N times.

Lists of strings?

Why can’t / shouldn’t we do something like this?

```
channelNames = [‘gfp’; ...
                ‘dapi’; ...
                ‘neun’]
```

You can have multidimensional arrays of characters, but this doesn’t work if the strings have different lengths! All rows must have the same number of columns. You could pad with spaces to make all rows the same length, but there’s a better way…

Collections of unlike items

Or what if we wanted to store lists of spike times?

The spike times on a given trial would be an array of class **`double`**

But, we probably don’t have the same number of spikes on every trial, so we again can’t (shouldn’t) use a multidimensional array where each row is a trial and each column is a spike number.

```
spikesTrial1 = [3.2 5.8 9.1];
spikesTrial2 = [1.3 5.3 9.3 10.1];
spikesTrial3 = [0.3 2.1];
```
(please don’t ever do this)

Cell arrays

Cell arrays work similarly to numeric arrays or character arrays, except inside each element you can store whatever you want:

- Numerical arrays
- Strings (character arrays)
- Other cell arrays
- etc.

Creating cell arrays

Use the cell() function just like you would zeros() or ones(), except it returns an array of empty cells.

```
channelNames = cell(5,1);
```

`channelNames` evaluates to

```
[]
[]
[]
[]
[]
```

5 rows, 1 column

Each of these represents the empty content stored in each cell of the array.

`size(channelNames)` evaluates to `[5 1]`

Creating cell arrays

Multidimensional cell arrays work too. Say you have 5 subjects, 3 conditions. Create a cell array where each subject is a row, each condition is a column.

```
dataByCondition = cell(5,3);
```

`dataByCondition` evaluates to

```
[]  []  []
[]  []  []
[]  []  []
[]  []  []
[]  []  []
```

`size(dataByCondition)` evaluates to `[5 3]`

Cell array indexing

There are two different ways to index into a cell array. They have different uses and different syntaxes.

The most common case is when you want to extract or store something into a particular cell of the array. For this you use `{ }` (curly brackets, braces).

Cell array indexing

```
dataByCondition{1,3} = [5 10 102];
```

`dataByCondition{1,3}` evaluates to `[5 10 102]`

```
dataByCondition{3,2} = [192 10];
```

`dataByCondition` evaluates to

```
[]  []          [5 10 102]
[]  []          []
[]  [192 10]    []
[]  []          []
[]  []          []
```

Cell array indexing

```
nChannels = 3;
channelNames = cell(nChannels,1);
channelNames{1} = ‘dapi’;
channelNames{2} = ‘gfp’;
channelNames{3} = ‘neun’;
```

`channelNames` evaluates to

```
‘dapi’
‘gfp’
‘neun’
```

Cell array indexing

There are two different ways to index into a cell array.

1) The most common case: extract or store something into a particular cell

use: { } (curly brackets, braces).

2) Less common case: select part of a cell array and **keep it as a cell array, without extracting the contents**. For this you can use ( ).

Cell array indexing

dataByCondition: each subject is a row, each condition is a column.

What if we want to select all the data for a given **subject**? But keep the filtered data in a cell array.

```
dataSubj1 = dataByCondition(1,:);
```

`dataSubj1` evaluates to `[] [] [5 10 102]`

`size(dataSubj1)` evaluates to `[1 3]`

`dataSubj1{3}` evaluates to `[5 10 102]`

Note that `dataSubj1` is still a cell array, so use `{ }` to extract the contents.

Cell array indexing

dataByCondition: each subject is a row, each condition is a column.

What if we want to select all the data for a given **condition**? But keep the filtered data in a cell array.

```
dataCond2 = dataByCondition(:,2);
```

```
dataCond2    evaluates to  []

                           []

                           [192 10]

                           []

                           []
```

Try it: Cell array indexing

1. Download assignment 5 from course website
2. load `J20110809_M1.mat`
3. >> spikeTimes = reachingData(1).spikeTimes; % extract one cell array
4. What is in the cell on the third row, third column?
5. Create a variable containing the entire third column of spikeTimes

Cell array creation

Remember that you can create numeric arrays by placing a list of numbers in square brackets `[ ]` separated by spaces/commas (horizontal) or semicolons (vertical).

You can build cell arrays by listing the items inside `{}` brackets.

`channelNames = {‘dapi’, ‘gfp’, ‘neun’};`

`channelNames` evaluates to

```
            ‘dapi’ ‘gfp’ ‘neun’
```

`size(channelNames)` evaluates to `[1 3]`

Cell array creation

```
channelNames = {‘dapi’; ‘gfp’; ‘neun’};
```

`channelNames` evaluates to

```
‘dapi’
‘gfp’
‘neun’
```

`size(channelNames)` evaluates to `[3 1]`

Cell array creation

If you have two cell arrays that you wish to concatenate (join) together, enclose them in `[]` brackets, separated by a space/comma (horizontal) or semicolon (vertical).

If you accidentally enclose them in `{}`, you'll end up with the two cell arrays nested inside an outer cell array.

Cell array creation

```
channelNames = {‘dapi’; ‘gfp’; ‘neun’};
moreChannelNames = {‘vglut’; ‘syn’};
allChannels = [channelNames; moreChannelNames]
```

evaluates to

```
‘dapi’
‘gfp’
‘neun’
‘vglut’
‘syn’
```

`size(allChannels)` evaluates to `[5 1]`

Outline

1. **Control flow review**
    1. `if`, `for`, `while`, `switch`
2. **Cell arrays**
    1. Creating and concatenating cell arrays
    2. `{}` Indexing vs. `()` Indexing
3. **Struct arrays**
    1. Associating related data
    2. Structs, struct arrays, and indexing
    3. Accessing fields, aggregating data across a struct array
4. **Functions and variable scope**
5. **Debugging tools**
    1. Breakpoints
    2. Stepping through code
6. **Start Assignment 4**

Associating related data

Suppose we have many cells that we've patched, each with some **metadata** (like the date and the opsin it's expressing) alongside some **data** (spikes evoked vs. frequency).

How could we organize this in MATLAB?

Associating related data

Approach 1: use separate variables to hold each type of data.

```
nPatched = 3;

recordingDates = cell(nPatched,1);

constructName = cell(nPatched,1);

frequencyTracking = cell(nPatched,1);

for iNeuron = 1:nPatched

     frequencyTracking{iNeuron} = ...

          zeros(nFreq, 1);

end
```

Associating related data

Approach 1: separate variables for each type of data.

Pros:

- Easy to access all patched cells' data at once, i.e. get a list of dates on which the cells were recorded

Cons:

- Requires you to keep track of many different variables
- When passing this information to a utility function, you have to pass several different variables for a particular patched cell, rather than just one.
- If you want to select particular patched cells or remove some invalid ones, you have to do this for each of the variables you're using.

Associating related data

Approach 2: Use one big cell array where each column stores a particular type of information and each row stores a particular.

```
data = cell(nPatched, 3);
data{1,1} = ‘2011-07-09’;
data{1,2} = ‘ChR2’;
data{1,3} = zeros(nFreq,1);
data{2,1} = ‘2011-07-10’;
           .
           .
           .
```

Associating related data

Approach 2: Use one big cell array where each column stores a particular type of information and each row stores a particular.

Pros:

- Only one variable to keep track of
- You can filter for particular patched cells by indexing whole rows

Cons:

- You need to keep track of what each column means
- Difficult for other people to read the code

Associating related data

Approach 3: Use a structure array.

What’s a structure?

Structures

A `struct` is a collection of **fields** and their **values** that are grouped into one variable. Access fields using the ‘dot’ notation

```
patchData.date = ‘2011-07-09’;

patchData.opsin = ‘ChR2’;

patchData.freqTracking = zeros(nFreq,1);
```

`patchData` evaluates to

```
        date: '2011-07-09'
       opsin: 'ChR2'
freqTracking: [7x1 double]
```

`patchData.date` evaluates to `'2011-07-09'`

Struct arrays

You can also create an array of structs, known as a struct array. **Each struct in the array must have the same set of fields as the others**, but the values stored in each struct can be completely different.

```
patchData(1).date = ‘2011-07-09’;
patchData(1).opsin = ‘ChR2’;
patchData(1).freqTracking = zeros(nFreq,1);
patchData(2).date = ‘2011-07-10’;
patchData(2).opsin = ‘ChETA’;
patchData(2).freqTracking = zeros(nFreq,1);
```

Struct arrays

You can also create an array of structs, known as a struct array. Each struct in the array must have the same set of fields as the others, but the values stored in each struct can be completely different.

`patchData` evaluates to

```
1x2 struct array with fields:
    date
    opsin
    freqTracking
```

Struct array indexing

Indexing on struct arrays works the same as indexing on numeric arrays, except you get the whole structure (or array of structures) that you've selected.

`patchData(1)` evaluates to

```
         date: '2011-07-09'
        opsin: 'ChR2'
 freqTracking: [7x1 double]
```

`patchData(1).date` evaluates to `'2011-07-09'`

Assigning into a struct array

You can either assign into the struct array one field at a time:

```
patchData(3).date = ‘2011-08-11’;
patchData(3).opsin = ‘C1V1’;
patchData(3).freqTracking = zeros(nFreq,1);
```

Or you can build a struct with identical fields in the same order and assign it in or concatenate it.

```
newData.date = ‘2011-08-11’;
newData.opsin = ‘ChETA’;
newData.freqTracking = zeros(nFreq,1);
patchData(3) = newData;
```

Try it: Structures

1. load `J20110809_M1.mat`
2. What are the fields in this struct array?
3. how big is the array?
4. add a field to the first structure, name it whatever you want
5. what happens to the second structure in the array?

Grabbing all values in a certain field

What if we want to know the list of all the opsins we've used? Or all the dates we've patched on?

The idea is to grab the field from entire the struct array (without indexing first)

Then to "**capture**" the results in an array using `[]` or a cell array using `{}` depending on whether the contents are numeric or strings

```
opsinNamesByCell = {patchData.opsin}
```

evaluates to

```
'ChR2'
‘ChETA’
‘ChR2’
```

Try it: Struct Arrays

1. create a cell array containing each of the reachingData.targetDir values across the array. (hint: can you do it in one line)

Outline

- Control Flow Review
  - For loops, while loops
  - if/then statements
- Cell arrays
  - Creating and concatenating cell arrays
  - `{}` Indexing
  - `()` Indexing
- Structures
  - Associating related data
  - Structs, struct arrays, and indexing
  - Accessing fields, aggregating data across a struct array
- **Functions and Variable Scope**
- Debugging
  - Breakpoints
  - Stepping through code
- Assignment 3 Overview

Functions review

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

Functions are saved as .m file to disk (like scripts)

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

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

Outputs: whatever value you store in the variables you name here will be returned to the caller in the order specified

```
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 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 in the order specified, so you can access the values from your code

```
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 should be named this.m on disk, e.g. sqrtNewton.m on disk. 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

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

Documentation: tells the user how to use this function, including a quick summary and what the inputs and outputs mean. You can see this by typing **`help sqrtNewton`** 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

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

end
```

Function Example

Function code: this is what runs when you call the function. Whatever you call it with will be stored in the arguments and whatever you assign to out will be returned to the caller.

```
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

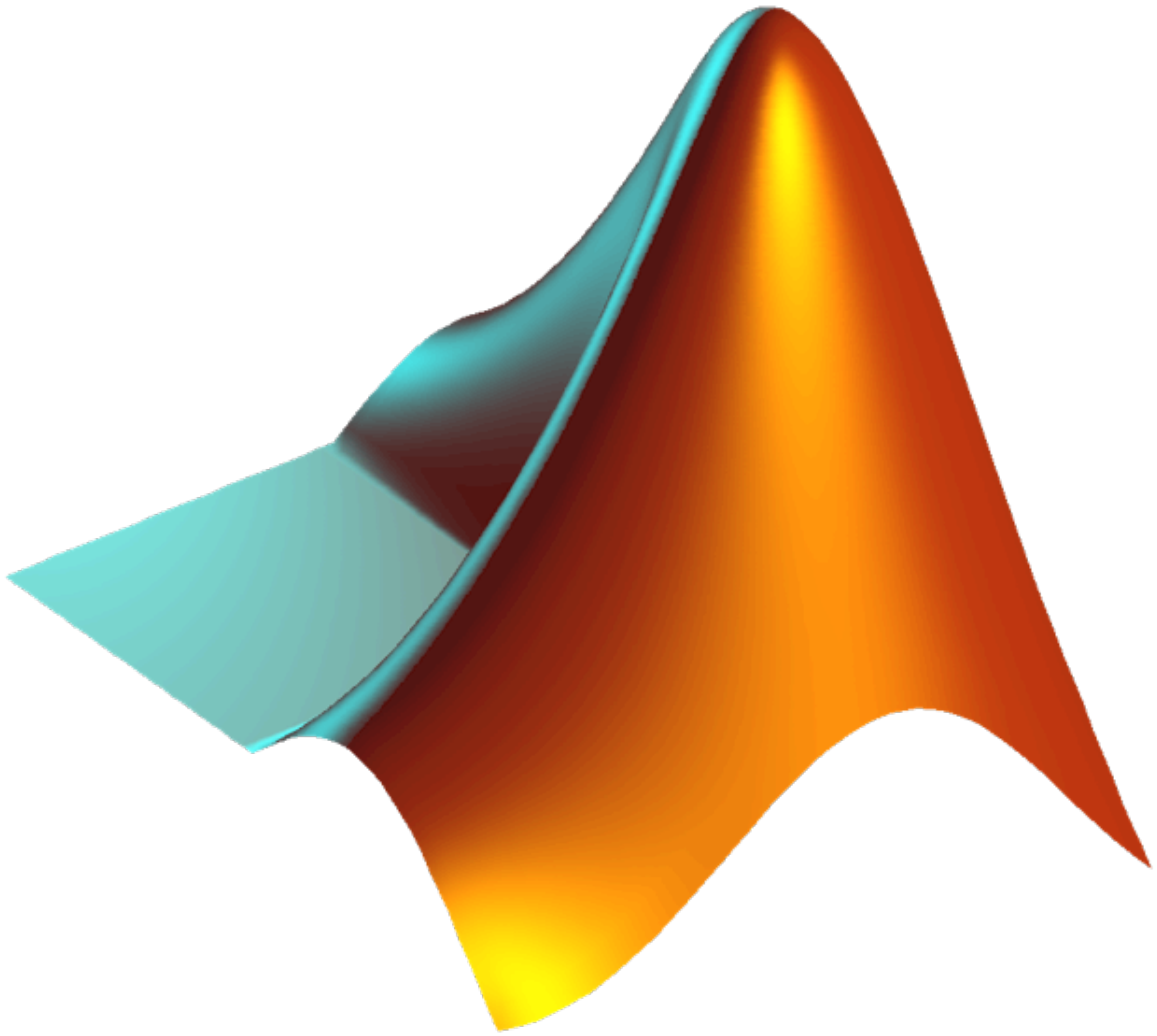
```matlab
function out = sqrtNewton(in)
% sqrtNewton Finds the square root of a number using Newton's Method
% out = sqrtNewton(in)
%
% INPUTS
% in - the number to take the square root of
%
% OUTPUTS
% out - the square root of in

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**.

- That 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

Demo: Functions and variable scope



Outline

- Control Flow Review
    - For loops, while loops
    - if/then statements
- Cell arrays
    - Creating and concatenating cell arrays
    - {} Indexing
    - ( ) Indexing
- Structures
    - Associating related data
    - Structs, struct arrays, and indexing
    - Accessing fields, aggregating data across a struct array
- Functions and Variable Scope
- **Debugging**
    - **Breakpoints**
    - **Stepping through code**
    - **printf() and other methods**
- Assignment 3 Overview

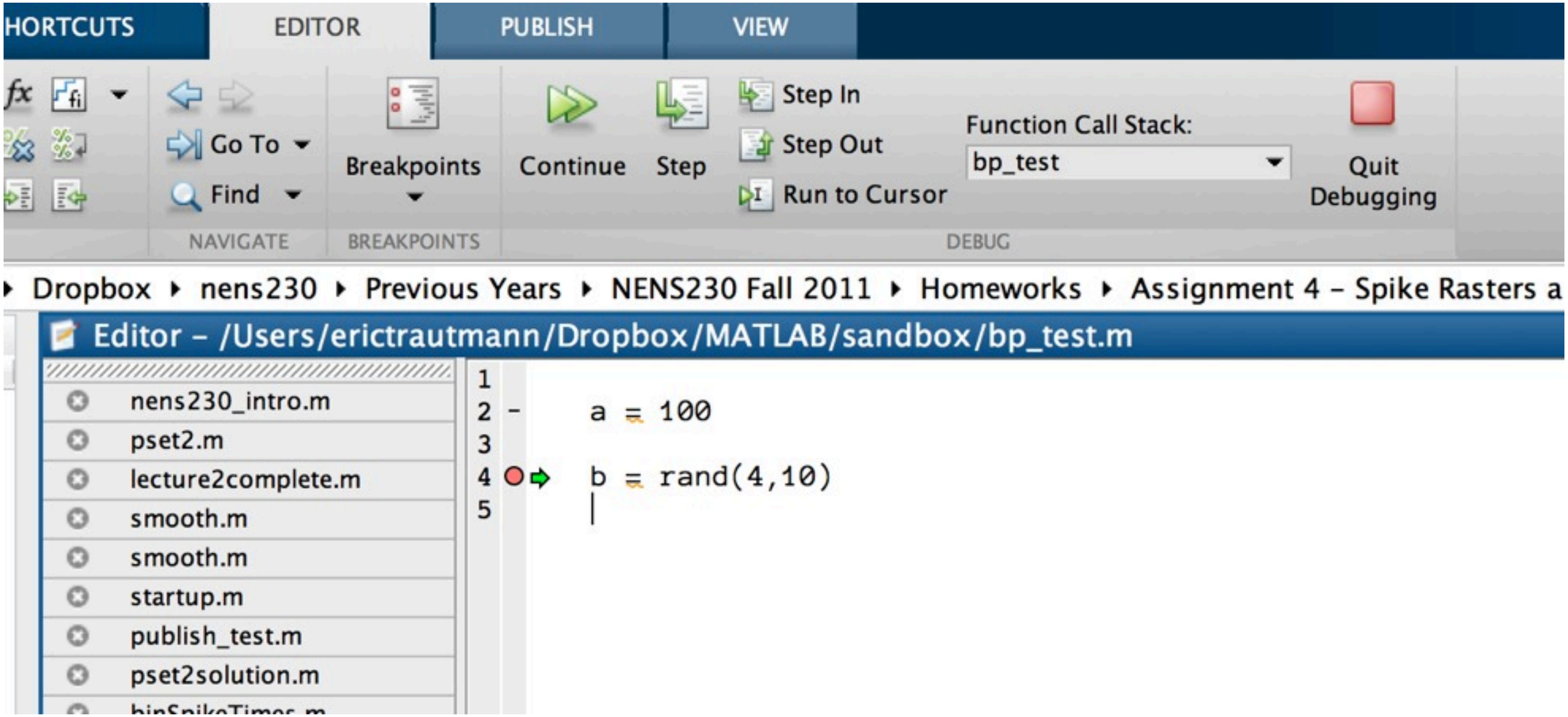
Debugging Methods

1. Breakpoints and stepping through code
2. keyboard (keyword)
3. datatips - hover w/ mouse over variables in editor.
4. printf() statements
5. cell-mode execution in scripts

Breakpoints

- Allow you to stop execution at any line of code.
- This allows you to inspect variables and interact with the program state.
- After pausing execution, can step through code line by line.
- Used very often!

Debugging Toolbar (after 2012)



Dropbox ▸ nens230 ▸ Previous Years ▸ NENS230 Fall 2011 ▸ Homeworks ▸ Assignment 4 – Spike Rasters a

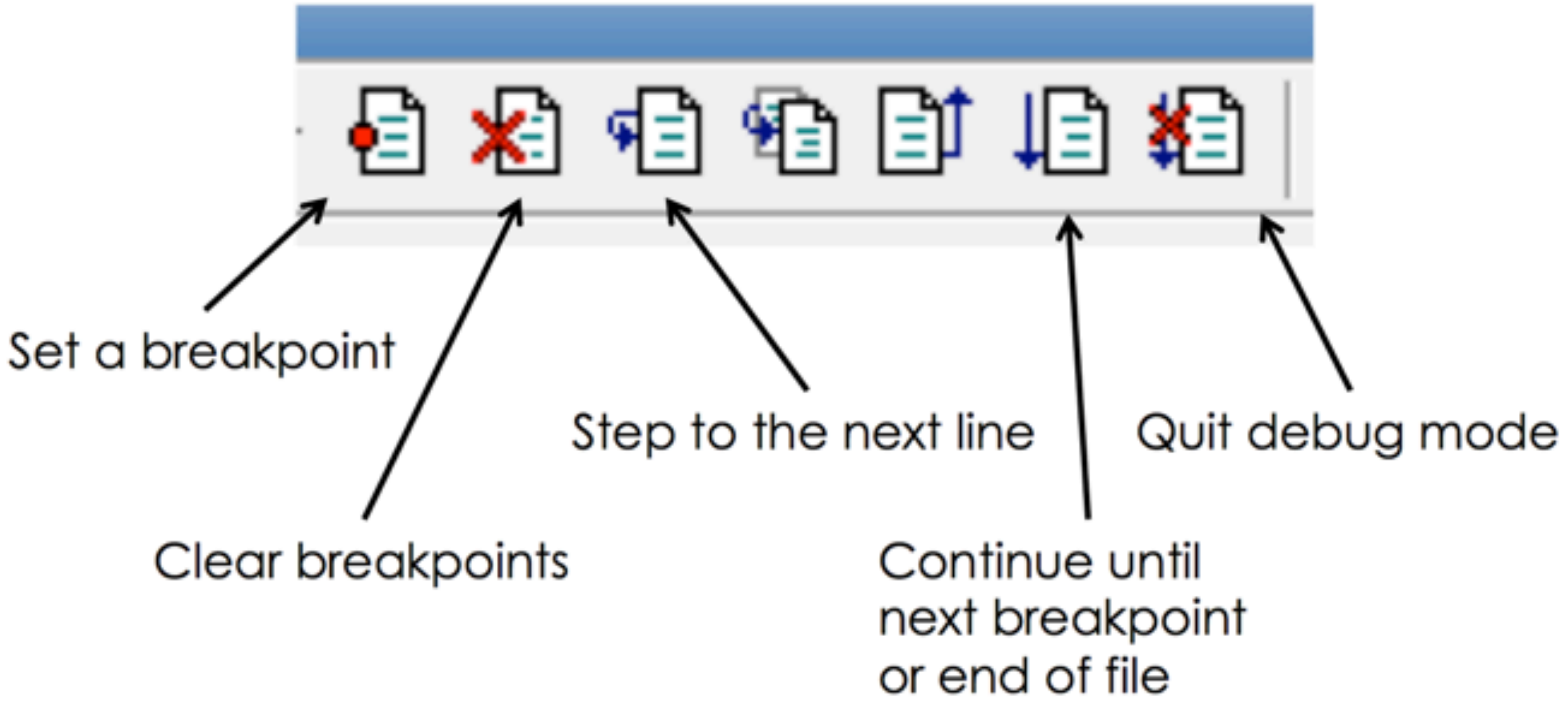
Editor – /Users/erictrautmann/Dropbox/MATLAB/sandbox/bp_test.m

nens230_intro.m
pset2.m
lecture2complete.m
smooth.m
smooth.m
startup.m
publish_test.m
pset2solution.m

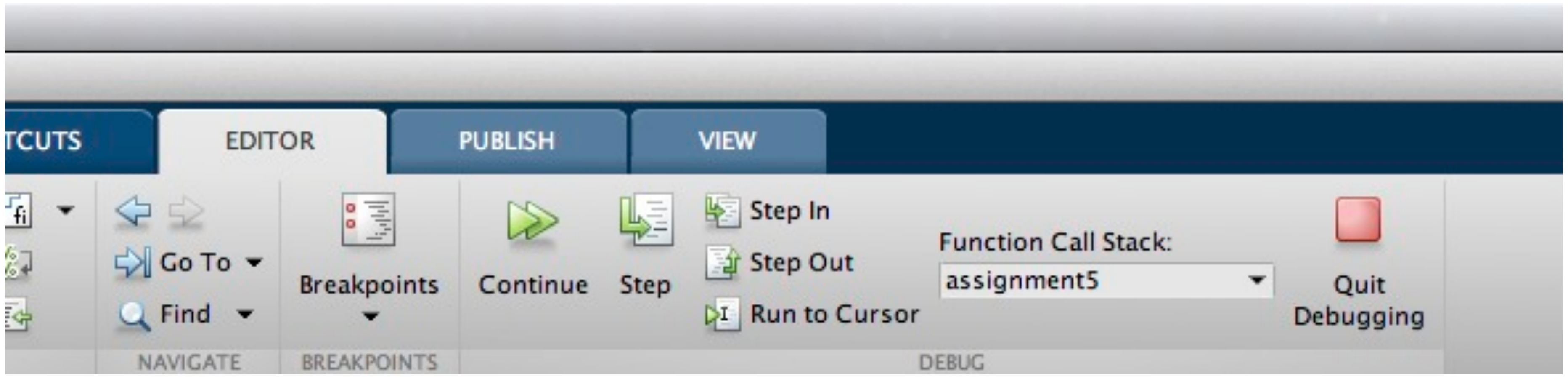
```

a = 100

b = rand(4,10)

```

Debugging Toolbar

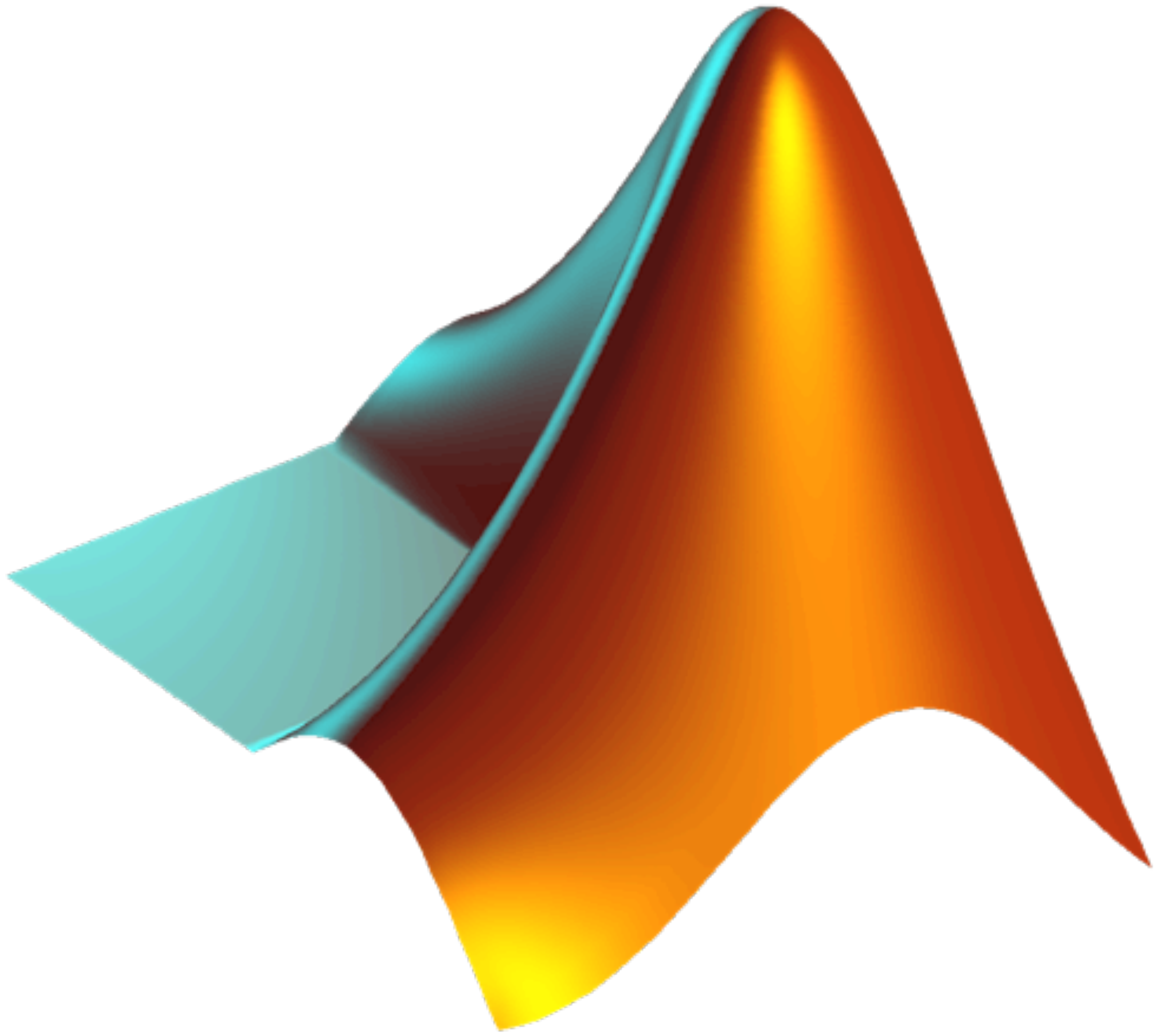
(pre 2012)



(post 2012)



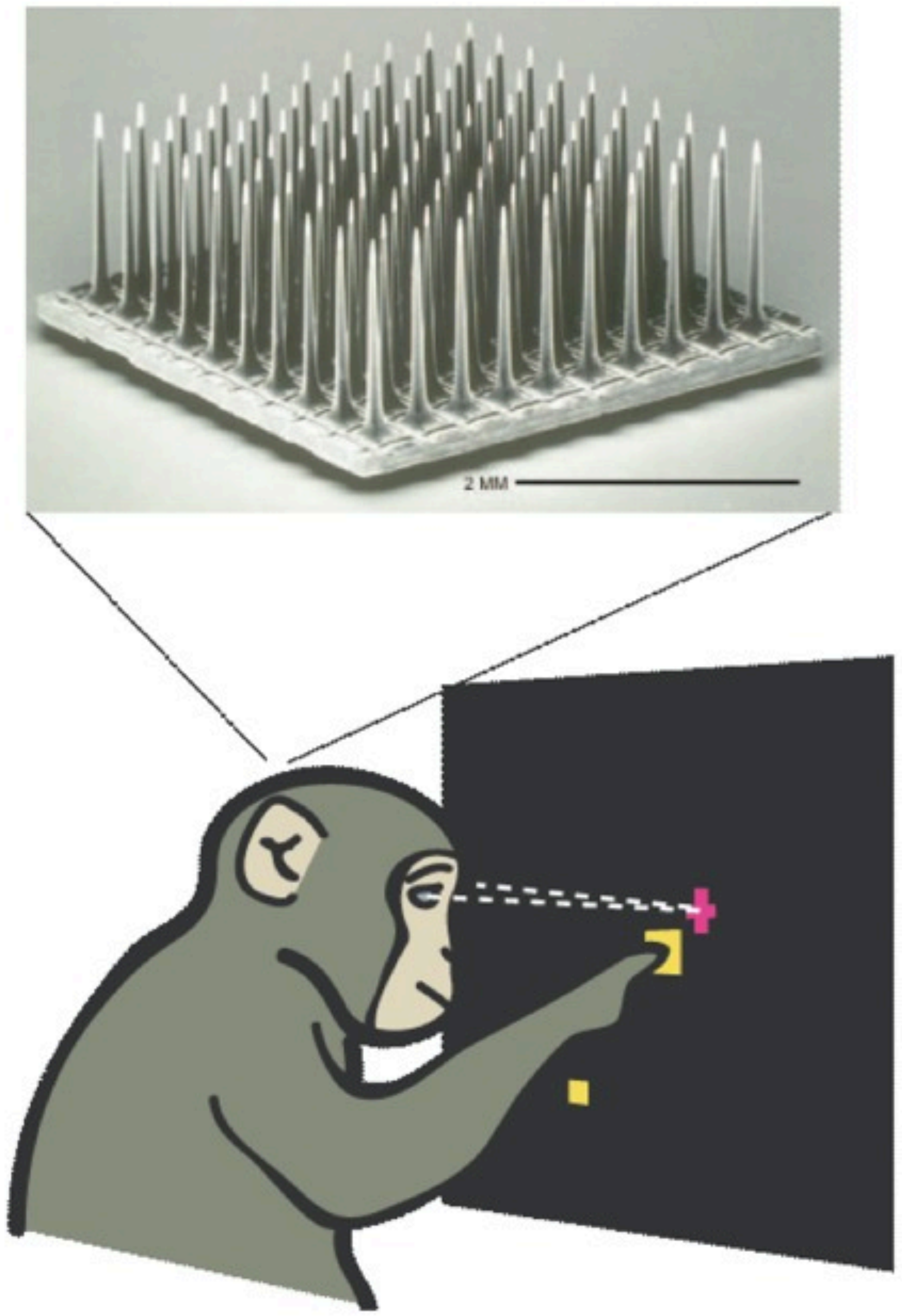
Demo: Breakpoints and Stepping



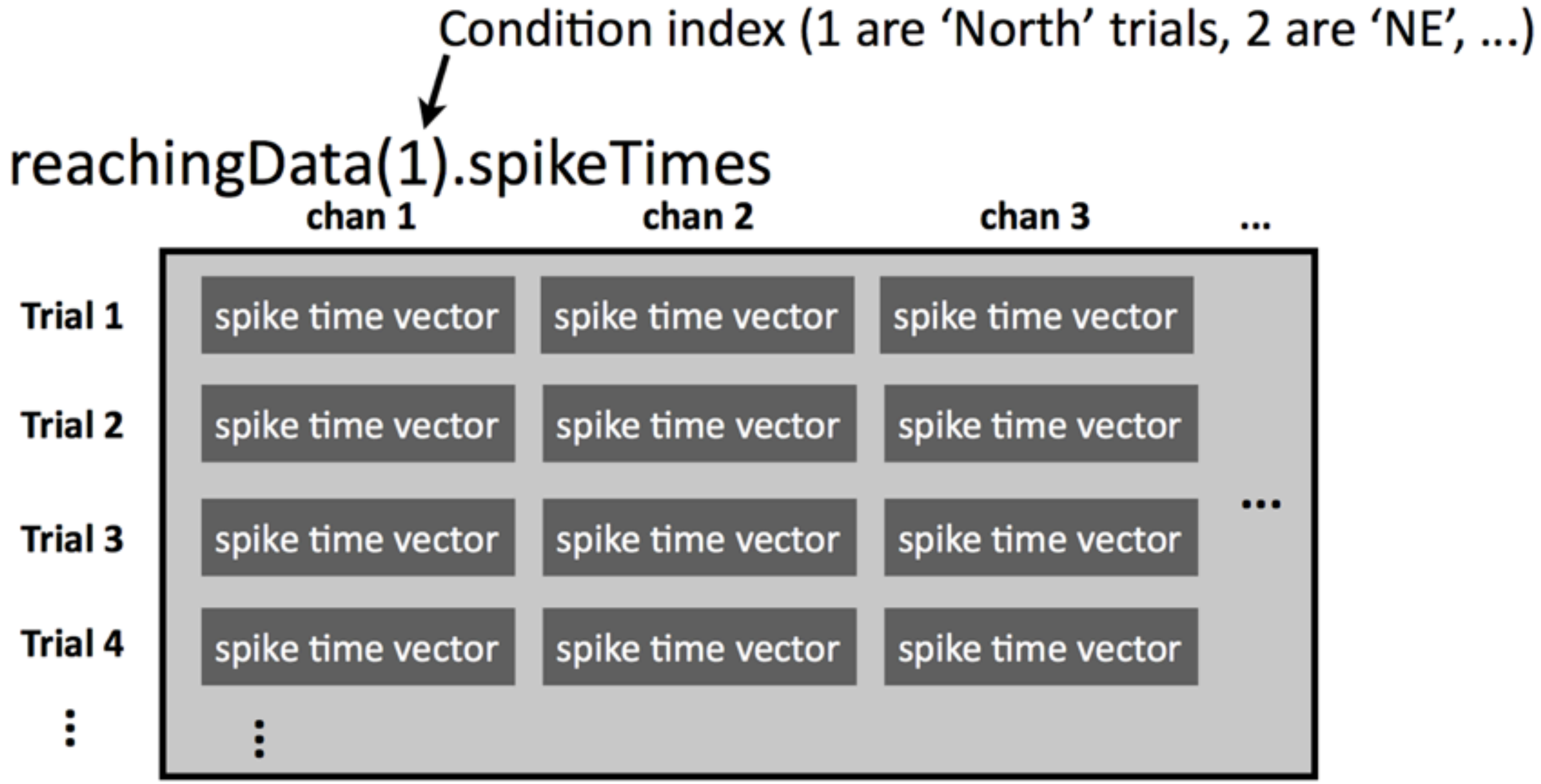
`Keyboard` keyword

- Similar to a single breakpoint.
- Pauses execution of function and allows user to interact with variables in current scope via command line.
- continue execution with **`return`** keyword
- use **`dbquit`** to stop if stuck (ie: used **`keyboard`** inside loop).

Assignment Overview



Assignment Overview


Condition index (1 are 'North' trials, 2 are 'NE', ...)
reachingData(1).spikeTimes
chan 1
chan 2
chan 3
...
Trial 1
Trial 2
Trial 3
Trial 4
spike time vector
...