Lecture 07: Best Practices

NENS 230: Analysis Techniques in Neuroscience

Lecture 07 Outline

Questions?

Code Style

Flexible Functions

Measuring Performance

Improving Efficiency

Assignment 7 Overview

Lecture 07 Outline

Questions?

Code Style

Flexible Functions

Measuring Performance

Improving Efficiency

Assignment 7 Overview

Lecture 07 Outline

Questions?

Code Style

Flexible Functions

Measuring Performance

Improving Efficiency

Assignment 7 Overview

Code Style

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

It is best practice to have good "**coding style**" -- writing neat and clear code that is easy to read and understand

No "right way", but today we're presenting some useful conventions that you may wish to adopt

People tend to adopt some conventions (from courses, mentors, collaborators) and also make up their own

Naming functions

Give descriptive names to functions:

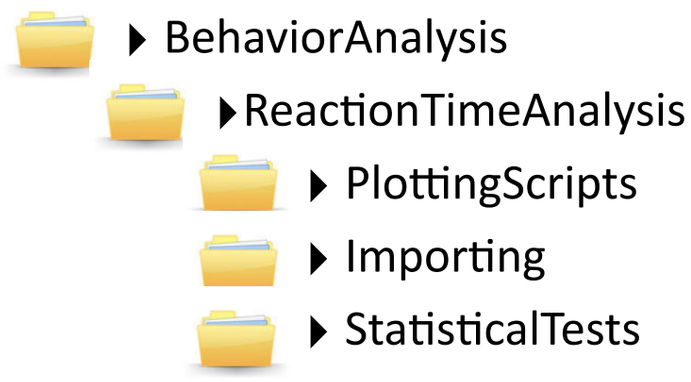
e.g. `binSpikeTimes`, `drawReactionTimesHistogram`, `loadAbf`

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. `GenerateReachingFigure.m` or `GENERATE_REACHING_FIGURE.m`, which has helper functions `addShadedEpoch.m`, `binSpikeTimes.m`, `drawRasters.m`, 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 `My Questionable(?) Data Directory`

Naming variables

Similarly, give descriptive names to variables. Use capital letters to denote start of word

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 = 30000; BOLTZMANN = 1.38e-23;`

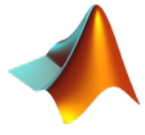
Some people come up with a convention for identifying types of variables, e.g. `sParams` for structures, `goodTrialsIdx` for indices, or `keepGoingBool` for boolean/logical

Indenting and Spacing

Put one blank line between steps within the same task, and more to mark a larger change of what you're doing

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

- Nested blocks are indented one level deeper, analogous to indented bulleted list in a report
- Highlight code and Command-i (Mac/Linux) or Ctrl-i (Windows) to automatically do indenting



indentExample.m

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 )` **versus** `myVariable(colIdx, rowIdx)`

Comments

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;
```

Function Header

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

```
% functionName.m
%
% 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 November 2011
function [outvar1 outvar2] = functionName( arg1, arg2 )
    % CODE GOES HERE
end
```

Lecture 07 Outline

Questions?

Code Style

Flexible Functions

Measuring Performance

Improving Efficiency

Assignment 7 Overview

Multiple Output Arguments

A function can have **multiple outputs**, which we can get 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 or it will not work

If you don't care about some of the outputs when calling a function, put a tilde in their place:

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

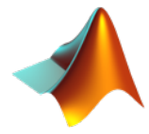
A more advanced feature is to define a variable number of output arguments using **varargout**:

```
[defined1, defined2, varargout] = flexibleFunction( )
```

Inside the function, you can define these extra output arguments using

```
varargout{i} = something;
```

`nargout` tells you how many output arguments the function was called with



varargoutDemo.m

Multiple Input Arguments

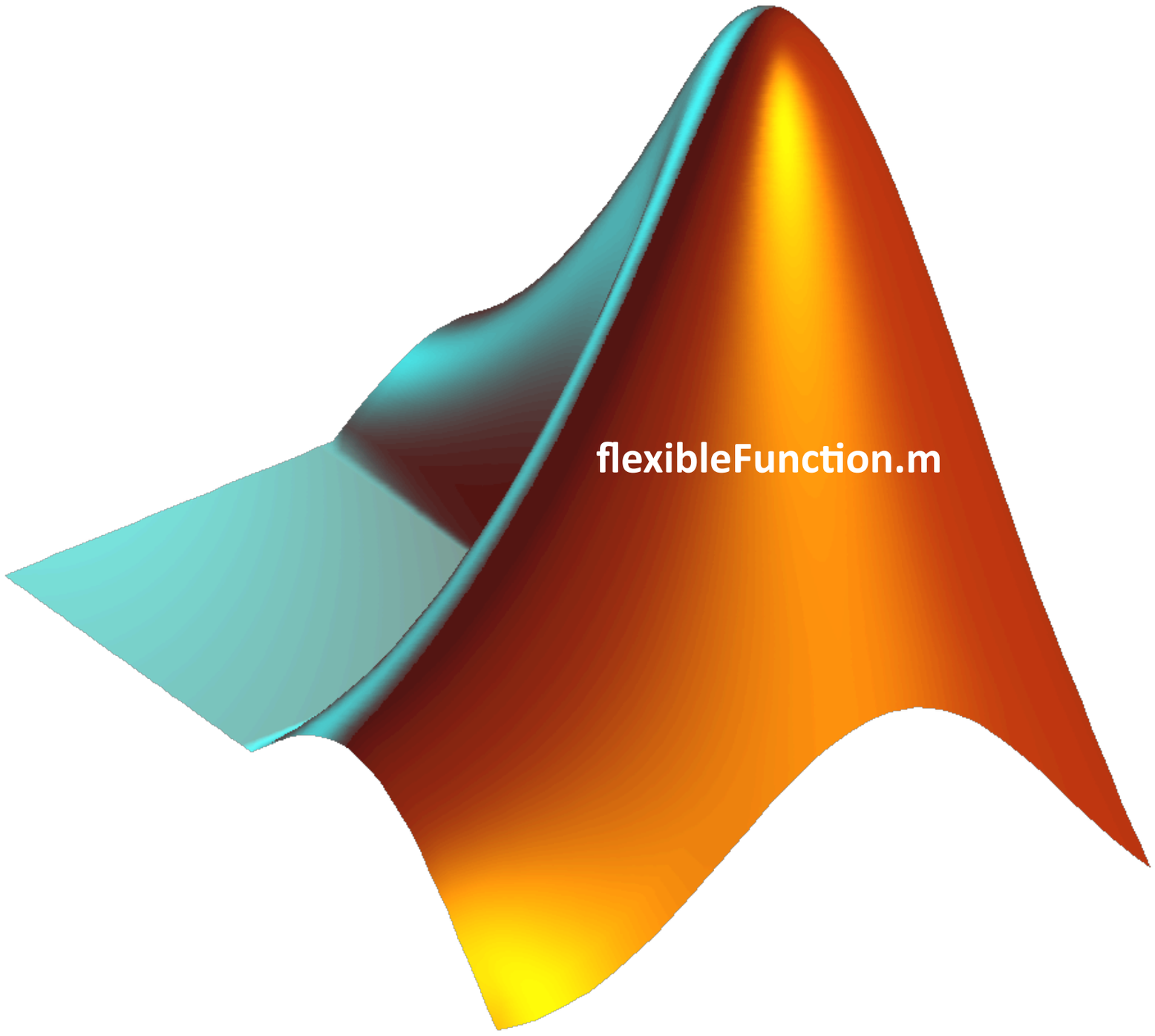
You've already been using functions with **multiple input arguments**:

```
[out] = myFunction( arg1, arg2, arg3 )
```

When calling a function, you do not need to provide all input arguments

This is useful for making **optional arguments** which, if provided, alter function behavior

Demo 1: Flexible Function



Multiple Input Arguments

You've already been using functions with **multiple input arguments**:

```
[out] = myFunction( arg1, arg2, arg3 )
```

When calling a function, you do not need to provide all input arguments

This is useful for making **optional arguments** which, if provided, alter function behavior

If you want to provide e.g. arg1 and arg3 but not arg2, you must still enter something for arg2:

```
[out] = myFunction( 'a', [], 'c' )
```

Error is caused when an input argument is not provided but you try to use it in the function

Avoid this by checking the number of inputs provided to the function using `nargin`

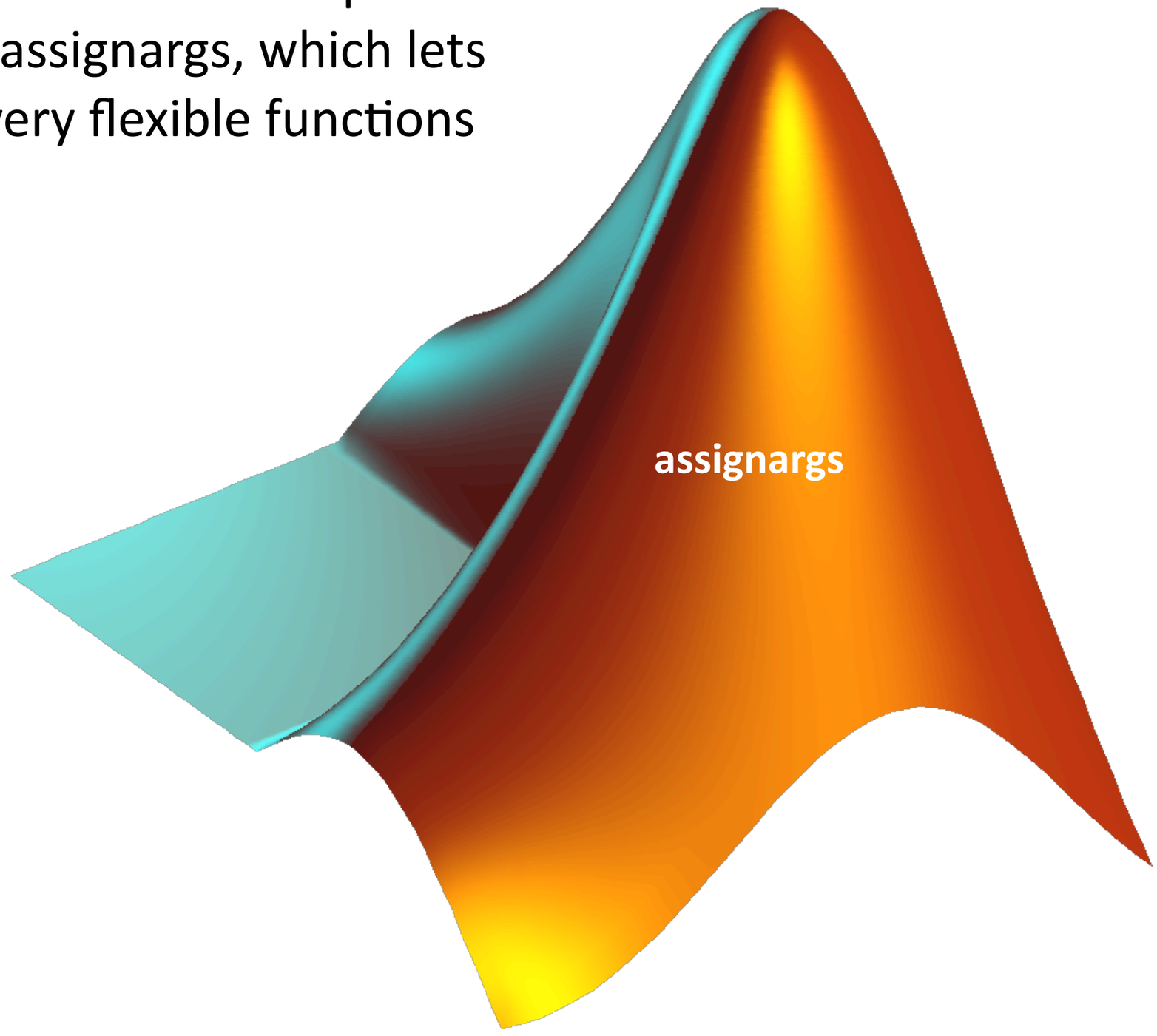
In addition to (or in lieu of) defined inputs, can have **variable number of input arguments** using `varargin`

Another good trick is to pass in a structure with multiple fields, e.g. params.xlim, params.color, ...

You can check if a certain field exists using `isfield( structure, 'fieldName' )`

Demo 2: assignargs

Dan O'Shea has created a powerful tool called assignargs, which lets you write very flexible functions



Functions and Modularity

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

If you'll do something again in your program, make it its own function

Make functions more flexible by adding optional arguments rather than creating a very similar function

A good rule of thumb is to separate data importing, data processing, and data visualization into their own functions

Lecture 07 Outline

Questions?

Code Style

Flexible Functions

Measuring Performance

Improving Efficiency

Assignment 7 Overview

Performance

Clarity, modularity, and flexibility are usually the most important qualities of code

We sometimes also care about **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 processors and memory chips, performance typically only matters when working with large datasets or doing computationally complex analyses

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

Measuring Memory Use

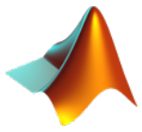
Variables (and objects) that exist consume **memory** (also called “RAM”; this is not the same as disk space)

How much total memory MATLAB has available depends on your computer hardware, whether your operating system is 32- or 64-bit, and what else is running

Typically, MATLAB will have access to 1-3GB of memory on a standard laptop, and as much as 15-60GB on high-end computers

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`



whosDemo.m

Measuring Memory Use With whos

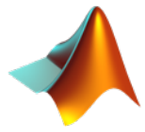
Variables (and objects) that exist consume **memory** (also called “RAM”; this is not the same as disk space)

How much total memory MATLAB has available depends on your computer hardware, whether your operating system is 32- or 64-bit, and what else is running

Typically, MATLAB will have access to 1-3GB of memory on a standard laptop, and as much as 15-60GB on high-end computers

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`



whosDemo.m

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

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 to measure how long various parts of it take

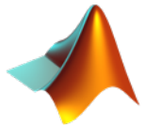
You can measure how fast a group of commands is executed by a "stopwatch"-like tool:

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 );
```



timingExample.m

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 **Tools --> Open Profiler**

Enter the function or script you want to run in the box, then press “Start Profiling”

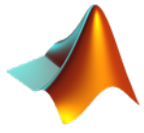
Calls is how many times a function was called

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 07 Outline

Questions?

Code Style

Flexible Functions

Measuring Performance

Improving Efficiency

Assignment 7 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:

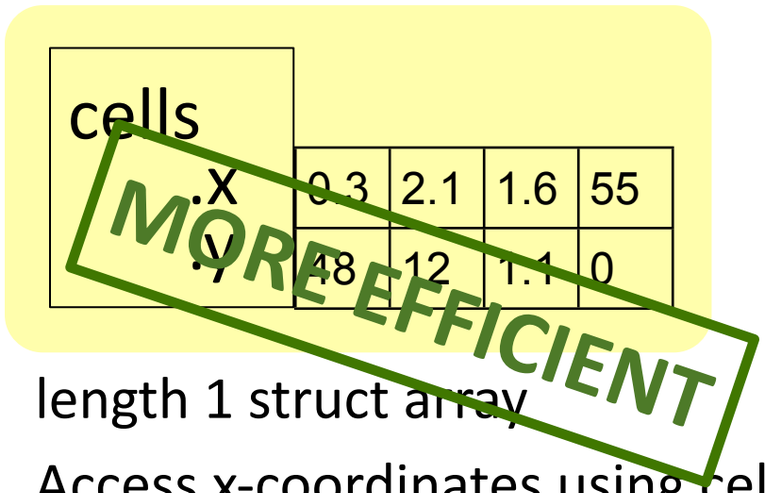
```
% Raw data comes in several large data Cerebus .ns5 files
spikeTimes = []; % will have list of all my spike times across recording
for iFile = 1 : numRawFiles
    thisRawDat = load( fileList{iFile} );
    spikeTimes = [ spikeTimes extractSpikeTimes( thisRawDat ) ];
    clear( 'thisRawDat' ) % Don't 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(1ⁱ);	logical	1 Byte [why not 1 *bit*? It's a MATLAB peculiarity...]

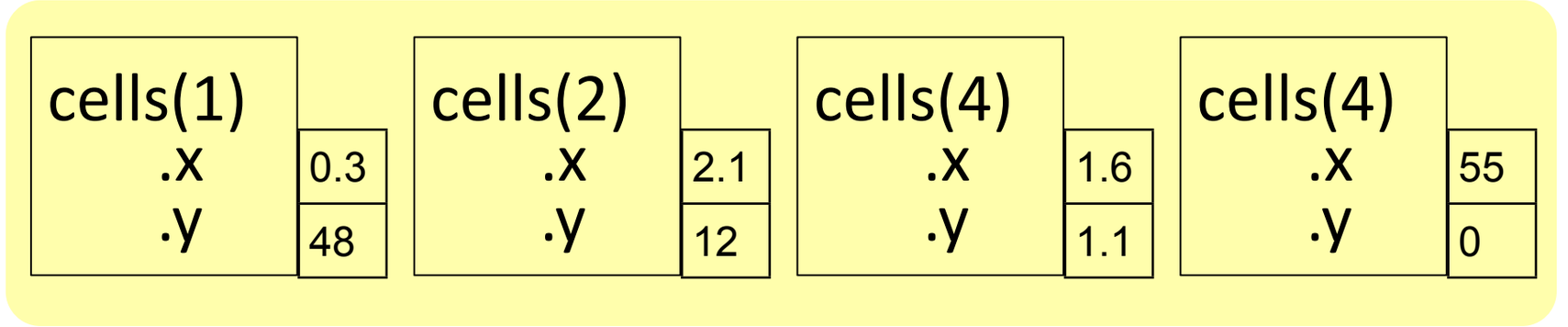
Improving Memory Efficiency (continued)

When using structures, it is more memory efficient to have a fewer top-level structure elements:



length 1 struct array

Access x-coordinates using cells.x(i)



length 4 struct array

Access x-coordinates using cells(i).x

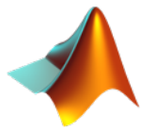
This is because there is memory overhead to define many elements structure array, each containing multiple fields

structureOrgExample.mat

Speeding Up Computation - 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

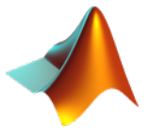
Can preallocate with any value; typically the zeros( ) command is used



fasterCodeDemo.m

Speeding Up Computation - Less Display Output

Display outputs such as `fprintf`, `display`, or unsupressed output (no semicolon after assignment) burn a lot of time



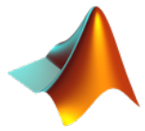
fasterCodeDemo.m

Speeding Up Computation - Vectorization

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

MATLAB has a very fast underlying linear algebra library

Performing one operation on a matrix 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):

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

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

2 horizontal repetitions

3 vertical repetitions

B

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

The function **reshape** is useful; it lets you reshape the matrix to have different numbers of rows/cols while preserving the same elements

fasterCodeDemo.m

Vectorizing Operations on Cell Arrays

The function **cellfun lets you apply the same operation to each element of a cell array**

First, let's learn about **anonymous functions:**

These let you create a temporary function inside your code and give it a handle:

```
functionName = @(arg1, arg2, ...) = (     expression   )
```

e.g. `squareExpansion = @(x,y) (x*x + 2*x*y + y*y)`

`squareExpansion(2,3)` gives 25

To use cellfun:

```
myOutput = cellfun( @functionName, InputCellArray)
```

Can be more sophisticated and take a function with multiple inputs and also multiple cell arrays

repmat works on cell arrays too
(this turns out to be useful when constructing cell arrays as arguments for multivariate cellfun)

cellfun is not always faster than a for loop, but makes for shorter, often more readable code

fasterCodeDemo.m

Lecture 07 Outline

Questions?

Code Style

Flexible Functions

Measuring Performance

Improving Efficiency

Assignment 7 Overview

Assignment Seven: Improving Bad Code

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

- Poorly documented
- Bad coding style
- Doesn't use functions for modularity
- Inefficient

Your job will be to edit it to make it both more readable, and also run faster (as determined by tic and toc)

Because it's poorly written, you will have to spend some time figuring out what various parts of the program do. Learning to figure out other people's code is an important skill.

- Use breakpoints and the step debug functions to help you figure out what's going on

Lecture 07 Review

Key Concepts

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 space between lines helps separate conceptually related lines
Code should be **documented** with **function headers** and descriptive **comments**
varargout allows you to have a **variable number of function outputs**
varargin allows you to have a **variable number of function inputs**
nargout and **nargin** can be called inside a function to report the caller's expectations
eval lets you build a command as a string and then evaluate it
varargin combined with eval is very powerful; e.g. you can define **property-value pairs**
Passing in a **structure argument** lets you easily add more fields to the input
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 is helos vectorizing many operations
anonymous functions let you create a named function on the fly
cellfun applies a function to every element of a cell array

Functions

varargout
nargout
varargin
nargin
eval, evalin
isfield
whos
tic
toc
repmat
reshape
cellfun