

Data Organization, Control Flow, and File Importing

NENS 230: Analysis Techniques in Neuroscience

Outline

Strings

- Comparing and concatenating strings
- String formatting, `fprintf()`, `sprintf()`
- Working with filenames

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

Control Flow

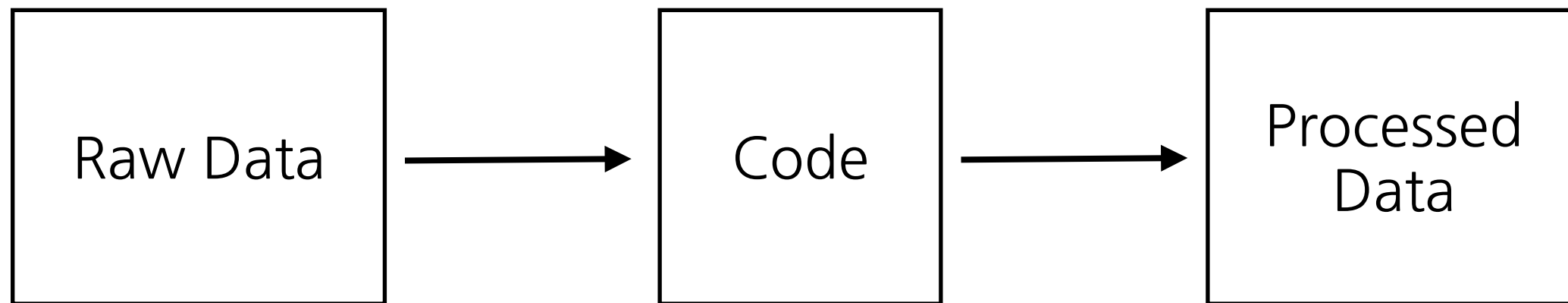
- Branching
- Loops
- Functions and variable scope

File importing

- `csvread`, `dlmread`, `xlsread`
- `textscan` and file handles

Assignment 3 Overview

Why are data types so important?



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.

We'll get to these next week...

Outline

Strings

- Comparing and concatenating strings
- String formatting, `fprintf()`, `sprintf()`
- Working with filenames

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

Control Flow

- Branching
- Loops
- Functions and variable scope

File importing

- `csvread`, `dlmread`, `xlsread`
- `textscan` and file handles

Assignment 3 Overview

Strings

An array of characters as opposed to numbers

Start and end with single quotes (apostrophe).

```
opsinName = 'ChR2';
```

```
opsinName(1)    evaluates to    'C'
```

```
opsinName(4)    evaluates to    '2'
```

```
length(opsinName) evaluates to    4
```

Comparing strings

What happens if we just use the `==` operator?

Compares the two arrays element-wise

```
channelName = 'gfp';
```

```
channelName == 'gfp'    evaluates to  [1 1 1]  
                                (logical)
```

```
channelName == 'dapi'    Error using ==> eq  
                        Matrix dimensions must  
                        agree.
```

strcmp() function

Use strcmp to test whether two strings are equal.

```
channelName1 = 'gfp';
```

```
channelName2 = 'dapi';
```

```
strcmp(channelName1, 'gfp') evaluates to 1
```

(logical)

```
strcmp(channelName2, 'gfp') evaluates to 0
```

(logical)

Concatenating strings

You can concatenate or join together strings:

- Like you would concatenate a numeric array, by wrapping them in [] brackets separated by a comma or space

```
prefix = 'data';
```

```
dayName = '20110909';
```

```
fullName = [prefix dayName]
```

evaluates to `'data20110909';`

Concatenating strings

You can concatenate or join together strings:

- Like you would concatenate a numeric array, by wrapping them in `[]` brackets separated by a comma or space
- Using the `strcat()` function

```
strcat(string1, string2, ...)
```

```
fullName = strcat(prefix, dayName)
```

evaluates to `'data20110909'`;

Concatenating strings

Be careful with combining strings with numbers.
Use the function `num2str()` to convert numbers to characters before building a string.

```
prefix = 'data';
```

```
year = 2011; month = 9; day = 19;
```

Use ellipses to
continue code on
the next line!

```
fullName = [prefix num2str(year) ...  
            num2str(month) num2str(day)]
```

evaluates to `'data20110909'`;

num2str() and str2num()

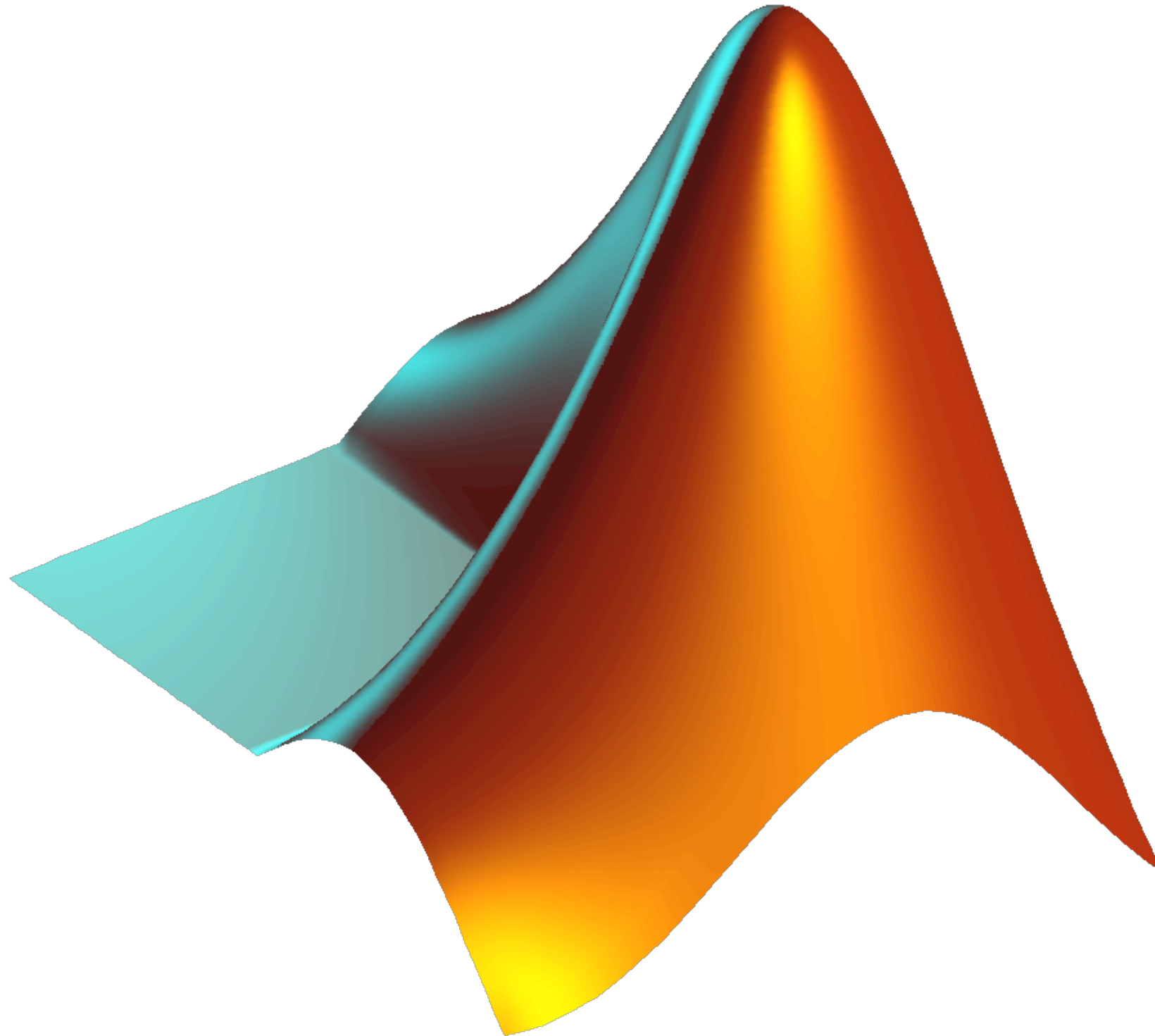
`num2str()` function converts a numeric type (e.g. `double`) into a string representation of that number

`num2str(21)` evaluates to `'21'`

`str2num()` function converts a string representation of a number into a `double`

`str2num('21')` evaluates to `21`

Demo: Strings



Other string functions

There are a number of other useful functions for dealing with strings:

- String formatting: `sprintf`, `fprintf`
- String parsing: `sscanf`, `fscanf`
- Date formatting: `datestr(now, ...)`
- File names: `fileparts`, `fullfile`, `filesep`

String formatting

`fprintf` comes from a long legacy of `printf`-like functions popularized by C and Lisp.

The basic idea is to generate a string by piecing together the values stored in a set of variables.

First, you specify a **format string** which instructs `fprintf` how exactly to build the string.

Then, you pass all of the variables that are to be included in that string.

Super quick intro to `fprintf()`

```
fprintf(formatString, var1, var2, ...)
```

What does `formatString` look like?

Anything you want. In the places where you'd like the values of variables to be inserted, you write a percentage sign and a letter signifying what type of value you want to insert

Integers and `fprintf()`

```
fprintf( 'Count: %d', currentCount);
```

```
prints    Count: 5
```

`%d` means insert the next variable in the list (here it means the first) **as an integer**.

Note that `currentCount` can still be of class `double`, but it should probably be a integer value or you should be using the `round()` function.

Integers and `fprintf()`

```
fprintf( 'Count: %d\n', currentCount);
```

prints Count: 5

`%d` means insert the next variable in the list (here it means the first) **as an integer**.

`\n` means go the next line. Tack this on at the end whenever you don't want subsequent output to continue on the same line

Floating points and `fprintf()`

```
fprintf( 'Mean: %f\n', goldenMean);
```

```
prints    Mean: 1.61803399
```

`%f` means insert the next variable in the list (here it means the first) **as an floating point number**.

`\n` means go the next line. Tack this on at the end whenever you don't want subsequent output to continue on the same line

Strings and `fprintf()`

```
fprintf( 'Monkey Name: %s\n', ...  
        subjectName);
```

```
prints   Monkey Name: Quincy
```

`%s` means insert the next variable in the list (here it means the first) **as an string**.

`\n` means go the next line. Tack this on at the end whenever you don't want subsequent output to continue on the same line

Err..wait...

Couldn't I have just done this:

```
disp(['Count: ' num2str(count)])
```

```
disp(['Golden Mean: ' num2str(goldenMean)])
```

```
disp(['Monkey Name: ' subjectName])
```

Sure. The real power of `fprintf` lies in **formatting** the values of variables to look pretty...

Field width in `fprintf()`

```
fprintf( 'Count: %6d\n', currentCount )  
fprintf( 'Count: %6d\n', 4*currentCount )
```

```
prints      Count:      5
```

```
            Count:     20
```

(useful for nice alignment)

`%d` means insert the next variable in the list (here it means the first) **as an integer**.

`%6d` means insert as an integer and **pad** the value to be 6 characters wide. That is, **insert spaces** to the left of it until it takes up 6 characters width.

Field width in `fprintf()`

```
fprintf( 'Count: %06d\n', currentCount)
```

```
fprintf( 'Count: %06d\n', 4*currentCount)
```

```
prints    Count: 000005
```

```
          Count: 000020
```

(useful for nice alignment)

`%d` means insert the next variable in the list (here it means the first) **as an integer**.

`%06d` means insert as an integer and **pad** the value to be 6 characters wide. That is, **insert zeros** to the left of it until it takes up 6 characters width.

Field precision in `fprintf()`

```
fprintf( 'Mean: %.3f\n', goldenMean)
```

```
prints    Mean: 1.618
```

`%f` means insert the next variable in the list (here it means the first) **as an floating point number**.

`%.3d` means insert as a floating point number and round the value to **3 decimal places**.

Field precision in `fprintf()`

```
fprintf( 'Mean: %6.3f\n', goldenMean)
```

```
fprintf( 'Mean: %6.3f\n', 10*goldenMean)
```

```
prints    Mean:    1.618
```

```
          Mean: 16.180
```

`%f` means insert the next variable in the list (here it means the first) **as an floating point number**.

`%6.3d` means insert as a floating point number and round the value to **3 decimal places**, then pad it with spaces to make it **6 characters wide**

Multiple variables in `fprintf()`

```
fprintf('Opsin %s: %3d spikes\n', ...  
       opsinName, nSpikesEvoked);
```

```
prints    Opsin ChR2:  40 spikes
```

Just use multiple % markers and then be sure to include the same number of arguments

Very similar: `sprintf()`

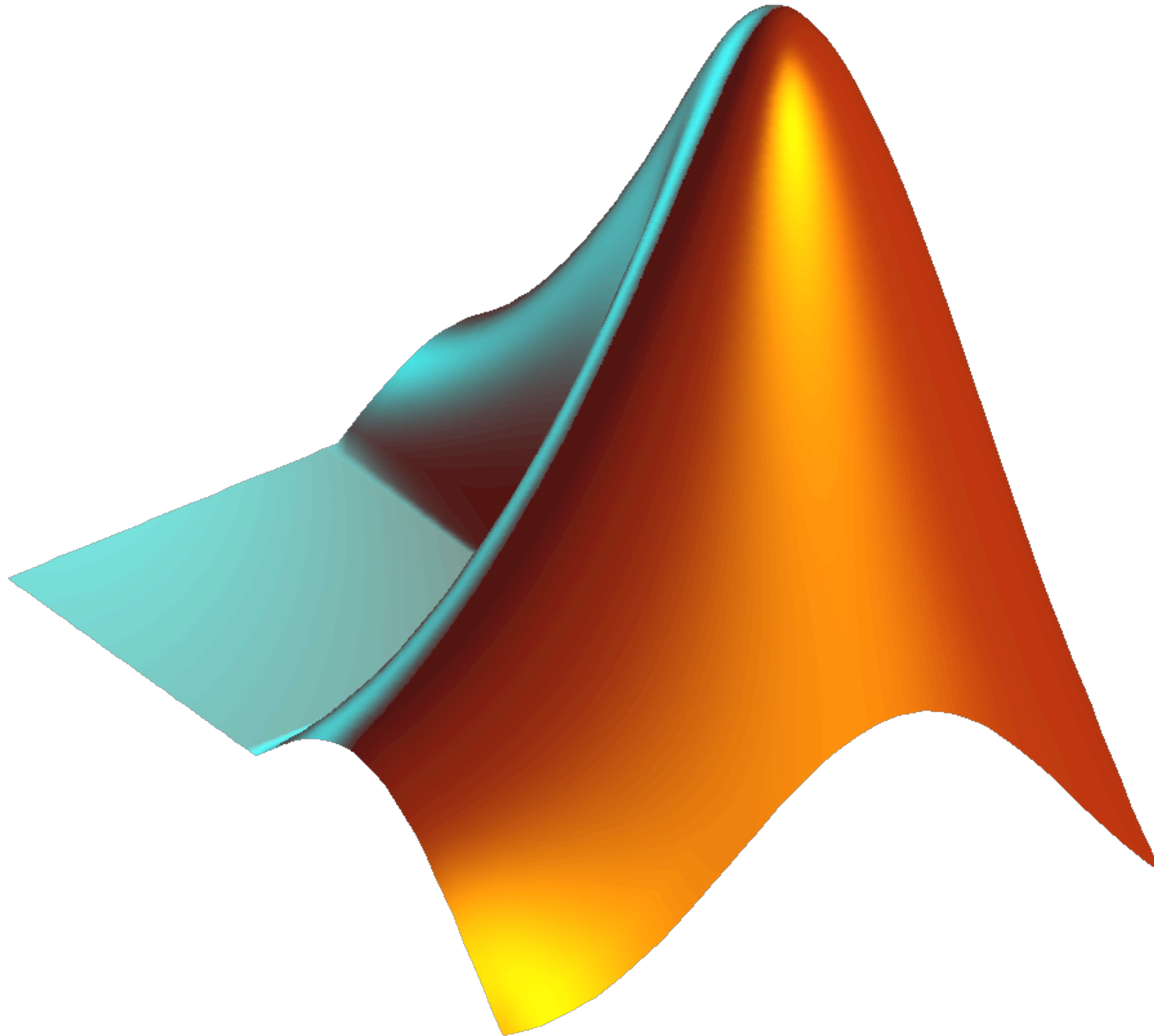
`fprintf()` builds a string and then prints it to the command window.

`sprintf()` builds a string and then returns that string. The arguments you pass to it are the same.

```
imgName = sprintf( '%s_sweep%02d.png', ...  
    opsinName, iSweep );
```

`imgName` evaluates to `ChR2_sweep01.png`

Demo: String formatting



Strings and filenames

There are a few useful functions for dealing with filenames. The main benefit of using them is to maintain platform independence (e.g. Windows uses \ to separate path tokens, UNIX/Linux/Mac use /).

fullfile() function

Concatenates two parts of a path and intelligently deals with slashes.

```
fullfile('/Users/djoshea/NENS 230', 'file.abf')
```

evaluates to `'/Users/djoshea/NENS 230/file.abf'`

```
fullfile('/Users/djoshea', 'NENS 230', 'file.abf')
```

evaluates to `'/Users/djoshea/NENS 230/file.abf'`

```
fullfile('C:\Documents\', 'NENS 230', 'file.abf')
```

evaluates to `'C:\Documents\NENS 230\file.abf'`

fileparts() function

Splits a file name into the containing folder path, the filename sans extension, and the extension.

```
fname = '/Users/djoshea/NENS 230/recording.abf';  
[parent name ext] = fileparts(fname);
```

parent evaluates to '/Users/djoshea/NENS 230'

name evaluates to 'recording'

ext evaluates to '.abf'

fileparts() function

Splits a file name into the containing folder path, the filename sans extension, and the extension.

Also useful for navigating up one folder.

```
pathToData = '/Users/djoshea/NENS 230/Data';
```

```
parent = fileparts(pathToData)
```

evaluates to `'/Users/djoshea/NENS 230'`

Outline

Strings

- Comparing and concatenating strings
- String formatting, `fprintf()`, `sprintf()`
- Working with filenames

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

Control Flow

- Branching
- Loops
- Functions and variable scope

File importing

- `csvread`, `dlmread`, `xlsread`
- `textscan` and file handles

Assignment 3 Overview

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);
```

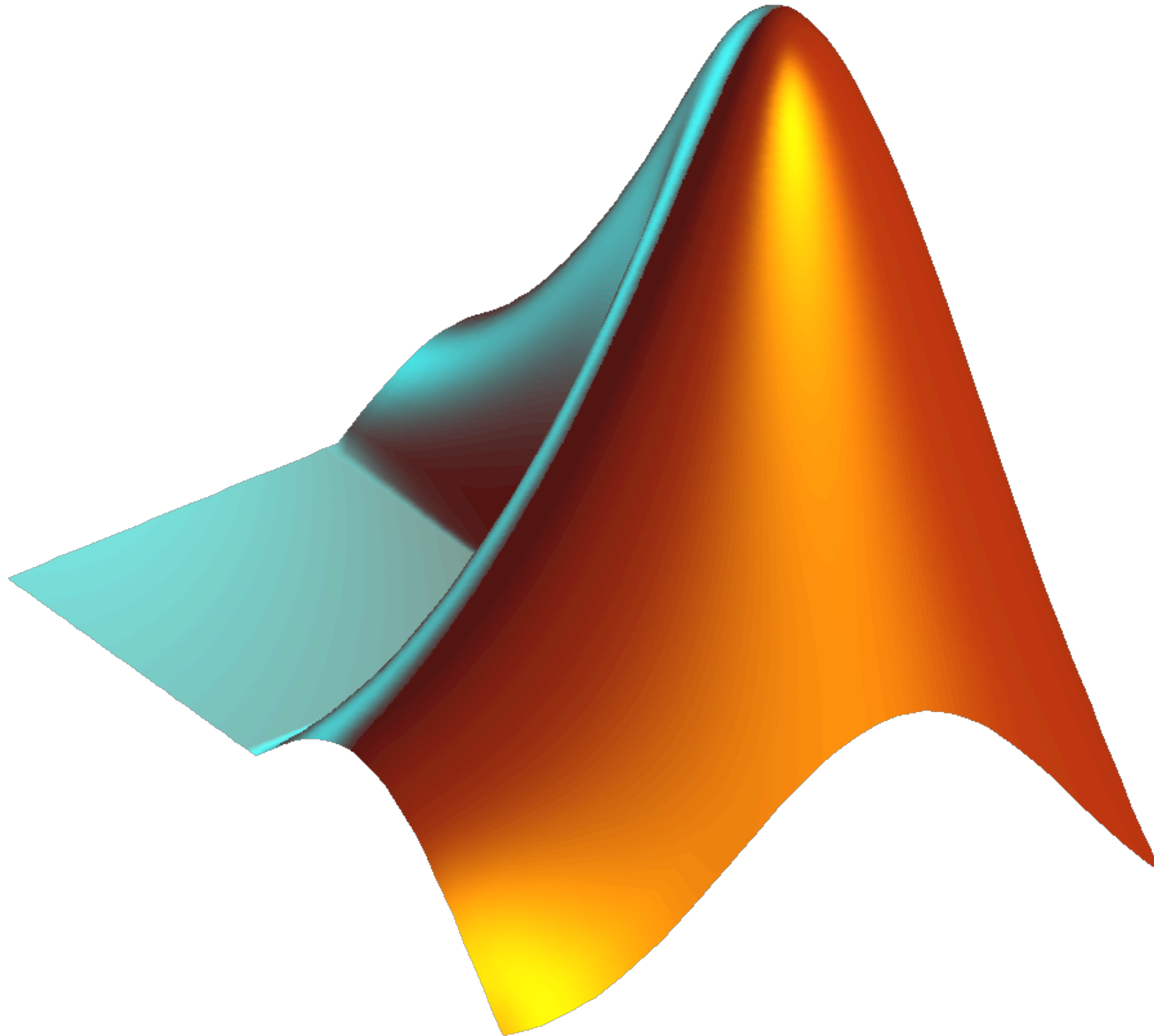
```
channelNames{1} = 'dapi';
```

```
channelNames{2} = 'gfp';
```

```
channelNames{3} = 'neun';
```

```
channelNames    evaluates to    'dapi'  
                                                         'gfp'  
                                                         'neun'
```

Demo: Cell arrays and `{}` indexing



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).

The other, less common case is when you want to select part of a cell array (i.e. several cells) 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,:);
```

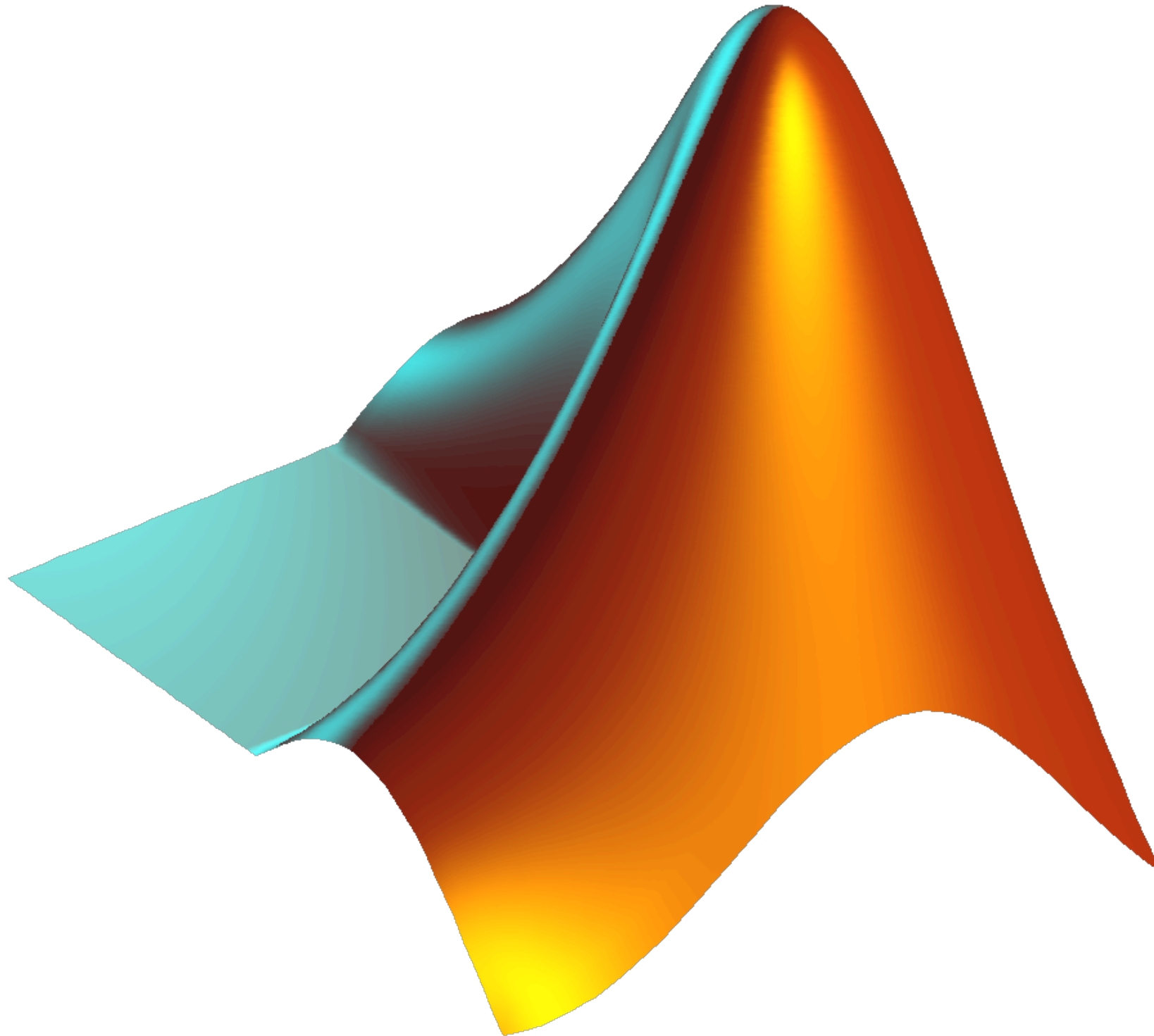
```
dataSubj1 evaluates to [] [] [5 10 102]
```

```
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.

Demo: Cell arrays and `()` indexing



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]  
[]  
[]
```

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

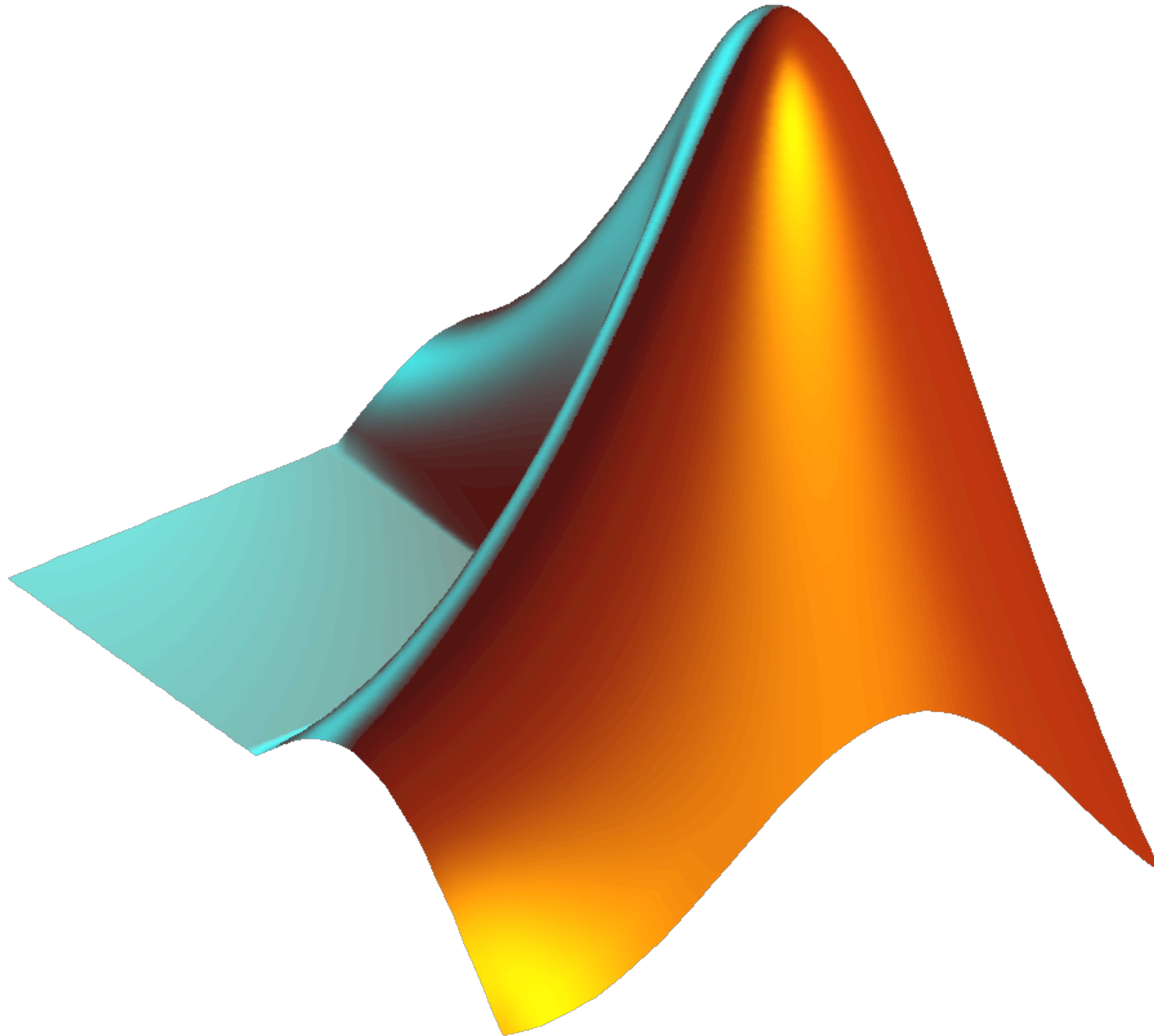
```
channelNames = {'dapi'; 'gfp'; 'neun'};  
moreChannelNames = {'vglut'; 'syn'};  
allChannels = [channelNames; moreChannelNames]
```

evaluates to

- 'dapi'
- 'gfp'
- 'neun'
- 'vglut'
- 'syn'

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

Demo: Cell array concatenation



Outline

Strings

- Comparing and concatenating strings
- String formatting, `fprintf()`, `sprintf()`
- Working with filenames

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

Control Flow

- Branching
- Loops
- Functions and variable scope

File importing

- `csvread`, `dlmread`, `xlsread`
- `textscan` and file handles

Assignment 3 Overview

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
```

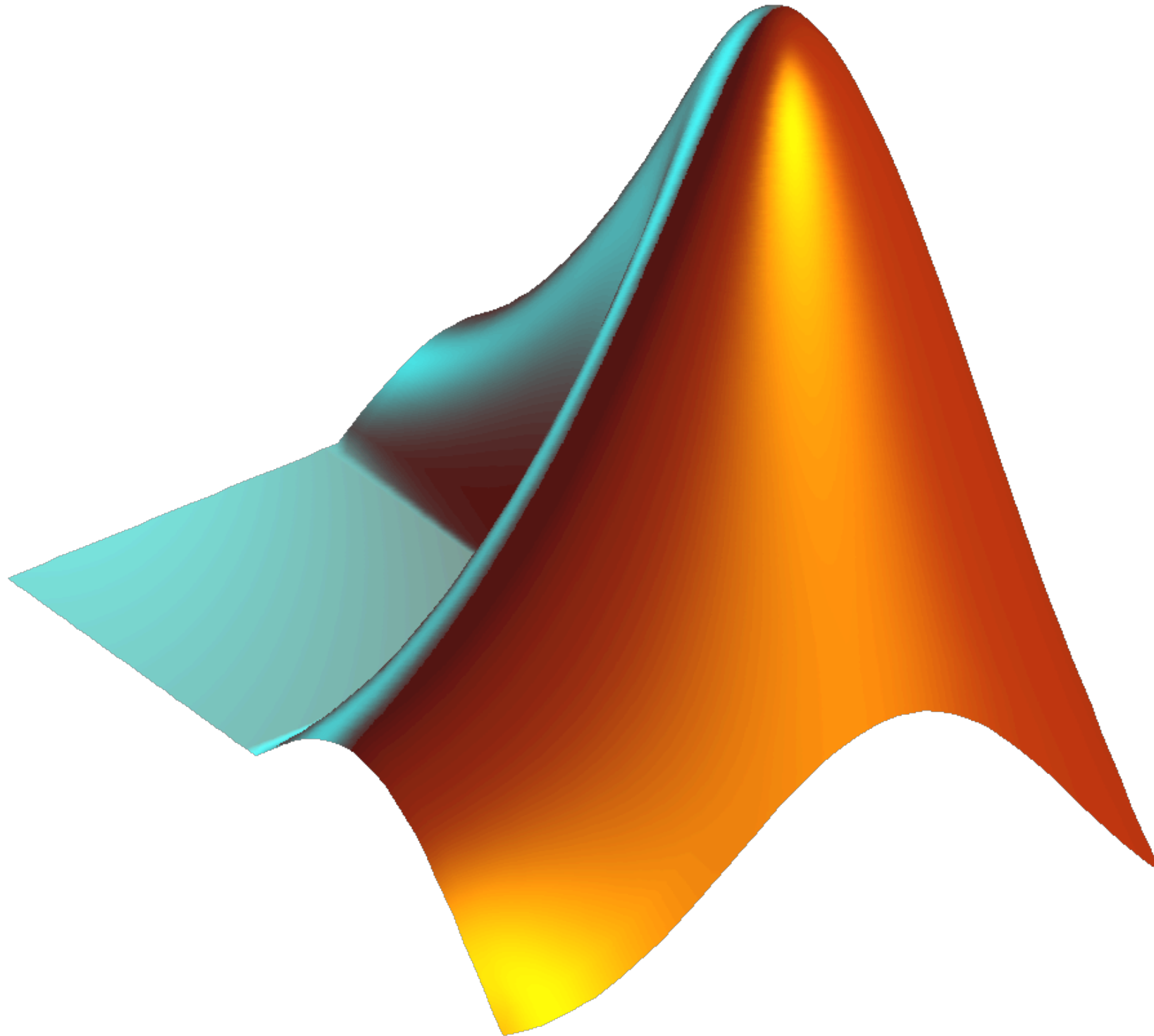
Struct array indexing

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'
```

Demo: Structures



fieldnames() function

Returns the **list of fields** within a structure (or struct array) as a **cell array of strings**.

patchData evaluates to

1x2 struct array with fields:

date

opsin

freqTracking

fieldnames(patchData) evaluates to

'date'

'opsin'

'freqTracking'

isfield() function

`isfield(someStruct, fldName)` checks whether `fldName` is the name of a field found within `someStruct`. Returns logical 0=no or 1=yes

`patchData` evaluates to

1x2 struct array with fields:

date

opsin

freqTracking

`isfield(patchData, 'date')` evaluates to **1**

`isfield(patchData, 'patcher')` evaluates to **0**

▪ (fieldName) notation

Occasionally useful if you want to access a field whose name is specified by a string (say, as an input from the user).

```
fieldNameToGrab = 'date';
```

```
patchData(1).(fieldNameToGrab) evaluates to
```

```
'2011-07-09'
```

This is functionally equivalent to writing:

```
patchData(1).date
```

Or, if you find this syntax unwieldy, look at `getField()`

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;
```

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?

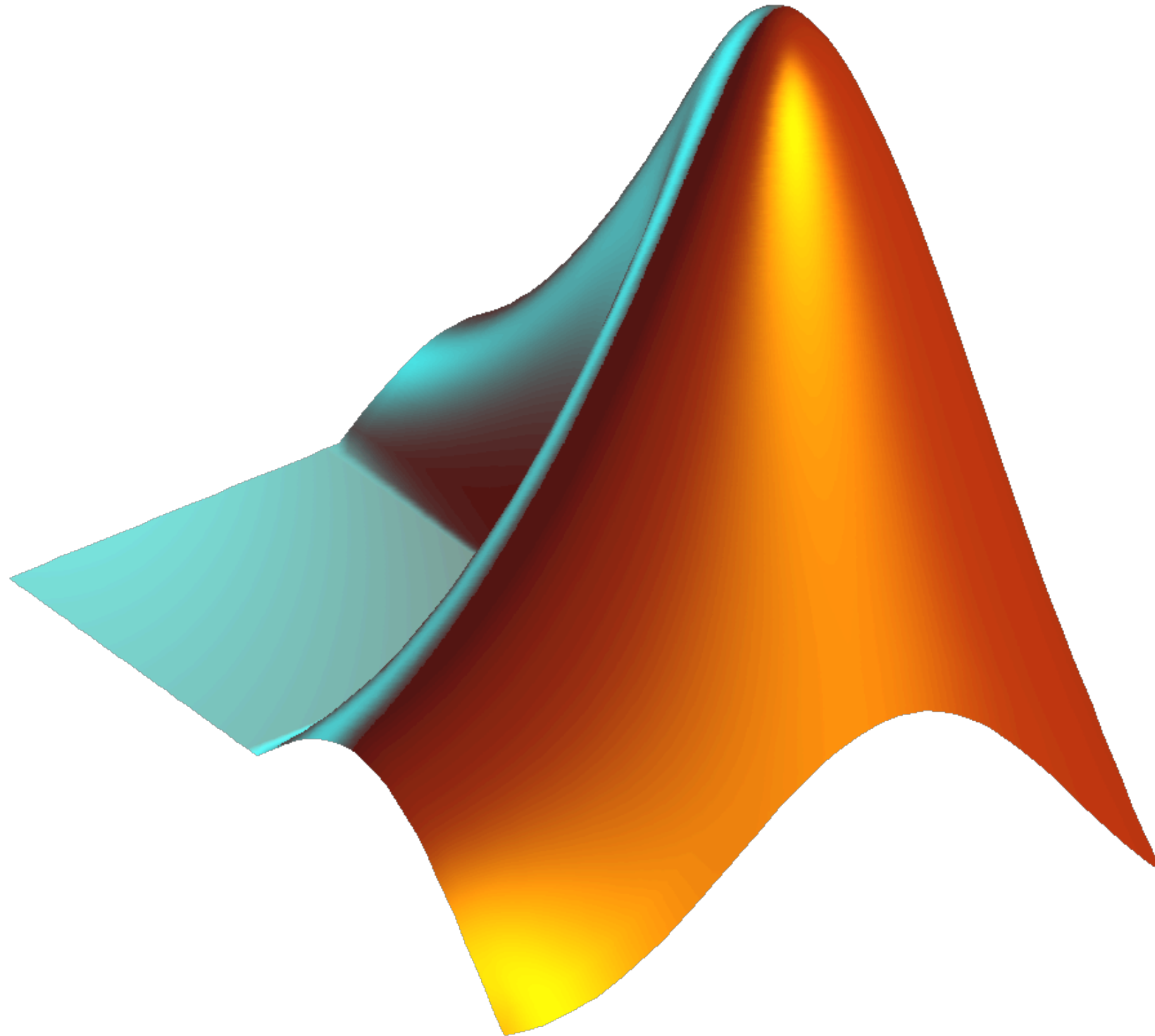
The idea is to grab the field from entire the struct array (without indexing first)

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'
```

Demo: Struct arrays and field capturing



Outline

Strings

- Comparing and concatenating strings
- String formatting, `fprintf()`, `sprintf()`
- Working with filenames

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

Control Flow

- Branching
- Loops
- Functions and variable scope

File importing

- `csvread`, `dlmread`, `xlsread`
- `textscan` and file handles

Assignment 3 Overview

Control flow

Code becomes more flexible when it **need not execute in a linear fashion**, one line after the other, every time it's executed.

We can use MATLAB keywords like if, for, while, etc. to control how MATLAB's execution flows through the code.

The real benefit is that controlling the flow of execution allows you to make sure you **don't repeat yourself** by duplicating code that does the same thing.

Branching: `if`

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`

A word about conditions

When testing a condition:

- 0 is considered false
- Anything non-zero means true
- Doesn't have to be logical, though logicals work fine

Combine conditions using the double `&&` and `||` operators

- Use single `&` and `|` to combine conditions when building logical arrays
- Use double `&&` and `||` to combine conditions that are scalar (i.e. a single value, 0 or non-zero)

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
```

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 modeName
    case 'mode1'
        % run this code if modeName is 'mode1'
    case 'mode2'
        % run this code if modeName is 'mode2'
    otherwise
        % run this code if modeName is none
        % of the above
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
```


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: 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: `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!
```

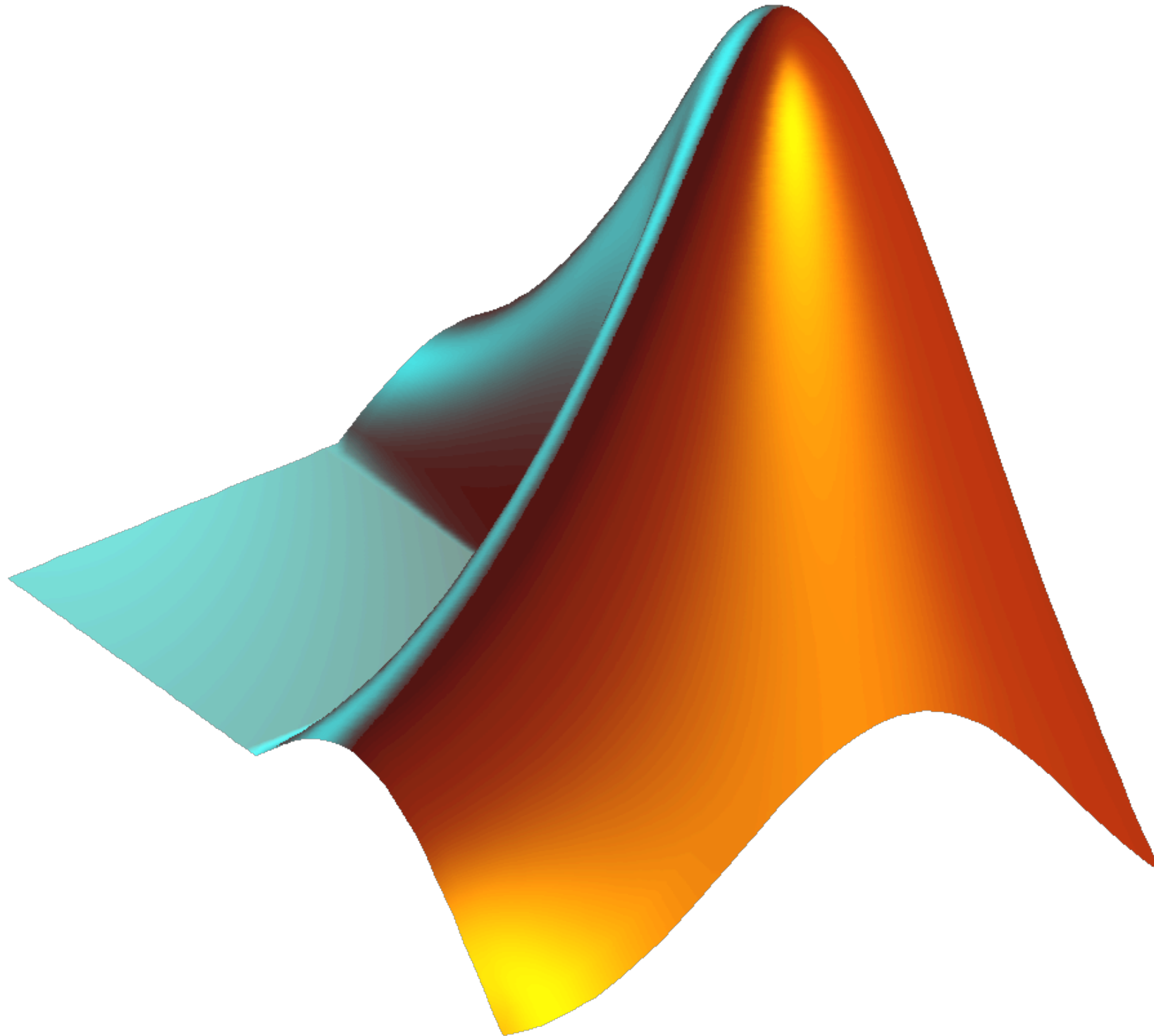
Looping: break and continue

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
```

Demo: Branching and loops



Functions

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

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

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

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

```
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

Scope: where variables exist

In bar.m:

```
function foo = bar(x)
```

```
    x = 2*x; % when bar(1) is called, x==1
```

```
    foo = 3*x; % inside bar, x==2, foo==6
```

```
end
```

```
x = 1;
```

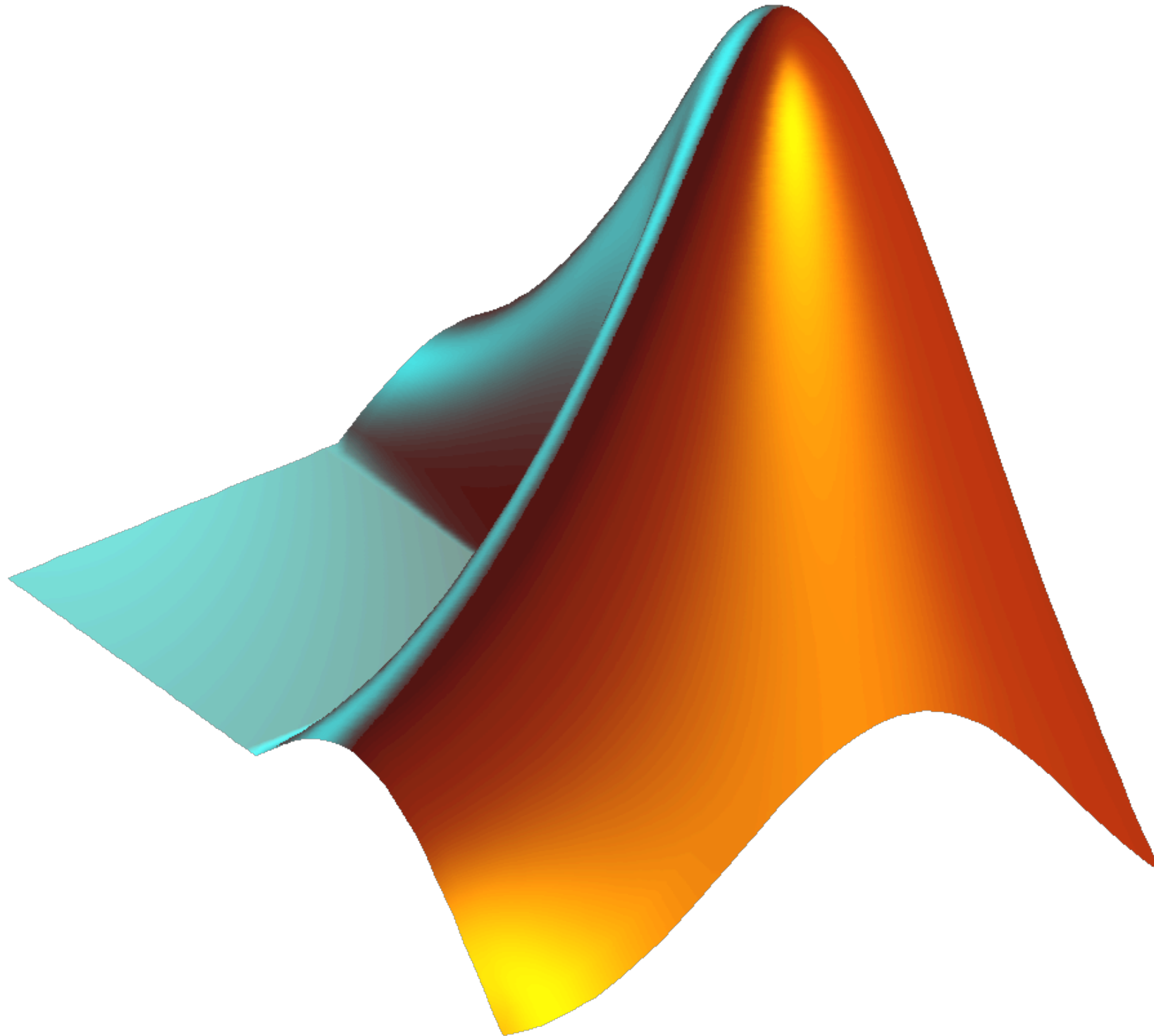
```
z = bar(x); % only z gets set here, not x
```

x still evaluates to 1 because this x is completely separate from the x inside bar()

z evaluates to 6

foo doesn't exist here, it only exists temporarily inside bar()

Demo: Functions and variable scope



Outline

Strings

- Comparing and concatenating strings
- String formatting, `fprintf()`, `sprintf()`
- Working with filenames

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

Control Flow

- Branching
- Loops
- Functions and variable scope

File importing

- `csvread`, `dlmread`, `xlsread`
- `textscan` and file handles

Assignment 3 Overview

File importing

File importing

MATLAB offers functions that load some common file formats:

File importing

MATLAB offers functions that load some common file formats:

- `csvread`: comma separated value .csv files containing only numeric data

File importing

MATLAB offers functions that load some common file formats:

- `csvread`: comma separated value .csv files containing only numeric data
- `dlmread`: delimited dat file containing only numeric data separated by a delimiter character (space, tab, newline, etc.)

File importing

MATLAB offers functions that load some common file formats:

- `csvread`: comma separated value .csv files containing only numeric data
- `dlmread`: delimited dat file containing only numeric data separated by a delimiter character (space, tab, newline, etc.)
- `xlsread`: read Excel spreadsheet

File importing

MATLAB offers functions that load some common file formats:

- `csvread`: comma separated value .csv files containing only numeric data
- `dlmread`: delimited dat file containing only numeric data separated by a delimiter character (space, tab, newline, etc.)
- `xlsread`: read Excel spreadsheet
- `textscan`: read data in a file with a custom format

File importing

MATLAB offers functions that load some common file formats:

- `csvread`: comma separated value .csv files containing only numeric data
- `dlmread`: delimited dat file containing only numeric data separated by a delimiter character (space, tab, newline, etc.)
- `xlsread`: read Excel spreadsheet
- `textscan`: read data in a file with a custom format
- `imread`: numerous image formats

File importing

MATLAB offers functions that load some common file formats:

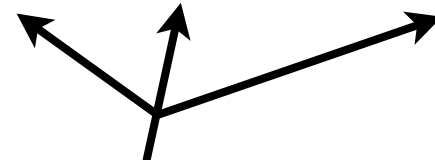
- `csvread`: comma separated value .csv files containing only numeric data
- `dlmread`: delimited dat file containing only numeric data separated by a delimiter character (space, tab, newline, etc.)
- `xlsread`: read Excel spreadsheet
- `textscan`: read data in a file with a custom format
- `imread`: numerous image formats
- `fread`, `fgetl`, `fscanf`, `fseek`: low-level line by line I/O

csvread() function

Reads a file with only numeric data separated by commas and newlines. Returns a matrix of those values. Use row, col, and range to select particular rows and columns.

Not very useful if your data has a mix of numeric and text information in it. In that case, see `textscan()`

```
M = csvread(filename, row, col, range)
```



These three arguments are optional

If filename contained:

M would evaluate to:

02,	04,	06,	08,	10,	12	2	4	6	8	10	12
03,	06,	09,	12,	15,	18	3	6	9	12	15	18
05,	10,	15,	20,	25,	30	5	10	15	20	25	30
07,	14,	21,	28,	35,	42	7	14	21	28	35	42
11,	22,	33,	44,	55,	66	11	22	33	44	55	66

csvread() function

Reads a file with only numeric data separated by commas and newlines. Returns a matrix of those values. Use row, col, and range to select particular rows and columns.

If you need to skip a header line, use 1 in the second argument.

```
M = csvread('data.csv', 1)
```

↖ Means skip the first 1 row

If data.csv contained:

a,	b,	c,	d,	e,	f
02,	04,	06,	08,	10,	12
03,	06,	09,	12,	15,	18
05,	10,	15,	20,	25,	30
07,	14,	21,	28,	35,	42
11,	22,	33,	44,	55,	66

M would evaluate to:

2	4	6	8	10	12
3	6	9	12	15	18
5	10	15	20	25	30
7	14	21	28	35	42
11	22	33	44	55	66

dlmread() function

Reads a file with only numeric data **separated by a specific character** (like tabs, spaces, etc.) and newlines. Returns a matrix of those values.

Same optional arguments as csvread

`M = dlmread(filename, delimiter, row, col, range)`

`M = dlmread('data.csv', ' ')`

If data.csv contained:

M would evaluate to:

02	04	06	08	10	12
03	06	09	12	15	18
05	10	15	20	25	30
07	14	21	28	35	42
11	22	33	44	55	66

2	4	6	8	10	12
3	6	9	12	15	18
5	10	15	20	25	30
7	14	21	28	35	42
11	22	33	44	55	66

xlsread() function

Reads an Excel spreadsheet. Only opens XLS 97-2000 unless you have Excel installed and you're running Windows.

[num, txt, raw] = xlsread(filename, sheet, range)

Optional: index of cells, e.g. 'B2:D5'

Optional: Name or number of sheet to load

Numeric data as 2d array

Text data as cell array

All data (numeric and text) as cell array

The diagram illustrates the `xlsread` function signature: `[num, txt, raw] = xlsread(filename, sheet, range)`. Arrows point from descriptive text to the arguments and return values. An arrow points from 'Optional: index of cells, e.g. 'B2:D5'' to the `range` argument. Another arrow points from 'Optional: Name or number of sheet to load' to the `sheet` argument. Three arrows point from the return values to their descriptions: from `num` to 'Numeric data as 2d array', from `txt` to 'Text data as cell array', and from `raw` to 'All data (numeric and text) as cell array'.

textscan() function

Reads a text file with text data and numeric data in a user-specified format, very similar to `fprintf`'s format string.

This you get from `fopen()`:
It's a file handle, not a file name



```
C = textscan(fid, format)
```



This is the format string

`textscan` attempts to match the format string against each line of the file **until it fails**. Then it returns the **results in a cell array with one element for each field**. The contents of each element reflect the type of data matched by the `%` marker.

Quick detour: file handles

A file handle, like all handles in MATLAB, is essentially just a number. MATLAB's internals use this number to keep track of information about something, in this case the state of a file it is reading from / writing to.

Open a file for reading:

```
fid = fopen(filename);
```

Close the file handle when you're done with it:

```
fclose(fid)
```

textscan() example

data.dat contains:

09/12/2005	Level1	12.34	45	1.23e10	inf	Nan	Yes	5.1+3i
10/12/2005	Level2	23.54	60	9e19	-inf	0.001	No	2.2-.5i
11/12/2005	Level3	34.90	12	2e5	10	100	No	3.1+.1i

Then run:

```
fid = fopen('data.dat');  
C = textscan(fid, '%s %s %f %d %u %f %f %s %f');  
fclose(fid);
```

floating point (i.e. double)

signed integer (i.e. int32)

unsigned integer (i.e. uint32)

string (i.e. char array)

textscan() example

```
fid = fopen('data.dat');  
C = textscan(fid, '%s %s %f %d %u %f %f %s %f');  
fclose(fid);
```

C is a 9x1 cell array:

C{1} = {'09/12/2005'; '10/12/2005'; '11/12/2005'}	class cell
C{2} = {'Level1'; 'Level2'; 'Level3'}	class cell
C{3} = [12.34; 23.54; 34.9]	class single
C{4} = [45; 60; 12]	class int8
C{5} = [4294967295; 4294967295; 200000]	class uint32
C{6} = [Inf; -Inf; 10]	class double
C{7} = [NaN; 0.001; 100]	class double
C{8} = {'Yes'; 'No'; 'No'}	class cell
C{9} = [5.1+3.0i; 2.2-0.5i; 3.1+0.1i]	class double

textscan() example

patching.csv contains:

```
Date,Cell Number,Opsin Construct,ABF Name  
7/10/2011,1,ChR2,10001.abf  
7/10/2011,2,ChETA,10002.abf  
7/11/2011,1,ChETA,10003.abf  
7/11/2011,2,ChR2,10004.abf
```

```
M = csvread('patching.csv')
```

```
??? Error using ==> dlmread at 145
```

```
Mismatch between file and format string.
```

```
Trouble reading number from file (row 1, field 1) ==> Date,
```

```
Error in ==> csvread at 50
```

```
    m=dlmread(filename, ',', r, c);
```

`csvread()` only works for numeric data, this file has strings in it, and not just in the header line

textscan() example

patching.csv contains:

```
Date,Cell Number,Opsin Construct,ABF Name  
7/10/2011,1,ChR2,10001.abf  
7/10/2011,2,ChETA,10002.abf  
7/11/2011,1,ChETA,10003.abf  
7/11/2011,2,ChR2,10004.abf
```

Design a format string to grab these four fields:

```
fid = fopen('patching.csv');  
C = textscan(fid, '%s %u %s %s');  
fclose(fid);
```

string (date)

unsigned int (cell number)

string (opsin name)

string (abf file name)

textscan() example

```
fid = fopen('patching.csv');  
C = textscan(fid, '%s %u %s %s');
```

C is a 4x1 cell array:

```
C{1} = {'Date,Cell'}  
C{2} = [0x1 uint32]  
C{3} = {0x1 cell}  
C{4} = {0x1 cell}
```

What went wrong here?

- First line of patching.csv is 'Date,Cell Number,Opsin Construct,ABF Name'
- `textscan()` tries to match `%s` against the first field, where fields are defined using the default delimiter of space ' ', so it grabs 'Date, Cell'
- Then it tries to match `%u` against the second field 'Number,Opsin', which fails, since it's not an integer.

Second quick detour: file handles

MATLAB keeps track of where it's read to in each of the files it has a handle open for.

So when `textscan()` reads until it fails, the position of `fid` is set to wherever it left off reading.

The next time you call `textscan()`, it picks up reading wherever the position is set to.

If you want to start over from the beginning, use:

```
frewind(fid);
```

Or alternatively, you can `fclose` it and then `fopen` it again.

textscan() example

```
frewind(fid);  
C = textscan(fid, '%s %u %s %s', 'HeaderLines', 1);
```

C is a 4x1 cell array:

```
C{1} = {'7/10/2011,1,ChR2,10001.abf'  
        '7/10/2011,2,ChETA,10002.abf'  
        '7/11/2011,1,ChETA,10003.abf'  
        '7/11/2011,2,ChR2,10004.abf'}  
C{2} = [0; 0; 0]  
C{3} = {''; '';  
C{4} = {''; '';
```

Skip the first line!



A little better, but why is the whole line in field 1?

- textscan looks for a delimiter character to separate the fields in each line.
- By default, the delimiter is the space character
- We need it to be a comma


textscan() example

```
frewind(fid);  
C = textscan(fid, '%s %u %s %s', ...  
            'HeaderLines', 1, 'Delimiter', ',');
```

Skip the first line!



Fields are separated
with commas

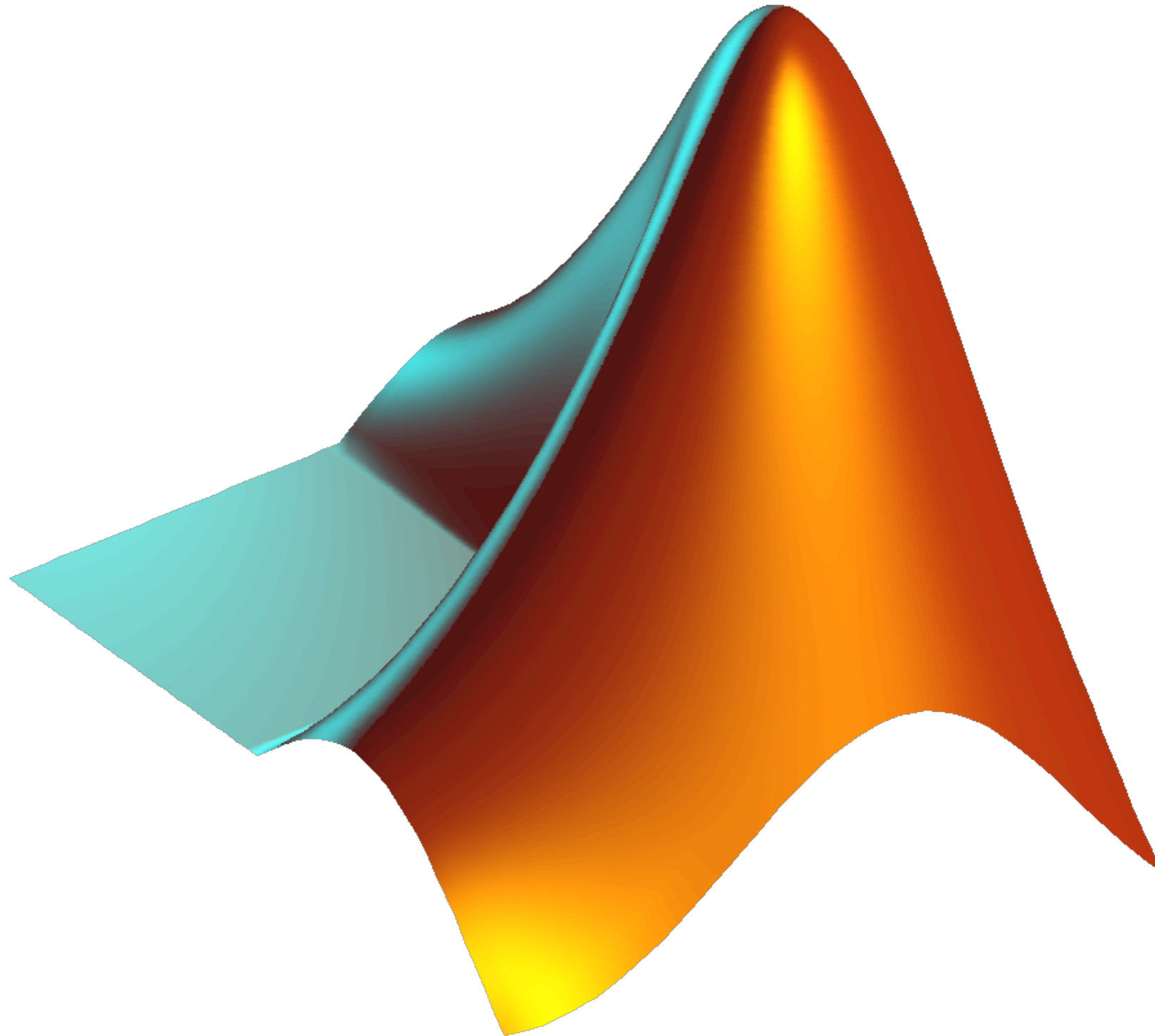


C is a 4x1 cell array:

```
C{1} = {'7/10/2011'; '7/10/2011'; '7/11/2011'; '7/11/2011'}  
C{2} = [1; 2; 1; 2]  
C{3} = {'ChR2'; 'ChETA'; 'ChETA'; 'ChR2'}  
C{4} = {'10001.abf'; '10002.abf'; '10003.abf', '10004.abf'}
```

Perfect!

Demo: textscan()



Organizing loaded data

Now how should we keep track of this data?

Approach 1 - Separate variables for each

```
patchDate = C{1};
```

```
patchId = C{2};
```

```
patchOpsin = C{3};
```

```
patchABFName = C{4};
```

Organizing loaded data

What if we want to filter this data based on the opsin?

```
selectedNeurons = strcmp(patchOpsin, 'ChR2');
```

```
patchDate = patchDate(selectedNeurons);
```

This is logical indexing!

```
patchId = patchId(selectedNeurons);
```

```
patchOpsin = patchOpsin(selectedNeurons);
```

```
patchABFName = patchABFName(selectedNeurons);
```

Note that we use () indexing on the cell arrays because we just want to select particular cells, not extract the contents.

This gets tedious, especially when you decide to add more information to the spreadsheet (a new variable) and have to add a line of code everywhere you index into the dataset like this.

Organizing loaded data

Now how should we keep track of this data?

**Approach 2 - Use a struct for each neuron,
aggregate the dataset into a struct array**

```
nPatched = numel(C{1});  
for iPatch = 1:nPatched  
    patchData(iPatch).date      = C{1}{iPatch};  
    patchData(iPatch).id        = C{2}(iPatch);  
    patchData(iPatch).opsin     = C{3}{iPatch};  
    patchData(iPatch).abfName   = C{4}{iPatch};  
end
```

Organizing loaded data

Now how should we keep track of this data?

**Approach 2 - Use a struct for each neuron,
aggregate the dataset into a struct array**

```
nPatched = numel(C{1});  
for iPatch = 1:nPatched  
    patchData(iPatch).date  
    patchData(iPatch).id  
    patchData(iPatch).opsin  
    patchData(iPatch).abfName  
end
```

Inside cell 1 of cell array
C, there is an entire cell
array of strings.



```
= C{1}{iPatch};  
= C{2}(iPatch);  
= C{3}{iPatch};  
= C{4}{iPatch};
```

Organizing loaded data

Now how should we keep track of this data?

**Approach 2 - Use a struct for each neuron,
aggregate the dataset into a struct array**

```
nPatched = numel(C{1});  
for iPatch = 1:nPatched  
    patchData(iPatch).date  
    patchData(iPatch).id  
    patchData(iPatch).opsin  
    patchData(iPatch).abfName  
  
end
```

Since C{1} is itself a cell array, we use {} again to extract the contents of the cell at index iPatch in C{1}

↓
= C{1}{iPatch};
= C{2}(iPatch);
= C{3}{iPatch};
= C{4}{iPatch};

Organizing loaded data

Now how should we keep track of this data?

**Approach 2 - Use a struct for each neuron,
aggregate the dataset into a struct array**

```
nPatched = numel(C{1});
```

```
for iPatch = 1:nPatched
```

```
    patchData(iPatch).date
```

```
    patchData(iPatch).id
```

```
    patchData(iPatch).opsin
```

```
    patchData(iPatch).abfName
```

```
end
```

C{2}, however, is a numeric array, so we just use () like we normally do on numeric arrays.

```
= C{1}{iPatch};
```

```
= C{2}(iPatch);
```

```
= C{3}{iPatch};
```

```
= C{4}{iPatch};
```


Organizing loaded data

What if we want to filter this data based on the opsin?

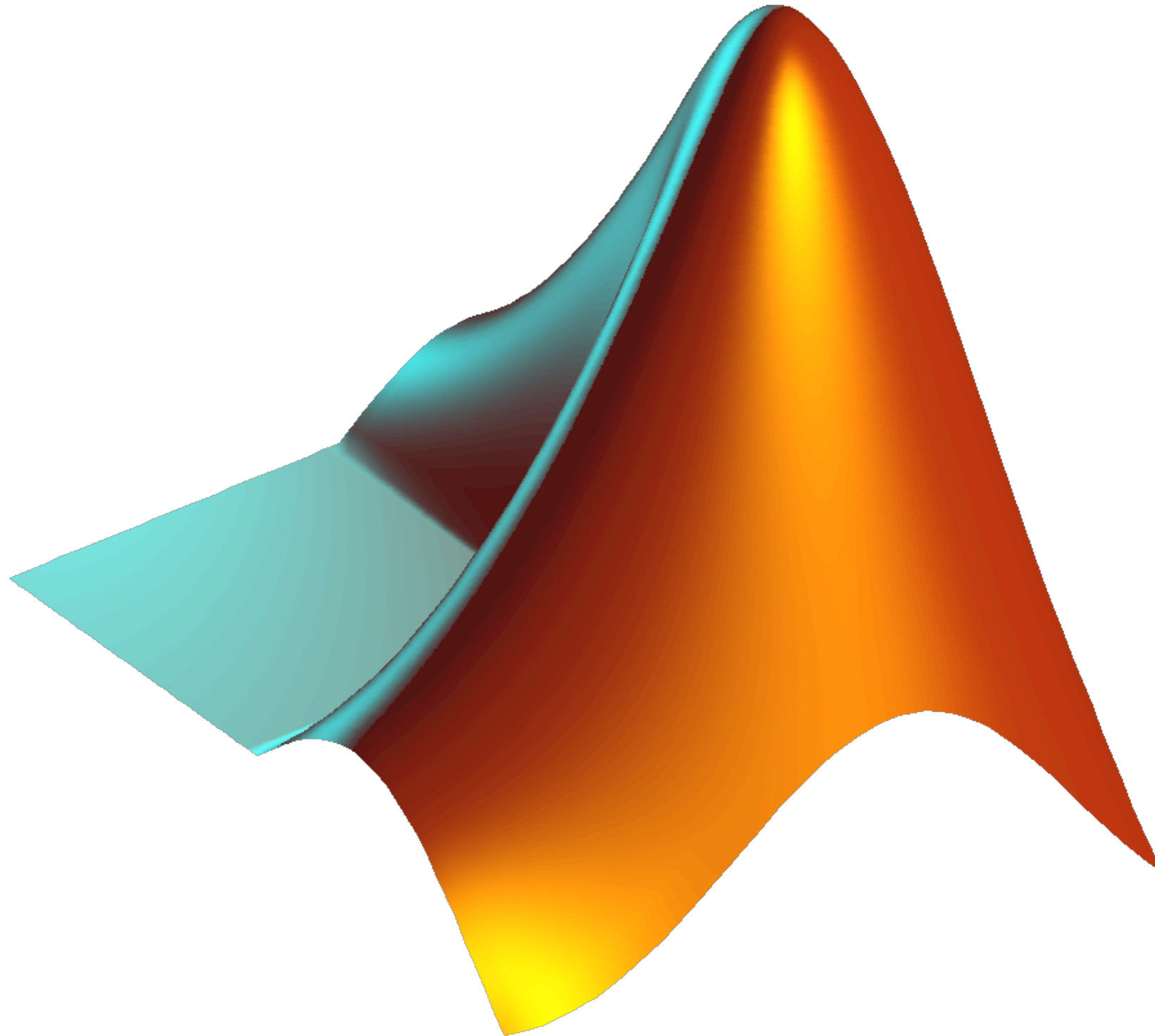
```
selectedCells = strcmp({patchData.opsin}, 'ChR2');  
patchData = patchData(selectedCells);
```

Very easy to filter out whole records (i.e. patched neurons), because all of the information for a specific record is encapsulated into a `struct`.

It's also very easy to pull out a specific record (e.g. to pass to a function or in a loop):

```
currentCell = patchData(iPatch);  
currentOpsin = currentCell.opsin;
```

Demo: Organizing data



unique() function

`unique()` returns the set of unique values in a list of numbers or a cell array of strings

What are the different opsins present in the dataset?

“Capture” the values in a cell array



```
opsinNameByNeuron = {patchData.opsin}
```

evaluates to: {'ChR2'; 'ChETA'; 'ChETA'; 'ChR2'}

```
opsinNames = unique(opsinNameByNeuron)
```

evaluates to: {'ChETA'; 'ChR2'} ← unique sorts the results too!

Outline

Strings

- Comparing and concatenating strings
- String formatting, `fprintf()`, `sprintf()`
- Working with filenames

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

Control Flow

- Branching
- Loops
- Functions and variable scope

File importing

- `csvread`, `dlmread`, `xlsread`
- `textscan` and file handles

Assignment 3 Overview

Looping over opsins, over neurons

```
opsinNames = unique({patchData.opsin});  
nOpsins = numel(opsinNames);  
% loop over the set of unique opsins in the dataset  
for iOpsin = 1:nOpsins  
    currentOpsin = opsinNames{iOpsin};  
    matching = strcmp(currentOpsin, {patchData.opsin});  
    currentOpsinData = patchData(matching);  
  
    % loop over neurons with the current opsin  
    for iNeuron = 1:numel(currentOpsinData)  
        currentNeuronData = currentOpsinData(iNeuron);  
        % do something useful with currentNeuronData  
    end  
end  
end
```

mean() function

`mean()` computes the average (sample mean) of a list of numbers. When dealing with matrices, you need to specify which dimension to average along.

`mean(X, 1)` means return the average row (average down the column, across the rows). This is the default if you only specify one argument.

`mean(X, 2)` means return the average column (average across the columns, down the row)

mean() function

`mean()` computes the average (sample mean) of a list of numbers. When dealing with matrices, you need to specify which dimension to average along.

$X =$

	Dim 2 ↔	
Dim 1 ↑ ↓	26	0
	15	15
	1	1
	2.4	0

`mean(X)`

`mean(X, 1)` evaluates to

11.1	4
------	---

`mean(X, 2)` evaluates to

13
15
1
1.2

std() function

std() computes the standard deviation of a list of numbers

- When dealing with matrices, you need to specify which dimension to average along, **except as the third argument**.
- The second argument should be 0 if you want the unbiased estimator that normalizes by $n-1$, where n is the number of samples

