MATLAB: A Comprehensive Introduction

Catherine Paverd

catherine.paverd@eng.ox.ac.uk

External Data

- MATLAB is able to import and export data from different file types
- Depending on the file there are different import and export methods
- We looked at some of the image data types in the previous lesson

MAT Files

- MAT-files:
  - The default data storage file type in MATLAB.
  - Binary files that can store the variables in the current workspace for later use.
  - They use the **.mat** extension.

- To import and export from MAT-files, the load and save functions are used:
  - **save**: creates a binary MAT-file named *matlab.mat* in the current folder and saves *all current variables in the workspace* into that file.
  - **load**: loads all the variables stored in matlab.mat into the current workspace, overwriting existing values.

MAT Files – save() function

- The function *save* creates a MAT file called *matlab.mat* in the current folder and **saves all current variables in the workspace to it**
- Note that *save* by itself **always saves to the file *matlab.mat*** and so it will overwrite any previous data saved in that file
- However, the *save* function **can take in a name of a specific file to save to**, and in that way we can stop it from overwriting the previously saved data

MAT Files – save() function

- In summary
  - *save* → saves all workspace variables to *matlab.mat*
  - *save('filename')* → save all workspace variables to *filename.mat*

- You can also specify which variables you would like to save by writing the names of those variables after the 'filename' argument:
  - *save ('filename', 'variable1', 'variable2', 'variable3')* → saves the specified variables to *filename.mat*
  - **Note that when using multiple arguments with save(), MATLAB will always take the first argument inside the bracket as the filename**

MAT Files – load() function

- Similar to save, load will
  - *load* → loads all workspace variables from *matlab.mat*
  - *load('filename')* → loads all workspace variables from *filename.mat*

- You can also specify which variables you would like to load by writing the names of those variables after the 'filename' argument:
  - *load ('filename', 'variable1', 'variable2')* → loads the specified variables from *filename.mat*
  - **Note that when using multiple arguments with load(), MATLAB will always take the first argument inside the bracket as the filename**

MAT Files – whos() function

- The function *whos* allows you to view the contents of the workspace or a file without loading it

- In a similar way to *save*, *whos* can also return the contents of a specified file, or details of a specified variable
  - *whos('-file', 'filename')* → returns the details about the variables in *filename*
  - *whos('-file', 'filename', 'variable1')* → return details about *variable1* in file *filename*

MAT Files – GUI Import and Export

- The MATLAB GUI can also be used to **inspect** and **load** MAT-file data.
- To see the variables in a MAT-file before loading: click the file in the Current Folder browser. Information appears in the Details Panel (bottom left of the GUI).
- **Double clicking loads** all the data from the selected file into the workspace.
- To **select** and **load** MAT-file variables interactively:
  - *Click Import Data* on the Desktop Home tab, or
  - *Drag* variables from the Details Panel of the Current Folder browser to the Workspace browser.
- To **save** data via the GUI:
  - *Click Save Workspace* on the Desktop Home tab, or
  - *Drag* variables from the Workspace browser to the Current Folder browser.

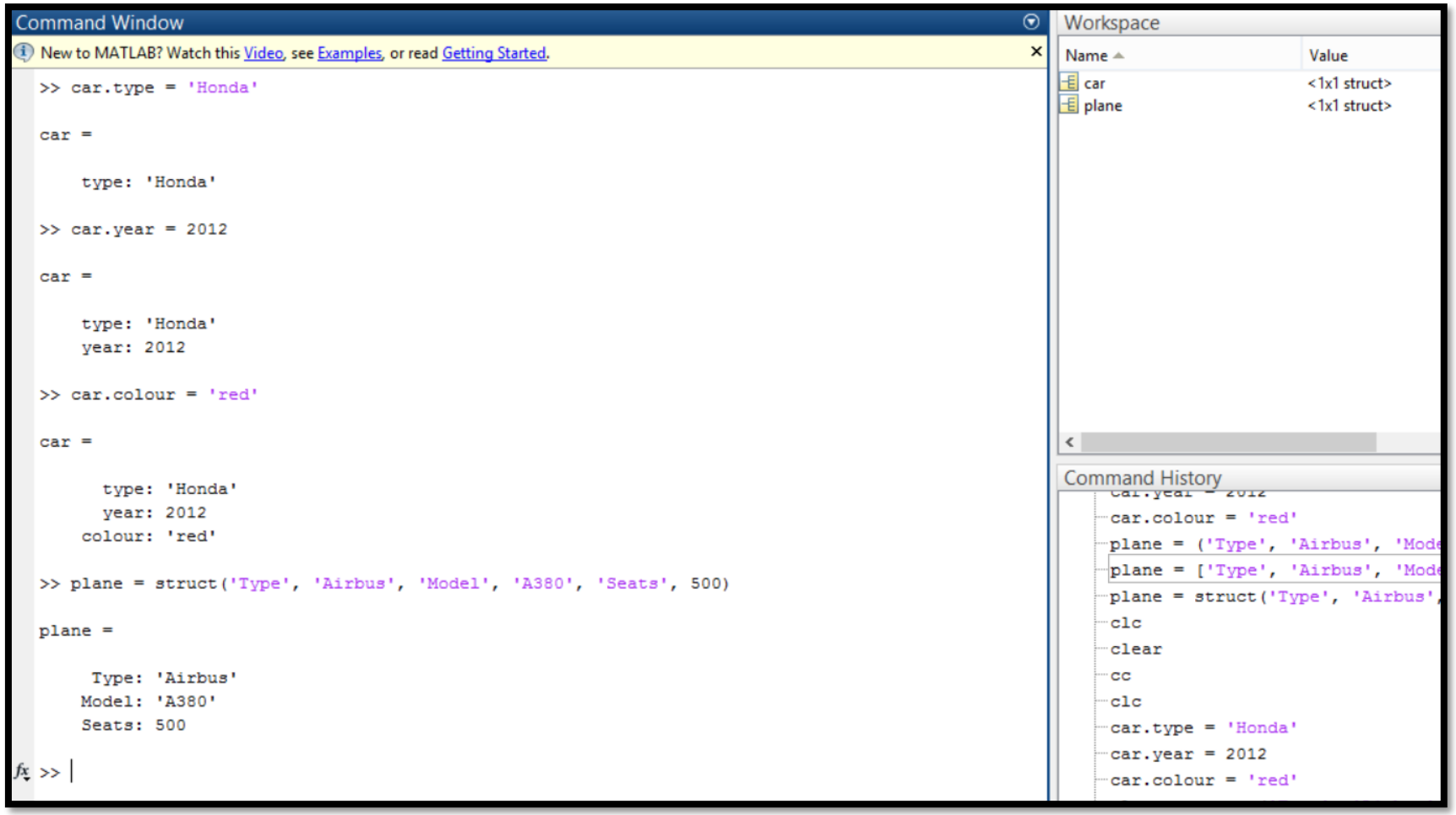
A Note on Cell Array and Structures

- So far we have mostly stored our data in either a numeric or char class in MATLAB
- However, if you have different types and lengths you may need to use either a **cell array** or a **struct**

- **struct**
  - Class used to store data of different types and lengths that you would like to access by name
  - Use . notation to access data in a struct

- **cell array**
  - Class used to store different types and lengths of data that you would like to access in array form
  - Access actual data using curly brackets { }

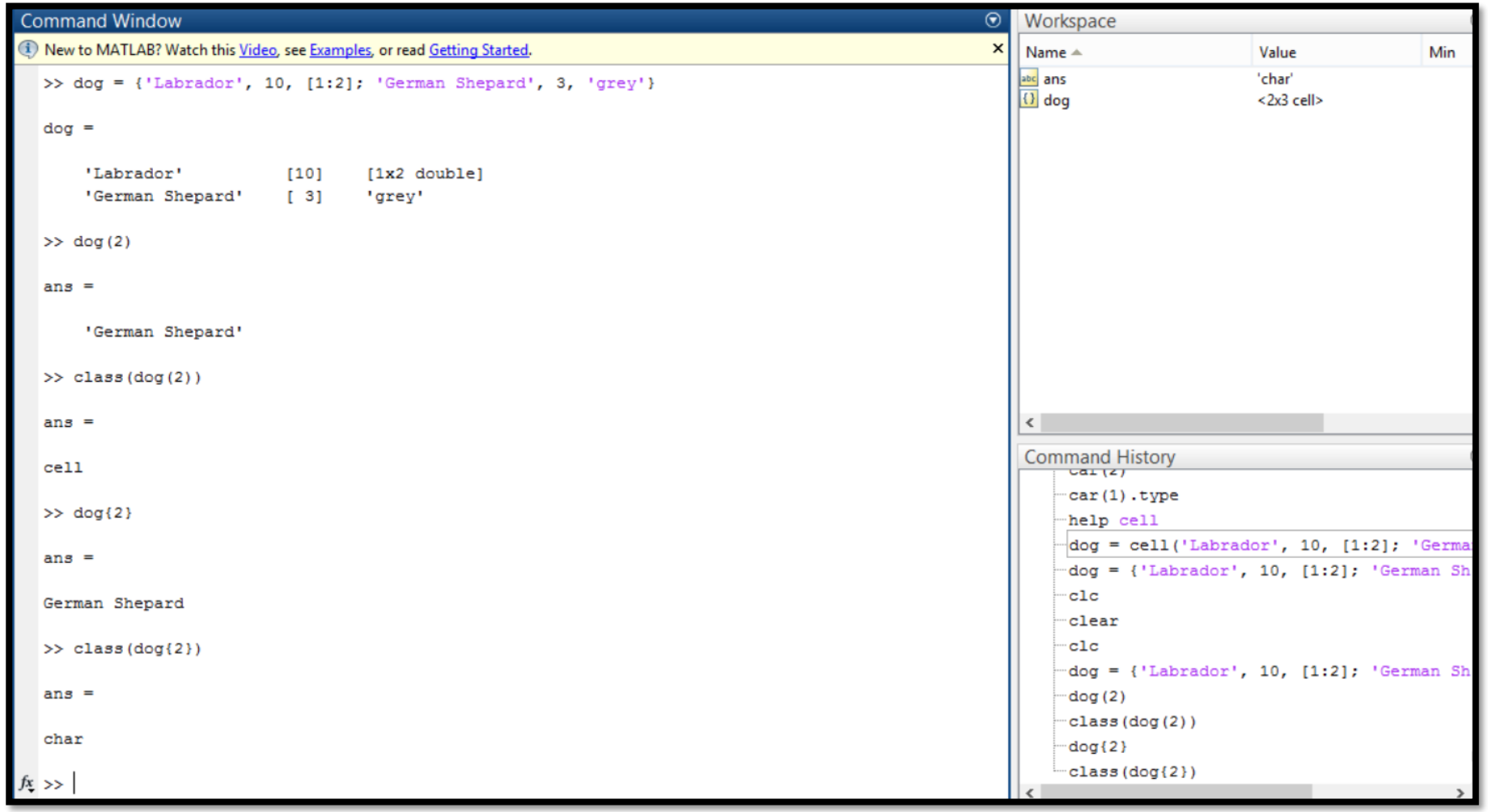
A Note on Cell Array and Structures



A Note on Cell Array and Structures

```
>> car(2).type = 'Audi';
>> car(2).year = 2000;
>> car(2).colour = 'Silver';
>> car(1)

ans =

      type: 'Honda'
      year: 2012
    colour: 'red'

>> car(2)

ans =

      type: 'Audi'
      year: 2000
    colour: 'Silver'

>> car(1).type

ans =

Honda
```

A Note on Cell Array and Structures



Excel Files – xlsread()

- In addition to reading MATLAB data files, you can also import data from other files, such as Excel
- *xlsread('filename.xls')* → reads in numeric and text data separately from *filename.xls*

```
>> clear all
[num,txt] = xlsread('hospital.xls');
>> whos
  Name         Size                Bytes  Class     Attributes

  num        100x9                  7200  double
  txt        101x12               138660  cell
```

Excel Files – xlsread() and xlsfinfo()

- By default, *xlsread('filename.xls')* only reads in the first worksheet
- To read in other worksheets, use the specific worksheet name after the filename
  - *[num, txt] = xlsread('filename.xlsx', 'worksheetname', 'range')*
  - *[num, txt] = xlsread('filename.xlsx',* worksheetnumber, *'range')*

- For interactive selection use *xlsread('filename.xlsx',* -1*)* → open a window to allow user to select data

- To gain information about worksheets in the Excel file, use *xlsfinfo('filename')*
  - *[type, sheets] = xlsfinfo('filename.xls')* → store document type in *type* and worksheet names in *sheets*

Excel Files – importdata()

- So far, we have imported data from an Excel file into an array (either with a numeric type or into a cell array)

- However, you can also import Excel data into MATLAB into a struct using *importdata('filename')*

```
>> treasuries = importdata('USTreasury.xls')

treasuries =

          data: [1x1 struct]
      textdata: [1x1 struct]
    colheaders: [1x1 struct]
```

Excel Files – xlswrite()

- *xlswrite('filename', 'array', 'sheet', 'range')*:
  - writes whatever is in *array* to file *filename* at specified *sheet* and *range* (*sheet* and *range* are optional*)*
  - used to export data from Workspace to any worksheet in an Excel file, and to any location within that worksheet.
  - by default, writes to the first worksheets in a file, starting at cell A1.
  - if the target file already exists, it writes data in the existing format.
  - if not, a new file is created using the format corresponding to the file extension specified (the default is .xls)
  - if Excel is not installed, xlswrite exports to a *.csv* file.

Text Files – load() and save()

- There are many methods available to import and export data to text files
- The *load()* and *save()* functions can work for text files if an additional argument is added

```
>> clear all
>> x = magic(3);
>> save('temp.txt','-ascii')
>> load -ascii temp.txt
>> whos
  Name      Size             Bytes  Class     Attributes

  temp      3x3                 72  double
  x         3x3                 72  double

>> isequal(x,temp)

ans =

     1
```

Text Files – other import methods

- Alternatively, we can launch the generic **import wizard GUI** by clicking the Import Data button or running **uiimport**.

- The **importdata** function is largely equivalent but without a graphical interface.

- In general, *load*, *uiimport* and *importdata* assume that the data in an ASCII file are:
  - Rectangular (the same number of data fields in each row)
  - numeric

Text Files – specific formats

- If the text file formatting is known and/or complex, it can be processed using specific import and export function as follows:
  - *Comma-separated value* (.csv) files can be read and written using **csvread** and **csvwrite**.
  - *Delimited numeric data*, with a specified delimiter such as tab, can be read and written using **dlmread** and **dlmwrite**.
  - *Complex text files* can be read using **textscan**.
  - *Single lines of text* can be read using **fscan**, **fgetl** or **fgets**, while fprintf prints a single line of formatted data to a file.

Binary Files

- Generic binary files are handled using low-level I/O functions: **fread**, **fwrite**, **fopen**, **fclose**, **frewind**
- Before reading or writing to a file, it must first be opened using **fopen** to obtain a file identifier.
  - The first argument to fopen is a file name string
  - The second argument is a permission string describing the type of access required (e.g. read, write, append, or update)
- After completing low-level I/O, a file should be closed using **fclose**

Binary Files

- By default, **fwrite** writes the values from a matrix in column order as 8-bit unsigned integers (uint8).
- By default, **fread** reads a file 1 byte at a time, interpreting each byte as a uint8.
- **fread** creates a column vector output, with one element for each byte in the file. The values in the output vector are of type double.

Binary Files

```
>> x = [1:3;4:6]

x =

     1     2     3
     4     5     6

>> fid = fopen('temp.bin','w');
>> fwrite(fid, x);
>> fclose(fid);
>> fid = fopen('temp.bin');
>> y = fread(fid)

y =

     1
     4
     2
     5
     3
     6

>> fclose(fid);
```

Binary Files

- To read only a specified number of values, or import into a 2D matrix, a size argument to **fread** can be added:

```
>> fid = fopen('temp.bin');
>> y = fread(fid, 4)

y =

     1
     4
     2
     5

>> frewind(fid);
>> y = fread(fid, [2,2])

y =

     1     2
     4     5

>> frewind(fid);
>> y = fread(fid, [2,inf])

y =

     1     2     3
     4     5     6

>> fclose(fid);
```

Binary Files

- To write and read with a different precision from uint8, add a precision string argument:

```
>> fid = fopen('temp.bin');
>> y = fread(fid, 4)
>> xpi = pi*x

xpi =

    3.1416    6.2832    9.4248
   12.5664   15.7080   18.8496

>> fid = fopen('xpi.bin','w+');
>> fwrite(fid, xpi, 'double');
>> frewind(fid);
>> y = fread(fid, [2,inf], 'double')

y =

    3.1416    6.2832    9.4248
   12.5664   15.7080   18.8496

>> frewind(fid);
>> y = fread(fid, [2,inf], 'single')

y =

  1.0e+038 *

    0.0000    0.0000    0.0000    0.0000    2.3811    2.3811
    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000
```

End of Part 1
Please start on Exercises 15 and 16

Part 2
Performance & Debugging

MATLAB Performance

MATLAB is an interpreted computing system.

- The code is not generally compiled into a standalone program.
- The code is interpreted on the fly at run-time.
  - Advantages: It has rapid development time and MATLAB GUI facilities.
  - Disadvantages: It runs slower than compiled code, particularly when it is not optimized.

Now, we are going to investigate ways to **monitor** and **improve the performance** of a MATLAB program.

Stopwatch Timing

- To check the speed of a particular section of code we can use the stopwatch timer functions: **tic** and **toc**.
- **tic**: initialises the stopwatch timer
- **toc**: follows a ‘tic’ call, and returns the time elapsed since the last call to ‘tic’.
- Wrapping any code with a tic-toc pair reveals how long that code takes to run. For small programs, the average mean time is more informative.

- Example:

```
>> tic
>> for i = 1:1000
>>     r = rand(500) * i;
>> end
>> t1 = toc / 1000
```

The Profiler

- The profiler utility:
  - is useful to work out where **bottlenecks** are in a program, particularly on large programs with many components.
  - **generates a report** that shows a breakdown of which components of a program take up most time during execution.
  - **Records information** about execution time, number of calls, parent functions, child functions, code line hit count, and code line execution time.

The Profiler

- The profiler is controlled using the profile function.
  - **profile on**: to initialise the profiler
  - **profile viewer**: to see the results
- Syntax and operation are explained in detail in the help files.
- To focus on function performance, GUI output from the program should be minimised.

Quick Tips to Improve Performance

- **load** and **save** are faster than low-level file I/O functions.
- Use **logical indexing** to quickly access non-contiguous matrix elements
  - e.g. A(isnan(A)) = 0
- Don’t change a **variable’s data type**. When converting, create a new variable.
  - x = 10;
  - x = num2str(x); vs. xs = num2str(x);

Quick Tips to Improve Performance

- Previously we explored how **matrices** can be **dynamically increased in size** via concatenation or out-of-bounds assignment.
  - This is useful, but repeated resizing carries a memory management cost and it can slow a program down.
- Solution – Pre-allocation: work out the maximum size of a matrix and allocate the required amount of space in advance.
  - This is especially relevant in *for* loops, where the size of a variable is increased by a fixed amount at each loop repeat.
  - Since the number of repeats in a *for* loop is known, it is better to pre-allocate the variable.

Quick Tips to Improve Performance

```
>> clear; tic
x = [];
for i = 1:10000
    x = [x,rand(10,1)];
end
toc

clear; tic
x = [];
for i = 1:10000
    x(:,i) = rand(10,1);
end
toc

clear; tic
x = zeros(10, 10000);
for i = 1:10000
    x(:,i) = rand(10,1);
end
toc
Elapsed time is 1.130610 seconds.
Elapsed time is 0.045281 seconds.
Elapsed time is 0.012945 seconds.
```

Quick Tips to Improve Performance

- Vectorization
  - We already mentioned that **nested loops** can be very slow.
  - MATLAB can perform vector and matrix operations very efficiently using built-in code.
  - ***Vectorization*** means converting loops into equivalent vector or matrix operations
  - They can result in significant speed-ups that tend to increase as code complexity increases.

End of Part 2
Please start on Exercise 17