Introduction to Matlab

Part 2

Luke Dickens

Imperial College London

January 15th, 2015

Reminder

This morning, we looked at:

- Overview of Matlab
- Vector and matrix construction
- Arithmetic with vectors and matrices
- Commonly used built in functions. Including:
  - Special functions, e.g. `help`
  - Operations on scalars, vector and matrices
- A focus on plotting

This Lecture

Today we are going to look at:

- Managing input and output, including `mat` files
- Writing scripts and functions. Including:
  - Flow control, i.e. looping and branching
  - Defining functions in-line
  - Input and output for functions
  - Sub-functions
  - Function handles
- A brief overview of non-array types and managing larger projects, namely:
  - Cells and Structs
  - `addpath` and packages
  - Classes

The `diary` Function

All or part of a MATLAB session (commands and output) can be recorded in text format, with the `diary` command.
To do so type

```
>> diary <filename>
```

At the **beginning** of the session you wish to record, and

```
>> diary off
```

at the **end**.

This stores your commands in the file `<filename>`.

`save` and `load`

The commands `save` and `load` can be used to save and load variables in files called a *Mat-files*. Mat-files are binary format and always have the `.mat` extension. Examples of usage are below:

`save somedata.mat x y`	Saves variables `x` and `y` in the file `somedata.mat`
`save moredata x y z`	Saves variables `x`, `y` and `z` in the file `moredata.mat`
`save xdata.dat x -ascii`	Saves `x` in 8-digit ASCII format in the file `xdata.mat`
`save`	Saves entire workspace in the file `matlab.mat`
`load somedata`	Loads variables stored in the `somedata.mat` file.
`load`	Loads variables stored in the `matlab.mat` file.

All these commands work on files in the current directory.

More on input and output

The command `load` can also be used to load ASCII data files, provided the data file contains only a rectangular matrix of numbers.

Data can be imported from a variety of file formats, e.g.

`dlmread`	Read ASCII files with special delimiters
`xlsread`	Import spreadsheet data, e.g. xls, odt files
`imread`	Import image data, e.g. jpg, png, tiff files
`auread`	Import audio data, e.g. au files
`importdata`	A more general import function
`iofun`	A more general import function
`uiimport`	Open Import Wizard to import data

Equivalent functions are available for writing data. Again, more information can be found in the `help` file or look at:

`www.mathworks.co.uk/help/techdoc/import_export/bst9qgh.html`

M-files

We can group sequences of commands in a file, and call it from within MATLAB. Called *m-files*, their filename extension must be ".m". There are two types:

Script files
A sequence of MATLAB commands that are executed (in order) once the script is called. E.g., for the script file `compute.m` , executing the command `compute` will cause the statements in that file to be executed.

Function files
Allow user's to define commands with input and/or output. These have the same status as other MATLAB commands.

Flow Control: Loops

For scripts and functions, we may need some flow control.

`for` blocks have the form:

```
for i = m:n
    ...
end
```

This iterates over elements `i` in the vector $[m : n]$.

`while` loops are similar and have the form:

```
while (boolean_statement)
    ...
end
```

And will loop until the statement returns false.

Flow control can be used at the command prompt, but is particularly useful in m-files.

Flow Control: If

Another useful form of *flow control* is the `if` block. This has the form

```
if (boolean_statement)
    ...
elseif (another_boolean_statement)
    ...
else
    ...
end
```

With 0 or more `elseif`, and 0 or 1 `else` blocks.
May be better to use `switch`, `case` and `otherwise` statements.

Flow Control: Nesting

As you might expect, these control flow blocks can be nested, e.g.

```
for j = 1 : n
    for k = 1 : n
        if (x( j , k ) > 0.5)
           x( j , k ) = 1.5;
        end
    end
end
```

Indentations are for clarity only.

Scripts

An M-file called magicrank.m contains the following code:

```
% Investigate the rank of magic squares
r = zeros(1,32);
for n = 3:32
    r(n) = rank(magic(n));
end
r
bar(r)
```

This should be stored in the local directory from which you launch MATLAB.
Executing `magicrank` runs this script, computes rank of first 30 magic squares, and plots bar chart of results

Anonymous Functions

We can define Anonymous Functions with the `@(`*args*`)` *expression* syntax. For example, to define a function for converting temperatures from Fahrenheit to Celsius use

```
>> convertfahr = @(x) (x-32)/9*5;
```

We can then use `convertfahr` as a normal function, e.g.

```
>> convertfahr(32)
ans =
  0
>> convertfahr(60)
ans =
  15.5556
```

You may find it useful to define lightweight functions at the command line, or within a script or function file.

An Example Function File

For more complex functions, write them in an m-file, e.g. store the following text in a file named `mylog.m`:

```
function [a] = mylog(x,b)
%  [a] = mylog(x,b) - Calc. the base b log of x.
    a = log(abs(x))./log(b);
% End of function
```

This can be called from the same directory with

```
>> mylog(x,b)
```

It returns the base `b` logarithm of value `x`.

Functions

Every MATLAB function begins with a header (first line), which consists of the following :

1. the word function,
2. the output(s) in square brackets, (our variable `a`)
3. the equal sign,
4. the name of the function, which **must** match the function filename (`mylog`)
5. the input(s) (our variables `x` and `b`).

The next block of *commented* lines describe the function and are output when you execute `help mylog`.
Finally, all arguments specified in the header, must be assigned during the function.
For us this is the single line starting `a=...`

Another Example

```
function [x] = gaussel(A,b)
% [x] = gaussel(A,b)
% Performs Gaussian elimination and back substitution
% to solve the system Ax = b.
% INPUT : A - matrix for the left hand side.
% b - vector for the right hand side
% OUTPUT : x - the solution vector.
N = max(size(A));
% Gaussian Elimination
 for j=2:N,
 for i=j:N,
 m = A(i,j-1)/A(j-1,j-1);
 A(i,:)  = A(i,:)  - A(j-1,:)*m;
 b(i) = b(i) - m*b(j-1);
 end
 end
% Back substitution
 x = zeros(N,1);
 x(N) = b(N)/A(N,N);
 for j=N-1:-1:1,
 x(j) = (b(j)-A(j,j+1:N)*x(j+1:N))/A(j,j);
 end
% End of function
```

Sub-Functions

Subfunctions can be defined within a function file, and :

- Sub-functions can only be called from within the m-file.
- Each sub-function begins with its own function definition line.
- The functions immediately follow each other.
- Sub-functions can occur in any order, as long as the primary function appears first.

A sub-function example

```
function [avg, med] = newstats(u) % Primary function
% NEWSTATS Find mean and median with internal functions.
n = length(u);
avg = mean(u, n);
med = median(u, n);

function a = mean(v, n) % Subfunction
% Calculate average.
a = sum(v)/n;

function m = median(v, n) % Subfunction
% Calculate median.
w = sort(v);
if rem(n, 2) == 1
 m = w((n+1) / 2);
else
 m = (w(n/2) + w(n/2+1)) / 2;
end
```

Function Handles

A function handle can be passed into and returned from functions, by using the `@` operator, either using the function name, i.e.

```
>> handle = @functionname
```

or anonymously, i.e.

```
>> handle = @(arglist)anonymous_function
```

Function handles are then called just like functions.

A function handle example

Define the `funcplot.m` file as follows:

```
function [] = funcplot(fhandle,xcoords)
% Takes function handle and list of x-coordinates
% and plots curve of function
plot(xcoords,fhandle(xcoords))
```

Then you can call this with:

```
>> funcplot(@sin,linspace(-pi,pi,51))
```

Applications

MATLAB provides many libraries for you to use and explore. A short list of some available application areas are:

Linear Algebra Including gaussian elimination, eigenvectors and -values, and matrix factorisation.

Curve Fitting and Interpolation Including linear and polynomial curve fitting, least squares curve fitting, non-linear fits, and interpolation.

Numerical Integration For when symbolic solutions are not available or intractable.

Ordinary Differential Equations For solving systems of first order differential equations (ODEs).

There are many more available.

Example: Polynomial Curve Fitting

We look at curve fitting in a bit more detail.
We have a pair of vectors, `x` and `y`, representing the independent and dependent variables of a (possibly noisy) relationship. We want the curve that most closely fits the data.
With the `polyfit` function, we try to fit the polynomial

$$y = a_k x^k + a_{k-1} x^{k-1} + \ldots + a_2 x^2 + a_1 x + a_0$$

to the data using command

```
>> as = polyfit(x,y,k)
```

The vector `as` contains the coefficients a_i, $i = 1 \ldots k$.
We create a new set of y points for our approximate curve with

```
>> yapprox = polyval(as,x)
```

Curve Fitting in Action

To illustrate this, we create some noisy data then try to fit it.

```
x = linspace(0,2*pi,100); % the x values
r = randn(1,100); % the noise
y = sin(x) + 0.1*r; % noisy y values
as = polyfit(x,y,1); % fit a straight line
y1 = polyval(as,x);
as = polyfit(x,y,2); % fit a quadratic curve
y2 = polyval(as,x);
as = polyfit(x,y,3); % fit a cubic curve
y3 = polyval(as,x);
plot(x,y,'xk',x,y1,'b',x,y2,'g',x,y3,'r')
 % plot them
```

Cells

Matrices can be a bit restrictive. For instance, they cannot hold matrices with different dimensions. E.g.

```
>> m = [[1]',[1,2]',[1,2,3]']
???  Error using ==> vertcat
CAT arguments dimensions are not consistent.
```

A `cell` is a matrix, such that the elements can be of different sizes or types. It is constructed like a matrix but using curly brackets, e.g.

```
>> c = {[1]',[1,2]',[1,2,3]'}
c =
  [1] [2x1 double] [3x1 double]
```

Elements are assigned like in a matrix but using curly brackets, e.g.

```
>> c{4} = [1,2,3]
  [1] [2x1 double] [3x1 double] [1x3 double]
```

Cell elements are read in the same way, e.g. `c{4}`.

More on Cells

Other ways to construct cells include:

`c = cell(d)`	creates a cell array of empty matrices. If `d` is a scalar, C is `d`×`d`. If `d` is an *n*-vector, C is `d(1)`×...×`d(n)`.
`c = cell(a,b)`	for scalars `a` and `b`, creates a `a`×`b` cell array of empty matrices.
`c{a,b} = []`	for scalars `a` and `b`, creates a `a`×`b` cell array of empty matrices.

Cells are particularly useful for storing lists of different lengths, e.g. strings, but are more general. You can even have cells of cells.
See `help` or online at:
`http://www.mathworks.co.uk/help/techdoc/ref/cell.html`

Structs

A `struct` is a data type that groups related data using data containers called fields. Each field can contain data of any type or size and is assigned using the dot (`.`) operator. For instance,

```
>> student.name = 'Tommy Atkins';
>> student.scores = [75,42,81,53]
student =
 name:  'Tommy Atkins'
 scores:  [75 42 81 53]
```

Fields can also be referenced with the dot operator, e.g.

```
>> student.name
ans =
Tommy Atkins
```

Structure Arrays

More generally, the `struct` command can be used to create structure arrays. This is useful for storing rows of data of potentially different types. For example, try:

```
names = { 'John', 'Paul', 'George', 'Ringo'}
birth_years = { 1940, 1942, 1943, 1940 }
roles = { 'Lyrics/Vocals/Guitar', 'Lyrics/Vocals/Rhythm
Guitar', 'Lead Guitar/Vocals/Lyrics', 'Drums'}

band = struct('name', names, 'birth_year', birth_years,
'role', roles)
```

For more on structures and structure arrays please see:
`http://www.mathworks.co.uk/help/techdoc/matlab_prog/br04bw6-38.html`

Addpath and Packages

As every function needs its own file, your file-system can quickly become very messy. Two ways to structure your code:

- With the `addpath` command:
  - Group similar m-files in their own folder
  - Add the folder's path with `addpath`
- With packages:
  - Put m-files in sub-folder whose name starts with +, e.g. `+mycode`
  - Call files from parent with `<folder_name>.<func_name>` (without the +), e.g. mycode.myfunc.

For large projects, you may want to use these in combination.

Classes

- MATLAB also allows us to define classes within their own m-files.
- In MATLAB, class instances can store data and have their own methods, as well has generating and responding to broadcast events.
- The details of this can again be found in the online help.
- If you think you need to define your own classes, try starting here:

  `http://www.mathworks.co.uk/help/techdoc/matlab_oop/brh2rgw.html`

The file `dict.m` in the lab materials, includes an example of a class file. This defines a dictionary type, which allows you to store objects referenced by strings.

Summary

This course has looked at:

- Using basic arithmetic and comparison operations with scalars, vectors and matrices.
- Using built-in functions for common operations, and looking up help files.
- Loading and saving variables, and recording MATLAB sessions.
- Plotting 2-d and 3-d graphs.
- Writing script and function m-files.
- Using the curve fitting libraries (very briefly).
- Using the non-array types `cell` and `struct`.
- Managing larger projects

Further Help and Support on MATLAB

MATLAB Help

`http://www.mathworks.com/help/`

MATLAB Suppport

`http://www.mathworks.com/support/`

The above web page includes a link to MATLAB based books.

See also

`http://www.mathworks.com/support/books/index.jsp`

Or contact me at:

`luke.dickens@imperial.ac.uk`

References

This course has been partly developed from material found in the following sources.

- Getting Started with MATLAB7: A quick introduction for scientists and engineers, Rudra Pratap. Oxford University Press, 2006.
- A Beginner's Guide to MATLAB, Christos Xenophontos. Department of Mathematical Sciences. Loyola College. 2005 version.
- Introduction to Scientific Programming in Matlab, Guy-Bart Stan (lecture slides). Imperial College. 2010.