Matlab

More Advanced Matlab

By Dawn Walker

(Accompanying slides developed by August Vouros)

In this session we will cover some features of Matlab that you will need to know about in order to understand an agent based model that you will be using in the next sessions. These are:

1. **Global data**
2. **Data Structures and cell arrays**
3. **Object Oriented Code**

It is important that you read the information in this document carefully.

1. Global data

Recall how we define a Matlab function

function [output1, output2, ..output n] = functionname(input1, input2, ….inputn)

Normally, your function will only be able to 'see' variables that are passed into it via the *input argument list*, to the right of the function name. Similarly, in order to be able to access variables created by the function, you need to declare them in the *output argument list*.

A way around this is to declare a variable as *global*. This is done by using the *global* declaration, followed by the name of the variable, or the list of variables, that you wished to be global. The variable can then be 'seen' (and modified) inside any function where it is declared as global, without having to pass it via the input or output list.
E.g. in your workspace type:

```
>> global X
>> X=10;
>> a=2;
>> b=3;
```

Now create and save the following function:

```
function demo_global(a,b)

global X

X=a+b;
```

d.c.walker@sheffield.ac.uk

Run this function from the command line and check to value of X:

```
>> demo_global(a,b);
>> X

X =

    5
```

The value of X has changed to 5 (=a+b) even though it was not declared as an input or output to demo_global.

Note that declaring a variable as global and assigning a value(s) to the variable are two separate steps

Notes on using global variables

- Global declarations are useful for small numbers of significant variables that are required by many different functions in a programme e.g. environmental variables in an agent model.
- Declaring variables as global can save on the computational overhead in passing large collections of data between functions, but should not be used as a method of avoiding declaring function inputs and outputs. Programmes with too many global variables can become complicated and difficult to debug.
- Any type of data can be defined as global if required (see later sections!).
- The convention is to use CAPITALISED variable names to represent global variables. This makes your programme easier to follow.

2. Data structures and cell arrays

So far you have come across some very simple types of data; essentially we have looked at different ways of storing numeric data – as single values or in 1D or 2D arrays. In order to do some more sophisticated programming, we need to look at some different data types, and more complicated ways of grouping data together.

As well as storing numeric data, Matlab can also store strings of characters e.g.

```
>> name='bob'
name =
bob

>> whos name
  Name      Size                 Bytes  Class

  name      1x3                      6  char array
Grand total is 3 elements using 6 bytes
```

In some circumstances, it is useful to group together different types of data relating to a particular item or idea. We can't do this using a standard Matlab array, as these will only handle numeric data. Instead, we can use a *data structure*.

Suppose that for some reason we were interested in cars driving down a particular road at a given time. We're interested in the type, make and model of each vehicle, its registration, and how fast it's travelling. We want to group together the data relating to each vehicle into a single 'package'.

The first vehicle we catalogue is a Ford focus car, registration MY CAR1 travelling at 33 mph. We could represent this information as follows:

```
car=struct('make','ford','model','focus','reg','MY CAR1','speed',33)

car =

    make: 'ford'
   model: 'focus'
   reg: 'MY CAR1'
   speed: 33
```

This is a data structure containing both numerical data and character strings.
The syntax for creating a data structure in Matlab is:

```
S = struct('field1',VALUES1,'field2',VALUES2,...)
```

Type 'help struct' for more information.

Suppose the next vehicle is a van. We also want to record the company the van belongs to, so in this case:

```
van=struct('make','robin','model','reliant','reg','MY VAN1','owner','cowboys co.','speed',37)
van =

    make: 'robin'
   model: 'reliant'
     reg: 'MY VAN1'
   owner: 'cowboys co.'
   speed: 37
```

If we type 'whos':

```
>> whos
  Name      Size                Bytes  Class

  car       1x1                 536  struct array
  van       1x1                  688  struct array
```

so we now have two separate variables in our environment: one representing a car, the other a van.

In order to see the names and parameters contained within an existing data structure, you need to use the inbuilt function ***struct*** with just one input parameter:

```
>>struct(car)

ans =

    make: 'ford'
   model: 'focus'
     reg: 'MY CAR1'
   speed: 33
```

Data inside the structure can be accessed in the following way, using a dot '.':

```
>>current_speed=car.speed

>> current_speed=33

>>type=car.make

>> type='ford'
```

We are planning to monitor the traffic on this stretch of road on a number of separate occasions. We want to group together data relating to the vehicles that we observe on each occasion, and we also want to record the order in which we observe the vehicles. We could do this by naming our data structures car1, van2 etc, but a more convenient way of doing this is to use a ***cell array***, as follows:

```
>>Traffic_May1{1}=car;
>>Traffic_May1{2}=van

>>Traffic_May1 =

   [1x1 struct]    [1x1 struct]
```

Traffic_May1 is a cell array containing 2 cells – the first cell contains the data relating to car 1 seen on May 1st, the second the structure representing the van.

Cell arrays are special types of array that can store more than one sort of data at a time and are extremely useful for constructing programmes where we want to consider discrete objects. **When entering data in a cell array, be careful to use curly brackets {}, not normal round ones ().**

We can access data inside the cells using:

```
>> first_car=Traffic_May1{1}

first_car =

    make: 'ford'
```

```
   model: 'focus'
   reg: 'MY CAR1'
   speed: 33
```

We do not have to go through the two separate steps of creating a data structure, then putting it into a cell array – we can do both at once e.g.

```
>> Traffic_May1{3}=struct('make','citroen','model','CV','reg','MY CAR6','speed',23)

Traffic_May1 =

   [1x1 struct]    [1x1 struct]    [1x1 struct]
```

Task 1: Read through the following sections in the Matlab help guide:

- **Structures**
- **Cell arrays**

In the case of an agent model, it makes sense to package the information associated with every agent into a data structure. This is discussed in more detail below. We can also package the data associated with the model environment in the same way, e.g.:

```
ENVIRONMENT=struct('shape','square','units','kilometres','size',100,'veg',[])

ENVIRONMENT =

   shape: 'square'
   units: 'kilometres'
   size: 100
   veg: []
```

we could later define *veg* to be, for example, a 100 x 100 matrix or array containing the level of food distribution in each km^2 of the environment.

----

3. Object-oriented Code structure

Consider an individual-based (agent-based) representation of a predator-prey system. Our model will consist of a type of agent that represents predators (in this case, foxes) and a second type representing prey (rabbits). In this case, both our agent types will require similar parameter sets (though this might not necessarily be the case for every system we might want to model). Also, both sets of agents will have rules defining similar behaviour types - breeding, eating and so on, but the nature of these rules might differ. For instance, consider a rule which represents eating: at each iteration, rabbits might eat a certain amount of vegetation within a given radius of their current position. However, foxes don’t eat vegetation – foxes eat rabbits! Whereas rabbit agents will be able to detect the vegetation in their vicinity, foxes will need to identify where the nearest rabbits (prey) are, then make some decision about which, if any, to eat. There is a fundamental difference in how this rule works for the different agent groups.

Using traditional programming (sometimes called *procedural* programming) we could handle this situation using the following methodology:

```
for agent m = 1:n
    if agent m is a fox
        call function fox_eat
    else if agent m is a rabbit
        call function rabbit_eat
```

and so on.

This coding methodology would work perfectly well. However, rather than defining two completely different sets of functions with different names, it is much neater to use an ***object-oriented*** programming approach. This allows us to define two separate classes of agents – a fox class and a rabbit class. Every data structure which we create to represent either a fox or rabbit is then defined as belonging to one or the other classes. It will have the memory parameters defined for that class, and have access to the functions which as designed specifically for that class. It will be an *instance of a class* which is often referred to as an *object.*

There are several examples of object-oriented programming languages, all of which lend themselves to agent-based modelling. Examples are Java, Python and C++. However, Matlab is used in this course as it is a language that includes the facility to develop object-oriented programs, but is also optimised for numerical modelling (i.e. ODEs).

Object oriented programming in Matlab involves using the *classdef* command to define a class definition function. Within this function, 'properties' (the memory parameters that define that class), and any number of 'methods' (functions that can access and modify the class properties) are included.

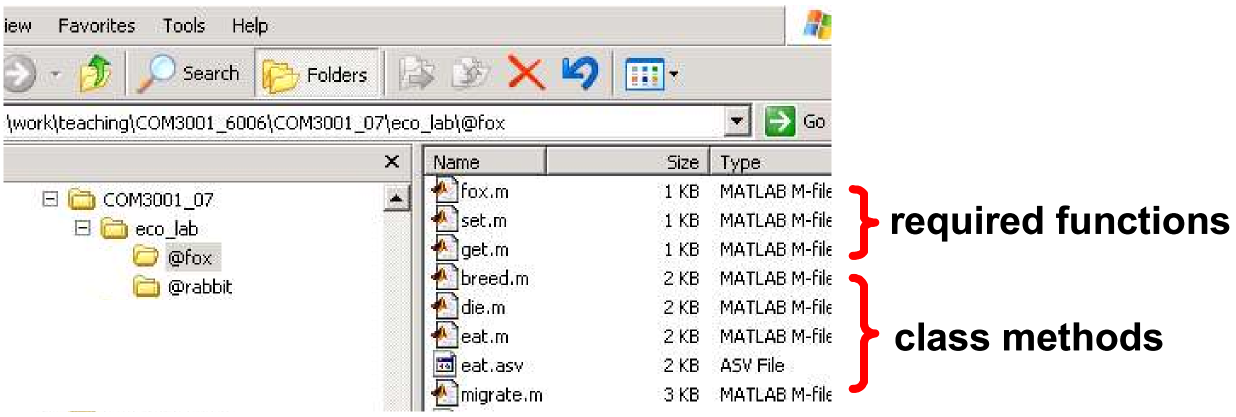
For ease of development Matlab allows you to organise classes in separate folders, the contents of which define a class package. For instance, we might create a folder to build our fox-rabbit application in and name it *eco_lab*. Underneath this folder we create two class directories called @fox and @ rabbit (the "@" as the first character in the directory name is required to tell Matlab this is an object class package folder). Each class directory is then required to contain at least the following:

- The class template function. This starts with the *classdef* statement and must have the same name as the relevant class name e.g. rabbit.m or fox.m. It will contain the list of class properties and the class constructor method (a default member function with the same name as the class that initialises class property values when a new class object is created.

You also need a number of other functions in each class directory. These are:

- set.m – allows you to set new parameter values in each object
- get.m – allows you to extract parameter values from each object

Note for programmers - in recent versions of Matlab, dedicated *set* and *get* function types have been included to do this within the member functions in the class definition file using: function obj = set.*PropertyName*(obj,value) . However, the method of defining classes described in his document is also supported.



\work\teaching\COM3001_6006\COM3001_07\eco_lab\@fox

Name	Size	Type
fox.m	1 KB	MATLAB M-file
set.m	1 KB	MATLAB M-file
get.m	1 KB	MATLAB M-file
breed.m	2 KB	MATLAB M-file
die.m	2 KB	MATLAB M-file
eat.m	2 KB	MATLAB M-file
eat.asv	2 KB	ASV File
migrate.m	3 KB	MATLAB M-file

Figure 2 Directory structure and key functions required for a Matlab object-oriented model

As well as these, you have a list of mfile functions which define your rule sets. For instance, both the fox and rabbit folders will contain a function called eat.m, but the code inside these functions could be completely different. Matlab automatically treats these files as class member functions, as they are within the class package directory. A constraint exists such that each of these functions must take an instance of that class (an 'object') as its first input i.e .the first line inside the mfile will be:

function new_obj=eat(old_obj,par1, par2, par3….)

if *old_obj* is a rabbit object, then eat.m in the rabbit folder will be automatically called. Conversely, if *old_obj* is a fox, then the fox version will be called. The other input parameters (par1, par2 etc) can be anything.

Object oriented programming contains other features that could be useful for modelling agent-based systems. For instance, ***inheritance*** allows the definition of child classes that can inherit selected methods from the parent class. This feature might be of interest if you were interested in building a multi-scale agent system (we will not use inheritance as part of this course).

Note: If you're a non-computer science student and you're confused by object oriented programming **DON'T WORRY**! You will not be required to write this sort of programme on your own!

Task2 :Read Matlab help material on Object Oriented Programming using Matlab:
Go to http://www.mathworks.co.uk/help/matlab/object-oriented-design-with-matlab.html
Then browse through the links.

Task3: Download and unzip the *Ecolab* model from MOLE.
Read the user guide and have a go at running the model from the command line.

This is an example of an agent-based model that your group may wish to adapt for use in your assignment. In the next lab session, your group will carry out a simple task using this model, to get you used to running and editing the code.