KICK OFF MATLAB

Marco Aiolfi Antonio Falanga

Programming

1. M-Files

Files containing Matlab® codes are called M-Filles and can be classified in:

- *Function M-Files:*
  Files that accept input arguments and produce outputs internal variables are local to the function by default. These are useful for extending the Matlab® language for your application.
- *Script M-Files:*
  Do not accept input arguments or return output arguments and operate on data in the workspace. These are useful for automating a series of steps you need to perform many times.

In order to create an M-file you simply need a text editor even if the easiest way is to use the Matlab® Editor by typing edit in the command line. To create a script file just open the Matlab® editor, write a list of statements, save the file with a name and ".m" extension and type the filename in the command window to run it. All the variables created in a script file remain in the workspace.
A function M-file consists of:

- *The Function Definition Line*: informs Matlab® that the M-file contains a function, and specifies the argument calling sequence of the function. To create a function m-file open the editor, define the function in the following way:
  function [output1,output2]=functionname(input1,input2)

- The H1 Line: so named because it is the first help text line, is a comment line immediately following the function definition line. This is the first line of text that appears when a user types *help functionname* at the Matlab® prompt. Further, the *lookfor* function searches on and displays only the H1 line.

- Help Text: When you type *help functionname*, Matlab® displays the comment lines that appear between the function definition line and the first non-comment (executable or blank) line.

- The Function Body: contains all the Matlab® code that performs computations and assigns values to output arguments.

- Comments: comment lines begin with a percent sign (%). Comment lines can appear anywhere in an M-file function

```
[output1,output2,...]=functionname(input1,input2,...)

% This is H1 line. Describe the general purpose of your function
% This is the Help text
% Inputs:
% input1: describe
% input2: describe
% Outputs:
% output1: describe
```

A note of caution in naming files: it's easy to get unexpected results if you give the same name to different functions or if you use a name given by Matlab® to another function. At this point you can start writing the body of the function, defining how do you use the inputs to get the outputs. Once you finished, save the function with the same name you used when declaring the function.
The following function evaluates and plots the function y = 2 + 3x - x2 for a given x

```
function y=myfun(x)
% evaluates and plots the function y=2+3*x-x^2
y=3*x-x.^2+2;
plot(x,y)
```

We can combine script m-file and function m-file. For example the following script defines a vector x, calls the previous function *myfun* and then save the data:

```
%example1
x=[-10:.5:10];
y=myfun(x); %calls the function myfun
save marco y
```

Furthermore function M-files can contain other functions. If the function called inside an existing (primary) function is used only by the latter, then you can define a subfuction only visible to the primary function or other subfunctions in the same file. To do this just defines the secondary function at the end of the primary function as follows:

```
function avg = newstats(u) % Primary function
% NEWSTATS Find mean with internal functions.
n = length(u);
avg = mean(u,n);
% Subfunction
function a = mean(v,n);
% Calculate average.
a = sum(v)/n;
```

Matlab® uses a search path to find M-files and other Matlab® related files, which are organized in directories on your file system. These files and directories are provided with Matlab® and associated toolboxes. Any file you want to run in Matlab® must reside in a directory that is on the search *path* or in the current directory. By default, the files supplied with Matlab® and MathWorks toolboxes are included in the search

path. If you create any Matlab® related files, add the directories containing the files to the Matlab® search path. Alternatively you have to switch to the directory where the m files are stored.

2. Flow Control

Matlab® has several flow control constructs:

- "if"
- "switch and case"
- "for"
- "while"
- "continue"
- "break"
- "try - catch"
- "return"

if

The if statement evaluates a logical expression and executes a group of statements when the expression is *true*. The optional *elseif* and *else* keywords provide for the execution of alternate groups of statements. An *end* keyword, which matches the *if*, terminates the last group of statements. The groups of statements are delineated by the four keywords — no braces or brackets are involved.
The Matlab® algorithm for generating a magic square of order *n* involves three different cases: when *n* is odd, when *n* is even but not divisible by 4, or when *n* is divisible by 4. This is described by

```
if rem(n,2) ~= 0
M = odd_magic(n)
elseif rem(n,4) ~= 0
M = single_even_magic(n)
else
M = double_even_magic(n)
End
```

In this example, the three cases are mutually exclusive, but if they weren't, the first *true* condition would be executed. It is important to understand how relational operators and *if* statements work with matrices. When you want to check for equality between two variables, you might use

```
if A == B, ...
```

This is legal Matlab® code, and does what you expect when A and B are scalars. But when A and B are matrices, A == B does not test *if* they are equal, it tests *where* they are equal; the result is another matrix of 0's and 1's showing element-by-element equality. In fact, if A and B are not the same size, then A == B is an error. The proper way to check for equality between two variables is to use the *isequal* function,

```
if isequal(A,B), ...
```

Here is another example to emphasize this point. If A and B are scalars, the following program will never reach the unexpected situation. But for most pairs of matrices, including our magic squares with interchanged columns, none of the matrix conditions A > B, A < B, or A == B is true for *all* elements and so the else clause is executed.

```
if A > B
'greater'
elseif A < B
'less'
elseif A == B
'equal'
else
error('Unexpected situation')
end
```

Several functions are helpful for reducing the results of matrix comparisons to scalar conditions for use with if, including

isequal
isempty
all
any

switch and case

The *switch* statement executes groups of statements based on the value of a variable or expression. The keywords *case* and otherwise delineate the groups. Only the first matching case is executed. There must always be an *end* to match the *switch*.

The logic of the magic squares algorithm can also be described by

```
switch (rem(n,4)==0) + (rem(n,2)==0)
case 0
M = odd_magic(n)
case 1
M = single_even_magic(n)
case 2
M = double_even_magic(n)
otherwise
error('This is impossible')
end
```

for

The *for* loop repeats a group of statements a fixed, predetermined number of times. A matching *end* delineates the statements.

```
for n = 3:32
r(n) = rank(magic(n));
end
```

It is a good idea to indent the loops for readability, especially when they are nested.

```
for i = 1:m
        for j = 1:n
                H(i,j) = 1/(i+j);
        end
end
```

while

The *while* loop repeats a group of statements an indefinite number of times under control of a logical condition. A matching *end* delineates the statements.

Here is a complete program, illustrating *while, if, else,* and *end*, that uses interval bisection to find a zero of a polynomial.

```
a = 0; fa = -Inf;
b = 3; fb = Inf;
while b-a > eps*b
        x = (a+b)/2;
        fx = x^3-2*x-5;
        if sign(fx) == sign(fa)
                a = x; fa = fx;
        else
                b = x; fb = fx;
        end
end
```

continue

The *continue* statement passes control to the next iteration of the for loop or while loop in which it appears, skipping any remaining statements in the body of the loop. In nested loops, *continue* passes control to the next iteration of the *for* loop or *while* loop enclosing it.

The example below shows a continue loop that counts the lines of code in the file *magic.m*, skipping all blank lines and comments. A continue statement is used to advance to the next line in *magic.m* without incrementing the count whenever a blank line or comment line is encountered.

```
fid = fopen('magic.m','r');
count = 0;
while ~feof(fid)
        line = fgetl(fid);
        if isempty(line) | strncmp(line,'%',1)
                continue
        end
count = count + 1;
end
disp(sprintf('%d lines',count));
```

break

The *break* statement lets you exit early from a *for* loop or *while* loop. In nested loops, break exits from the innermost loop only.

Here is an improvement on the example from the previous section. Why is this use of break a good idea?

```
a = 0; fa = -Inf;
b = 3; fb = Inf;
while b-a > eps*b
        x = (a+b)/2;
        fx = x^3-2*x-5;
        if fx == 0
                break
        elseif sign(fx) == sign(fa)
                a = x; fa = fx;
        else
                b = x; fb = fx;
        end
end
```

try - catch
The general form of a *try-catch* statement sequence is

```
try
statement
...
statement
catch
statement
...
statement
end
```

In this sequence the statements between try and catch are executed until an error occurs. The statements between *catch* and *end* are then executed. Use *lasterr* to see the cause of the error. If an error occurs between *catch* and *end*, Matlab® terminates execution unless another try-catch sequence has been established.

return
return terminates the current sequence of commands and returns control to the invoking function or to the keyboard. *return* is also used to terminate keyboard mode. A called function normally transfers control to the function that invoked it when it reaches the end of the function. You can insert a *return* statement within the called function to force an early termination and to transfer control to the invoking function.

3. Measuring execution time

To compare the speed of different implementations of a program, you can use the *stopwatch* timer functions, *tic* and *toc*. Invoking *tic* starts the timer, and the first subsequent *toc* stops it and reports the time elapsed. If the program executes too fast try measuring the program running repeatedly in a loop:

```
tic
for k = 1:100
        - run the program -
end;
toc
```

As a result you get for example

elapsed_time = 24.0140

4. Structures

Structures are Matlab® arrays with named "data containers" called fields. The fields of a structure can contain any kind of data. For example you can have a structure called results containing different subfields:

```
function results=myfun6(x)
[r,c]=size(x);
if r==1jc==1
type='vector';
else
type='matrix';
end
results.object=type;
results.min=min(x);
results.max=max(x);

>>c=myfun6(magic(3))

c =
object: 'matrix'
min: [3 1 2]
max: [8 9 7]
```

5. Tips and Tricks

- You can improve the speed of your for cycles by preallocating space (for example by using zeros) for storing variables in memory. In the opposite case matlab has to resize a matrix or a vector in each iteration.

- Your code executes more quickly if it is implemented in a function rather than a script. Every time a script is used in Matlab®, it is loaded into memory and evaluated one line at a time. Functions, on the other hand, are compiled into pseudo-code and loaded into memory once. Therefore, additional calls to the function are faster.

- A good step to speeding up your programs is to use the Matlab® Profiler to find out where the bottlenecks are. This is where you need to concentrate your attention to optimize your code. To start the Profiler, select View -> Profiler in the Matlab® desktop. If you don't have Matlab® you can use *tic* and *toc*.

- Take the problem you are trying to solve and break it down into a series of smaller, independent tasks. Implement each task as a separate function. Try to keep functions fairly short, each having a single purpose.

- If you have a function that is called by only one other function, put it in the same M-file as the calling function, making it a subfunction.

- Use debugging to see what's going on step by step inside a function. By setting breakpoints from the editor window you can pause execution of the function so you can examine values where you think the problem might be. If you are interested in econometrics you must give a look at the website www.spatial-econometrics.com. from which you can download a freeware econometrics toolbox for Matlab®.