Scientific Computing Lecture 2: Logical operations, `m`-files and functions

InFoMM CDT
Mathematical Institute
University of Oxford

Andrew Thompson
(slides by Stuart Murray)

Some new syntax

Comments and semicolons

In this lecture we will be looking at full MATLAB programs

It is useful to be able to add *comments* to a program

Anything on a line following a percent sign is a *comment* and is ignored

```
>> x = 1 % set the value of x to 1. This is a comment
```

Semicolons have two uses: to suppress the display of results:

```
>> x = 1
ans =
     1
>> x = 1;
```

and allow multiple statements on one line:

```
>> x = 1; y = 2; z = 3;
```

Logic

Logical expressions

We have met some variable classes already: string, integer, double precision

MATLAB has another for handling logic: the *logical* class

A logical variable can have the value `true` or `false`

```
>> x = true
x =
     1
>> class(x)
ans =
      logical
```

True and false are also represented by `1` and `0`:

```
>> x = logical(0); % sets x to false
```

Logical expressions: comparison

We can make *logical comparisons* in MATLAB

```
>> 2 > 1
ans =
     1
>> 1 == 0
ans =
     0
```

==	is equal to
~=	is not equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

Boolean operators

MATLAB has symbols for the operations `not`, `or` and `and`:

```
>> true & true
ans =
     1
>> false | true
ans =
     1
>> ~true
ans =
     0
```

XOR (exclusive or) has the function `xor`.

Short circuit operators

MATLAB also offers the operators && together with ||
These are *short circuit operators*
Give identical results to & and |
If the result can be obtained from the lhs, the rhs is not evaluated
An example:

```
>> true || verySlowFunction
>> false && bigDeterminant
```

Boolean operators have an order of precedence like /,*,+,-
Safe to use brackets

```
~       high precedence
&
|
&&
||      low precedence
```

Array logic

All logical expressions covered so far work with arrays *elementwise*

The result is an *array of logical values* (0s or 1s); a *logical array*

Here we see arrays being compared:

```
>> A = [1 2;3 4]; B = [1 2;-3 4];
>> A == B
ans =
     1    1
     0    1
```

We may perform Boolean operations with logical arrays as well:

```
>> a = logical([1 0 0]); b = logical([0 1 0])
>> a | b
ans =
     1    1    0
```

Logical indexing: powerful expressions

We may use a logical array to index another array

Why is this useful?

Suppose we wish to find all numbers in a matrix fulfilling some criteria

e.g. **all the positive entries**

We write an expression whose result is a logical array:

```
>> z = [1 2 -1 0 -4 20 -2];
>> z > 0
ans =
     1     1     0     0     0     1     0
```

Use this array **to index the original array**:

```
>> index = z > 0;
>> z(index)
ans =
     1     2    20
```

It is usually much neater to write a single expression:

```
>> z(z>0)
ans =
     1     2    20
```

A more complicated example: return all the elements that are on the diagonal:

```
>> a = 1:16;
>> A = reshape(a,4,4); % create a 4-by-4 matrix
>> index = logical(eye(4))
>> A(index)
ans =
     1     6    11    16
```

What will the following return from a matrix X?

```
X((mod(X,2)==0) & (X > 0))
```

The `find` function

The find function returns **indices of the nonzero elements of an array**

This is useful to find the *indices* of elements that fulfil certain criteria

Using `find`

```
>> a = [1 0 5 0 -1]
>> find(a)
ans =
     1    3    5
```

Combine find with a logical expression:

```
>> find(a < 0)
ans =
     5
```

The M-file

Getting started

We can write programs or *scripts* for MATLAB

At their simplest these are a list of statements one after another

Written in an M-file, using the `.m` extension

No special structure: simplest program is just a list of statements

A simple code:

```
% simple.m

A = rand(2);
display(eig(A));
```

Editing and running programs

Programs can be created in any text editor: simply save using the .m extension

MATLAB has a very good editor of its own with syntax highlighting

Simply go to `File > New` or use the command line

```
>> edit MyProgram.m
```

Open a file with `File > open` or

```
>> open MyProgram.m
```

Hit F5 to run a program or use

```
>> run MyProgram.m
```

Program flow: `if` statements

We can control whether certain parts of a program are executed

We can make execution conditional using an **`if`** statement

An example: compute a matrix inverse only if matrix is nonsingular:

```
if (abs(det(A)) > eps)
    display(inv(A));
end
```

We can allow the program to follow one of two paths using the **`else`** keyword:

Example: display a warning if the matrix is singular:

```
if (abs(det(A)) > eps)
    display(inv(A));
else
   display('matrix is singular to working precision')
end
```

The `elseif` keyword

The **`elseif`** keyword allows the program to follow one of several branches

Example: display a message about the size of a 2d array

```
% part of a program
x = min(size(A);
if (x==0)
    display('A is empty');
elseif(x==1)
     display('A is a vector');
else
    display('A is a matrix');
end
```

We used **`else`** here to catch all the other possible cases

N.B. spelling of `elseif` vs `elsif` as in some languages (Ruby, Perl)

The switch statement

MATLAB has a **switch** statement that replaces lots of **elseif** statements

We can switch on an integer or string

An example: produce plots depending on user input

```
% section of switching program
plottype = input('what type of plot?');
switch plottype
    case 'line'
        plot(x)
    case 'bar'
        bar(x)
    case 'pie'
        pie3(x)
    otherwise % use to catch other possibilities
        display('unknown plot type')
end
```

Loops

We can repeat sections of code using a **for** loop

Follow **for** with an index equal to a range, to control the number of loops:

```
% generate ten random numbers
for i=1:10
    display(rand);
end
```

The index can be used within the loop:

```
% calculate the ranks of some magic squares and store in v
for i=3:10
    v(i) = rank(magic(i));
end
```

Controlling loops: break

Suppose we only need to repeat a loop under certain circumstances:

An example: iterate the equation $z \longrightarrow z^2 + c$ until $|z| > 2$:

```
% iterator1
z = 0; c = 1 + 1i;
for i=1:1000
    z = z^2 + c
    if (abs(z)>2)
        break
    end
end
```

Controlling loops: `while`

In the last example we kept repeating until some condition was met

MATLAB has a type of loop that repeats while a condition is true: the **`while`** loop

It allows infinite loops if the condition is always true

```
% iterator2
z = 0; c = 1 + 1i;
while (abs(z)<=2)
    z = z^2 + c
end
```

Much neater than using **`if`** and **`break`**

Controlling loops: `continue`

Suppose the body of a loop only needs to run when some condition is met

It would be useful to skip on to the next pass if the condition is not met

The **`continue`** statement skips to the next pass of the loop

An example: display the size of magic squares of rank 3

```
% magic ranks
for i=1:100
    r = rank(magic(i));
    if (r~=3)
        continue
    end
    display(i);
end
```

Function files

Writing your own functions

We may add to the many MATLAB built-in functions

Simply write a function and save in an M-file, e.g MyFunction

Call the function in the normal way

```
>> MyFunction
```

MATLAB searches for the function in the current directory and executes it

Functions are also written in a `.m` file

Function structure

Functions all have the same structure

You can even look at the code for the built-in functions

A skeleton function:

```
function [out1,out2,...] = functionName(arg1,arg2,...)
    statements
      ⋮
    out1 =
    out2 =

end
```

First line is the function signature

Result/output variables are defined within the function

Function ends with an **end** (actually optional, but a good idea)

Simple functions

Example: some simple functions

```
function [] = proclaim()
    display('MATLAB is awesome');
end
```

```
function [xout] = jukowski(xin)
    xout = xin + 1./xin;
end
```

Call one function from another:

```
function [xout] = jukowski(xin)
    xout = xin + 1./xin;
    proclaim();
end
```

Scope

The only variables a function can "see" and use are the input arguments

Any others cannot be seen: they are outside the *scope* of the function

Example of a function invalid in this way:

```
function [] = doSomething
    display(x) % will cause an error; what is x?
    localvar = 2;
end
```

Similarly, a program calling doSomething can't "see" localvar;

`localvar` only exists within the scope of the function

Recursion

Functions may call themselves; sometimes this is useful

No special declarations are required (cf. Fortran)

An example: a function to recursively calculate the Catalan number C_n using the recurrence relation $C_n = \frac{4n-2}{n-1} C_{n-1}$.

```
function [cn] = catalan(n)
    if n==1
        cn = 1;   % base case
    else
        cn = (4*n-2)*catalan(n-1)/(n+1);
    end
end
```