Matlab Basics
Lecture 2

Juha Kuortti

October 28, 2017

Logical operators
Flow Control

Relational Operators

Relational operators are used to compare variables. There are 6 comparison available:

- "equal to", using $==$
- "not equal to", using $\sim=$
- "less than", using $<$
- "less than or equal to", using $<=$
- "greater than", using $>$
- "greater than or equal to", using $>=$

The result of a comparison is either TRUE (1) or FALSE (0). Note that MATLAB does difference between logical value and numerical one, but allows the usual scalar operations to take place.

Array comparisons

Suppose A and B are double arrays of the same size. Let **op** be any of the 6 relational operators $(==, \sim=, <, <=, >, >=)$.

Then the expression

$$\mathbf{A} \quad \mathbf{op} \quad \mathbf{B}$$

is a logical array of the same size. The relational operator is applied <u>elementwise</u>, comparing $A(i, j)$ to $B(i, j)$.

Example:

```
>> A = rand(2,4);
>> B = 0.5*ones(2,4);
>> A<B
```

Excercise

Using `meshgrid` create a 2d space of matrices X and Y that covers the area $[-4, 4] \times [-4, 4]$.

Then find all the elements in X and Y for which it holds that $X(i,j)^2 + Y(i,j)^2 < 2$. Call the resulting logical array Z.

Finally `mesh(X,Y,Z)`.

Array comparisons

The result of a relational operation is a *logical array*.

- A logical array contains only 0's and 1's
- It cannot contain any other numerical values
- Internal representation in MATLAB is different than for double arrays.

You can use a logical array in any numerical calculation as though it is a double array; the 0's and 1's behave normally. MATLAB automatically typecasts the logicals to doubles.

```
>> A = [1 0 1 1];
>> B = logical(A);
>> whos
>> A==B
>> isequal(A,B)
```

Logical indexing

In a typical row/column reference,

$$\mathbf{M}(\mathbf{RowIndex}, \mathbf{ColIndex})$$

both **RowIndex** and **ColIndex** are double arrays, whose positive, integer values specify which rows and columns of the array M are being referenced.

If **RowIndex** and **ColIndex** are logical arrays, the locations of the 1's specify which rows and columns of the array **M** are being referenced.

```
>> M = rand(4,5);
>> Ridx = logical([1 0 0 1]);
>> Cidx = logical([0 0 1 1 1]);
>> M(Ridx,Cidx)  %same as M([2 4],[3 4 5])
```

Using `find`

The command find returns the indices of the nonzero entries of logical statement.

```
>> m = rand(6,1);
>> m(find(m<0.5)) = 0;
```

Usually logical indexing will work just fine, so you can just do

```
>> m = rand(6,1);
>> m(m<0.5) = 0;
```

More on `find`

However, when you need the actual index, use `find`.

```
>> P = peaks;
>> maxInd = find(P==max(P(:)))
>> mesh(P); hold on;
>> [m,n] = ind2sub(size(P),maxInd);
>> plot3(n,m,P(maxInd),'r*')
```

Sidebar: `find` will always provide the *linear indices* (remember those?). In order to transform them to subscript indices we use a function called `ind2sub`.

All things are not equal

In finite precision arithmetic (MATLAB has about 17 digits of precision), it is not true that

$$(a + b) + c \text{ is equal to } a + (b + c)$$

In practice this means that when comparing doubles, equality is not a good test for similarity; instead we usually use `abs(x-y)<tol` to check for “equality”. There are also other metrics — well learn them as we go.

Logical Operators

Logical operators are used to operate on logical variables. There are 3 binary operations

- "logical AND", using &
- "logical OR", using |
- "logical exclusive OR", using `xor`

There is also the unary operation

- "logical NOT", using $\sim$

For arrays, the operators are applied elementwise, and the results have logical values of TRUE (1) or FALSE (0)

In case of scalar values, there are also operators && and ||, that perform more efficiently.

Logical Operators

If A and B are scalars (double or logical), then

- $A\&B$ is TRUE (1) if A and B are both nonzero, otherwise it is FALSE (0)
- $A|B$ is TRUE (1) if either A or B are nonzero, otherwise it is FALSE (0)
- **xor**(A, B) is TRUE (1) if one argument is 0 and the other is nonzero, otherwise it is FALSE (0)
- $\sim A$ is TRUE if A is 0, and FALSE if A is nonzero.

For arrays, the operations are applied elementwise, so A and B must be the same size, or one must be a scalar.

If you wish to check if two arrays are same, use `all`, if you wish see whether they have any similarities, use `any`.

One notable exception

Usually logical operators only function on arrays of equal dimensions, for example testing `[1,2,3]==[3,2]`, will result in an error, rather than FALSE.

There is one situation where this will lead to problems: namely, comparing strings. In string context, comparison `'Hello' == 'Oh, Hel'` should clearly be FALSE, but will result in an error. For string comparisons, use MATLAB function `strcmp`.

Control: `if`, `end`

To conditionally control the execution of statements, you can use

```
if expression
    statements
end
```

If the real part of all of the entries of `expression` are nonzero, then the statements between the if and end will be executed. Otherwise they will not be. If the `expression` is an array, then the check is implicitly `all(expression)`.

Execution continues with any statements after the end.

Control: `if`, `else`, `end`

```
if exp1
   statements1
else
   statements2
end
```

One of the sets of statements will be executed

- If `exp1` is TRUE, then `statements1` are executed
- If `exp1` is FALSE, then `statements2` are executed

Control: `if`, `elseif`, `end`

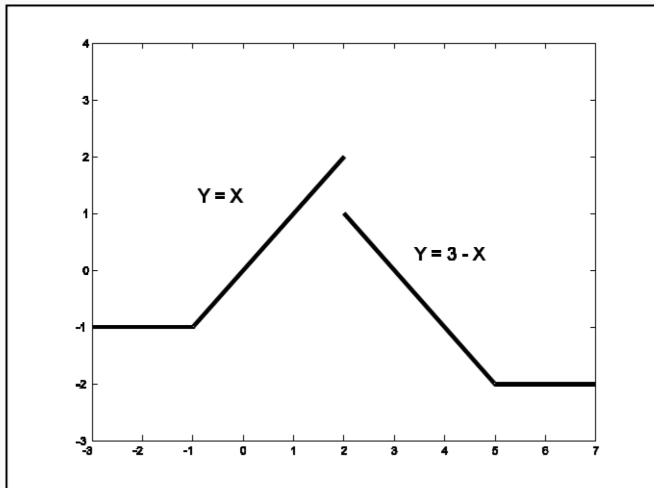
If you need to check for multiple cases, use `elseif`:

```
if exp_1
   statements1
elseif exp_2
   statements2
elseif exp_3
   statements3
end
```

Similar to `switch:case:break` structure in many other languages: execute statement involved with first true expression, then jump to `end`.

Excercise

Create a m-file function for the mathematical function $y = f(x)$ shown below.



Iterations
More control.

Control: `for`, `end`

Execute collection of statements a fixed number of times.

```
for x=expression
   statements
end
```

The expression is evaluated once before the loop starts. The value is called the controlvalue. The statements are executed one time for each column in the controlvalue. In the code above, x is the loopvariable.

Before each "execution pass," the loopvariable (x) is assigned to the corresponding column of controlvalue: 1st column on the first pass, 2nd on second and so on.

Control: `for, end`

The most common value for expression is a row vector of integers, starting at 1, and increasing to a limit n.

```
for x=1:n
   statements
end
```

The controlvalue is simply the row vector $[1, 2, 3 \ldots, n]$, hence the `statements` are executed n times.

Since

- The first time through, the value of x is set equal to 1
- the k'th time through, the value of x is set equal to k.

this is an extremely useful way to index an array.

Control: `for`, `end`

The expression can be created before the loop itself, so the loop doubles as `foreach` type loop.

```
for x=1:n
   statementxs
end
```

is same as

```
xValues = 1:n;
for x=xValues
   statements
end
```

Example

Write a function to compute the of amount owed for a loan amount L, given interest rate R, loan duration T and fixed payments of amount P.

Example

```
function P = loancalc(L,R,N,MP)
% P = loancalc(L,R,N,MP) computes the
% history of amount owed on a loan of amount
% L, interest rate R, duration N, and fixed
% monthly payment MP.
P = zeros(N+1,1);
P(1) = L; % amount owed at Month=0
% interest rate R is annual, but applied
% monthly, yielding a 1+R/12 factor.
G = 1+R/12;
for i=2:N+1
   P(i) = P(i-1)*G - MP;
end
```

Control: `while, end`

If you need to execute commands for an undetermined number of times, use `while` loop

```
while expression
   statements
end
```

`while` evaluates `expression`, and if it is TRUE, then executes the `statements`, and repeats, otherwise it jumps to `end`.

Notice, that expression need not become FALSE ever, leading to an infinite loop.

Example

Banach fixed point theorem states (paraphrasing heavily) that if a function f is a *contraction*, then iteration $x_{n+1} = f(x_n)$ will converge to a unique fixed point of the function regardless of the starting value x_0, i.e. a point x^* for which $f(x^*) = x^*$.

Let's use a `while` and fixed point iteration to find the fixed point of $f(x) = \cos(x)$.

Tiny bit of numerics
Few MATLAB built-ins

Basic building blocks

Famously, the Bolzano Theorem (later to be refined to intermediate value theorem) states that:

If a continuous function, f, with an interval, $[a, b]$, as its domain, takes values $f(a)$ and $f(b)$ at each end of the interval, then it also takes any value between f(a) and f(b) at some point within the interval.

More precisely, if we have a continous function that has values of opposite sign inside an interval, then it has a root in that interval.

Example: Bisection method

Let's apply the following, and write our own solver: this one will be very similar to binary search.

Method: given endpoints a and b

1. find the midpoint p in the middle of $[a, b]$.
2. if $f(p)$ is zero (or close enough) - stop, you've solved the problem.
3. if $f(p) > 0$ move the "positive end" to it, and go to step 1.
4. if $f(p) < 0$ move the "negative end" to it, and go to step 1.

Since the search interval is halved at each step, the convergence is fairly fast.

Example: Bisection method

```
function p = bisection(f,a,b)
if f(a)*f(b)>0
    disp('iteration is impossible')
else
    tol = 1e-7;
    p = (a + b)/2;
    while abs(f(p)) > tol
        if f(a)*f(p)<0
            b = p;
        else
            a = p;
        end
        p = (a + b)/2;
    end
end
```

Your turn: Newton's method

Suppose we have

$$f(x) = x - e^{-x^2}$$

and

$$f'(x) = 1 + 2xe^{-x^2}$$

Write a function that implements the Newton's method (teachers and Google will help if you've forgotten what it is), and use it to find the root of $f(x)$.

Note: it is technically possible to select a starting location so that the method does not converge — implement an iteration counter that stops the method after a large number of iteration has passed and solution has not been found to avoid an infinite loop.