Relational Operators

- We use relational operators to compare things (numbers, characters) in MatLab. There are 6 relational operators in MatLab:

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

- These operators return **true** or **false** (1 or 0) depending on whether some expression is true or false.

- These operators result in a matrix or vector of the same size as the operands, where if the relationship is true for an element the correspond-

ing resultant element is 1 and 0 otherwise. Remember: scalars (single numbers) are matrices of size 1×1 or vectors of length 1. Everything in MatLab is a matrix!!!

- If `x=[1 5 3 7]` and `y=[0 2 8 7]` then:

```
k= x <  y   % results in k=[0 0 1 0]
k= x <= y   % results in k=[0 0 1 1]
k= x >  y   % results in k=[1 1 0 0]
k= x >= y   % results in k=[1 1 0 1]
k= x == y   % results in k=[0 0 0 1]
k= x ~= y   % results if k=[1 1 1 0]
```

- Typically, these are used in conditional statements but can also be used to

do computing. For example, `u=v(v>=sin(pi/3))` finds all elements of `v` such that if the i^{th} element `v` is `>= sin(pi/3)` the i^{th} element is stored in vector `u`.

Logical Operations

- Two or more relational operators can be combined using logical operators. There are 4 logical operators:

`&&`	logical AND
`||`	logical OR
`~`	logical complement (NOT)
`xor`	exclusive OR

- `&` does the same as `&&` except `&&` short circuits the evaluation of the right expression if the left expression evaluates to false. Thus `A && B` does

not evaluate `B` if `A` is false.

- `|` does the same as `||` except `||` short circuits the evaluation of the right expression if the left expression evaluates to true. Thus `A || B` does not evaluate `B` if `A` is true.

- `x xor y` means or (`||`) except if both corresponding elements of `x` and `y` are the true, set `xor` for those elements to false.

- Thus `xor` yields:

```
false xor true  = true
true  xor false = true
false xor false = true
true  xor true  = false
```

while `or` (matlab symbol `|`) yields:

```
false | true  = true
true  | false = true
false | false = false
true  | true  = true
```

- For `x=[0 5 3 7]` and `y=[0 2 8 7]` then:

```
m = (x>y) & (x>4)    % results in m=[0 1 0 0]
m = x|y              % results in m=[0 1 1 1]
m = ~(x|y)           % results in m=[1 0 0 0]
m = xor(x,y)         % results in m=[0 0 0 0]
```

- `x((x>y) & (x>4))` gives the elements of `x` that are greater than the corresponding element of `y` and greater than 4 at the same time.

- There are also some built-in logical operators:

```
% all(condition on matrix) returns true (=1) if all
% elements of the matrix satisfy
% the condition, i.e. all(x>0) is true is all
% elements in x > 0
all(condition)


% any(condition on matrix) returns (true (=1) if any
% element of the matrix satisfies the
% condition, i.e. any(x) returns true is any
```

```
% element of x is non-zero
any(condition)

% true if x is an existing function or variable
exist(x)

% true if x is []
isempty(x)

% true for all non-defined elements of x
isnan(x)
```

```
% returns the coordinates of all elements of x
% that are non-zero
find(x~=0)
```

Character Strings

- Character strings are row vectors with one element per character. For example,

  ```
  message='Leave me alone'
  ```

  is a 1×14 vector.

- `names1=['John';'Ravi';'Mary';'Xiao']` creates a column vector with one name per row (thus `names1` is a 4×4 matrix). The

rows must all be the same size, so

```
names2=['Hi';'Hello';'Bonjour']
```

causes an error. On the other hand,

```
names2=['Hi     ';'Hello  ';'Bonjour']
```

works. Of course, using a "cell" array would work as well:

```
names2={'Hi';'Hello';'Bonjour'}
```

- In the case of arrays we can use `char` to pad these blank characters.

```
names2=char('Hi','Hello','Bonjour')
```

- To have a quote character inside a string you quote the quote. For example, `title('John''s')` to print `John's`. `title` puts a title on a graph (coming soon).

- Character strings can be manipulated just like matrices.

```
c=[names2(2,:) names1(3,:)]
```

  has `Hello Mary` as the output for `c`.

- Some built-in character functions:

  1. `char`, `abs` (convert characters to their ascii numeric values or vice versa),
  2. `eval` (execute the string as a command),

3. `num2str`, `str2num` (convert number to character or vice versa),
4. `ischar`, `isletter` (is operand a character or an integer?),
5. `lower`, `upper` (convert upper case characters to lower case characters or vice versa),
6. `strcmp` (compare character strings, could do comparison directly) and
7. `strcat` (concatenate 2 strings, could use `[` and `]` as well).

- `eval('x=5*sin(pi/3)')` computes $5\sin(\pi/3)$ and is equivalent to typing `x=5*sin(pi/3)` in the command window.
- String concatenation can be done using the left and right square brackets:

```
a=['abc' '123' 'cd' 'q45'];
```

gives the value `'abc123cdq45'` to variable `a`.

- One good use for string concatenation is to generate meaningful file names for I/O. Another good use is to generate meaningful titles for figures.

Some Examples

```
if 'john' < 'mary' & 'hello' > 'bonjour'
  disp('Yabba dabba doo');
else
  disp('Sad to say');
end
```

- prints `'Yabba dabba doo'`. `'john'` is less than `'mary'` (because the first characters, `j < m`) but `'hello'` is greater than `'bonjour'` (because `h > b`).

- To compare `'hello'` with `'bonjour'` pad `'hello'` with blank characters on the right until the result has the same number of characters as `'bonjour'`, i.e. `'hello  '`. Then compare the corresponding characters left to right until inequality is found. This comparison uses the ascii (American Standard Code for Information Interchange) values for the characters. The ascii character for '0' is 48, for 'A' is 65, for 'a' is 97, for blank is 32 (other visible special characters appear to be between 33 and 47 or between 123 and 126).

- Another comparison example: `'apple'` and `'apple pie`. First we

pad `'apple' with 4 blanks to get \verb`'apple '!. Now we compare the ascii code for characters from left to right. So, for the first five characters `a`! equals `a`!, `p`! equals `p`!, `p`! equals `p`!, `l`! equals `l`! and `e`! equals `e`!. For the 6^{th} characters `' '` and `'p'` has ascii codes 32 and 112 so `' ' < 'p'` and `'apple' < 'apple pie`.

- Find all elements of `x` greater than 3:

```
x =[1 2 3 4 5 6 7];
x(x>3)=1
x =[1 2 3 1 1 1 1];
```

- Find the coordinates of all elements greater than 100:

```
x=[12 34 56 78 910 1112];
```

```
find(x>100)

ans =

     5     6
```

- The use of string and concatenation operators:

```
a=strcat('abc','123')

a =

abc123


a=strcmp('abc','abdefg') % a=0 is for false

a =

     0

a=strcmp('abc','abc') % a=1 is for true
```

```
a =

     1
```

- Note that `true` and `false` are 1 and 0. So:

```
fprintf('true=%d\n',true);
fprintf('false=%d\n',false);
```

produces the output:

```
true=1
false=0
```

Example MatLab Functions for Mean/Standard Deviation Calculations using Arrays

- Consider computing the mean and standard deviation of an array of numbers stored in **x**. We write 2 MatLab functions, **my_mean.m** and **my_std.m** to compute these values and compare them to MatLab's **mean** and **std** functions.

- To compute the mean (or average) of a list of numbers in MatLab we can use **mean(x)** where **x** is a 1D array of valid numbers. One restriction we impose on the code we write here is that we must use arrays to compute these values (and not write faster, perhaps more efficient vectorized

code). The required calculation is:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{1}$$

where n is the number of elements in **x** (we can use **numel(x)** to compute the **num**ber of **el**ments of **x**). The MatLab code to compute this (in file **L05_my_mean.m**) is:

```
function [ave]=my_mean(x)
n=numel(x);
total=0.0;
for i=1:n
  total=total+x(i);
end
ave=total/n
end % my_mean
```

Note that the vectorized solution is simply **sum(x)/numel(x)**.

- To compute the standard deviation, **my_std**, we have to compute:

$$\sigma = \frac{1}{n}\sqrt{\sum_{i=1}^{n}\frac{(x_i - \bar{x})^2}{n-1}}, \tag{2}$$

where all variables are as before. The -1 in $n-1$ is often used to represent a "loss of 1 degree of freedom". **std(x)** or **std(x,0)** uses $n-1$ in the calculation while **std(n,1)** uses n instead.

- The MatLab code to compute the average of an array **x** (in file **L05_my_std.m**) is given as:

```
function [sigma]=my_std(x)
% compute x_bar (the average of x)
x_bar=my_mean(x);
n=numel(x);
if(n<=1)
   fprintf('Fatal error: must have at least 2 elements\n');
```

```
   fprintf(' in x for a standard deviation calculation\n');
   return;
end
term=0;
for i=1:n
   term=term+(x(i)-x_bar)^2;
end
sigma=sqrt(term/(n-1));
end % my_std
```

- Finally, we need some code to call and compare your **my_mean.m** and **my_std.m** functions with the **mean** and **std** functions built into MatLab. We write a last MatLab function, **compare_mean_std**, with no input or output parameters. The MatLab code (in file **L05_compare_mean_std**) is:

```
function compare_mean_std()
a=rand(769,1,'double');
b=randn(23595,1,'double');
```

```
% Generate the list of normally distributed random
% numbers with sigma=5.0 and mean 3
c= 5.0*randn(5000,1)+3;

% For uniformly distributed random numbers in [0,1] we
% expect the average to be around 0.5.
fprintf('Using Matlab builtin functions for a: average=%10.5f +- %10.5f\n',..
        mean(a),std(a));
fprintf('Using user written functions for a: average=%10.5f +- %10.5f\n',...
        my_mean(a),my_std(a));
% For normally (Gaussian) distributed random numbers we
% expect the average to be 0 and the standard deviation to be 1
% as these are the default settings of randn.
fprintf('Using Matlab builtin functions for b: average=%10.5f +- %10.5f\n',..
        mean(b),std(b));
fprintf('Using user written functions for b: average=%10.5f +- %10.5f\n',...
        my_mean(b),my_std(b));
%For second normal distribution of numbers stored in c
% we expect an average around 3 and a standard deviation value around 5
fprintf('Using Matlab builtin functions for b: average=%10.5f +- %10.5f\n',..
        mean(c),std(c));
fprintf('Using user written functions for b: average=%10.5f +- %10.5f\n',...
        my_mean(c),my_std(c));
end % compare_mean_std
```

- This program has the following output:

```
Using Matlab builtin functions for a: average=   0.50981 +-    0.29658
Using user written functions for a: average=   0.50981 +-    0.29658
Using Matlab builtin functions for b: average=  -0.00509 +-    1.00083
Using user written functions for b: average=  -0.00509 +-    1.00083
Using Matlab builtin functions for b: average=   3.06347 +-    5.07695
Using user written functions for b: average=   3.06347 +-    5.07695
```

We can see that the normal distribution means and standard deviations values worked out as predicted. The uniform average is also very close to its predicted value.

- Files **L05_my_mean.m**, **L05_my_std.m** and **L05_compare_mean_std.m** have been zipped into **L05files.m** put on the course webpage. Note that since all these functions are ended by an **end** statement **L05_my_mean.m** and **L05_my_std.m** could be put in **L05_compare_mean_std.m** at the end (i.e. as one file named **L05_compare_mean_std.m** with 3 functions).