Computational Mathematics with MATLAB

Topic 6

Conditional Statements

Outline

Array Comparison

- Logical data & relational operators revisited (from Topic 2)
- Logical operators (AND, OR, NOT, XOR)
- Short-circuit AND and OR operators
- Using conditional expressions to select array elements meeting certain criteria (Logical Indexing)

Branching Structures

- if blocks
- if-else blocks
- if-elseif-else blocks

More on the **disp** function

Introduction to WHILE loops

Logical Data Type (Topic 2 slides)

The logical data type represents **TRUE or FALSE states** using the numbers 1 and 0, respectively. Logical values can be created using **relational operators** and certain MATLAB functions.

Logical values (scalars):

1 – if the proposition/statement is **TRUE**

0 – if the proposition/statement is **FALSE**

Relational Operators					
<	**less than**	>	**greater than**	==	**equal to**
<=	**less than or equal to**	>=	**greater than or equal to**	~=	**not equal to**

Examples of Logical Variables (Topic 2 slides)

MATLAB Input	Output (ans)
`>>   2 < 6`	1
`>>   2 <= 6`	1
`>>   2 > 6`	0
`>>   2 >= 6`	0
`>>   2 == 6`	0
`>>   2 ~= 6`	1
`>>   isprime(4)`	0
`>>   isreal(3)`	1
`>>   any([0 0 2 0 0])`	1

Examples of Logical Arrays (Topic 2 slides)

Command Window

```
>>
>> x=[ 5 2 56 2]; y=[1 3 12 4]; x>y

ans =

     1     0     1     0
```

Output: 1 x 4 logical vector
the relationship x>y holds in the 1^{st} and 3^{rd} positions

```
>> M=randi(20,3)

M =

     2     7     9
    17    20     8
    14     1    16
```

M is a 3 x 3 matrix filled with integers chosen randomly from [1,20]

M(1,1), M(1,2) and M((2,1) are primes

```
>> isprime(M)

ans =

     1     1     0
     1     0     0
     0     0     0
```

Output: 3 x 3 logical matrix
the locations of prime numbers in M are indicated by the 1s

```
fx >>
```

More Examples

Let

```
>> a = [1 3 5 7 9];
   b = [4 6 5 6 1];
   c = [1 2 3 4 5 6];
```

then

```
>> disp(a < b)
            1        1        0        0        0
>> disp(a == b)
            0        0        1        0        0
>> disp(a ~= b)
            1        1        0        1        1
>> disp(a == c)
??? Error using ==> eq
Matrix dimensions must agree.
```

Basic Logical Operators

<table>
<tr><td>&</td><td>AND</td></tr>
<tr><td>|</td><td>OR</td></tr>
<tr><td>~</td><td>NOT</td></tr>
<tr><td>xor()</td><td>Exclusive OR
(no XOR operator symbol in MATLAB!)</td></tr>
</table>

<table>
<tr><th colspan="6">TRUTH TABLE</th></tr>
<tr><th colspan="2">Statements</th><th colspan="4">Operations</th></tr>
<tr><th>A</th><th>B</th><th>~A</th><th>A | B</th><th>A & B</th><th>xor(A,B)</th></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr>
</table>

Examples with Logical Scalars

```
>> A = 2 < 3;
   B = 2 > 3;

>> disp([A, B])
     1     0

>> disp(~A)
     0

>> disp(A | B)
     1

>> disp(A & B)
     0

>> disp(xor(A,B))
     1
```

Examples with Logical Arrays

```
Recall arrays a and b from earlier:
a = [1 3 5 7 9];
b = [4 6 5 6 1];

>> x=(a < b); disp(x)
     1     1     0     0     0

>> y=(a ~= b); disp(y)
     1     1     0     1     1

>> disp(~x)
     0     0     1     1     1

>> disp(x | y)
     1     1     0     1     1

>> disp(x & y)
     1     1     0     0     0

>> disp(xor(x, y))
     0     0     0     1     1
```

Short-Circuit AND (&&) and OR (||) for Scalar Comparison

```
>> A = 2 < 3; B = 2 > 3; C=mod(4,2)==0; D=isprime(10);

>> disp([A,B,C,D])
     1     0     1     0

>> disp(A & B & C & D)
     0    % because not ALL statements are TRUE

>> disp(A | B | C | D)
     1    % because AT LEAST one statement is TRUE
```

Consider (A & B & C & D)

MATLAB evaluates all 4 statements, and then uses the truth table to determine the final answer, which is FALSE.

The final answer is determined as soon as B is evaluated, so statements C and D could be ignored.

Using && and ||

Now let's look at (A | B | C | D)

Here the final answer is determined as soon as A (TRUE) is evaluated. But MATLAB still evaluates the rest of the statements.

The short-circuit AND (&&) and OR (||) operators give the same answers as the normal (elementwise) & and | operators, but work more efficiently.

&& stops evaluation at the first FALSE statement

|| stops evaluation at the first TRUE statement

MATLAB's editor prefers the short-circuit operators.

For more information visit:

http://www.mathworks.co.uk/help/matlab/ref/logicaloperatorsshortcircuit.html

Do not make this mistake!

Short-circuit operators cannot be used for array comparison.

Why is that? Let's look at an example!

```
% Define 4 random 1 x 5 logical arrays
>> p=isprime(randi(20,1,5)); q=isprime(randi(20,1,5));
   r=isprime(randi(20,1,5)); s=isprime(randi(20,1,5));

>> disp([p; q; r; s])
     0     1     1     1     0
     1     0     1     1     1
     1     0     0     1     0
     0     1     0     1     1

>> disp(p & q & r & s)
     0     0     0     1     0

>> disp(p && q && r && s)
Operands to the || and && operators must be convertible
to logical scalar values.

% The final answer cannot be determined without evaluating all
  statements - there is no way to stop early!
```

Selection with Conditional Expressions

```
>> M=randi(25,3,4); disp(M)
     8    21     9    23
    16    25    15    22
     7    19     3    21
```

We can use MATLAB to answer the following questions (and more):

1) which elements of M are prime numbers?
2) which elements are larger than 10?
3) which elements are larger than 10 but smaller than 20?
4) which elements are square integers?
5) which elements are can be divided by 6 (without a remainder)?
6) which elements are integer powers of 2?

Selection with Conditional Expressions

Conditional expressions to identify the elements in question:

(1) which elements of M are prime numbers?

```
isprime(M)
```

(2) which elements are larger than 10?

```
M>10
```

(3) which elements are larger than 10 but smaller than 20?

```
M>10 & M<20
```

(4) which elements are square integers?

```
sqrt(M)==round(sqrt(M))
```

(5) which elements are can be divided by 6 (without a remainder)?

```
mod(M,6)==0
```

(6) which elements are integer powers of 2?

```
mod(log2(M),1)==0   or   log2(M)==ceil(log2(M))
```

All of the above expressions produce 3 x 4 logical arrays.

Logical Indexing

Indexing with conditional expressions (i.e. logical arrays)

M(condition) produces a list of the elements that satisfy the condition (in linear order)

```
disp(M>10 & M<20) % this is a logical array
     0     0     0     0
     1     0     1     0
     0     1     0     0

>> M(M>10 & M<20) % the logical array is now used to index M

ans =
    16
    19
    15

>> disp(M(isprime(M))')
     7    19     3    23

>> disp(M(mod(M,6)==0))
>>
% there are no such elements!
```

Conditional Statements: if blocks

The **if** statement executes a statement (or list of statements) when the expression after the **if** statement is TRUE.

```
if   expression (TRUE or FALSE)
        block of statements
end
```

No action is taken if the expression is FALSE.

Example:

```
>> x=input('enter x > '); if  x>7 x=x-1;  end; disp(x)
enter x > 5
     5

>> x=input('enter x > '); if  x>7 x=x-1;  end; disp(x)
enter x > 45
    44
```

Conditional Statements: if-else blocks

```
if   expression (TRUE or FALSE)
         statement block 1
else
         statement block 2
end
```

Statement block 1 is executed if the expression is TRUE.
Statement block 2 is executed if the expression is FALSE.

```
% robot bartender
x=input('Enter your age > ');
if x>=18
    disp('Enjoy your drink!')
else
    disp('You are too young to drink. Go home!')
end
```

Conditional Statements: if-elseif-else blocks

```
if   expression 1
         statement block 1
elseif expression 2
         statement block 2
elseif expression 3
         statement block 3
(…)
else
optional final statement block
end
```

- Statement block 1 is executed if *exp 1* is TRUE.
- Statement block 2 is executed if the *exp 1* is FALSE and *exp 2* is TRUE.
- Statement 3 is executed only if both *exp 1* and *exp 2* are FALSE and *exp 3* is TRUE.
- Any number of **elseif** statements can be included.
- The **else** is optional (no **else** means *'in any other case do nothing'*)

Example Script

```
% quantity based pricing
% the total price depends on the number of units purchased

x=input('Enter quantity> ');
if      x<10
    price=15*x;
elseif  x<20
    price=13*x;
elseif  x<50
    price=12*x;
else
    price=11*x;
end
disp(['The total price of ' num2str(x) ' items is £ ' num2str(price)])
```

Notice the way the disp function is used in the example!

More on the disp function

Recall that the disp( ) function only takes one argument!

The argument can be any array (numerical, string, logical).

```
>> x1=12;  x2=23;  x3=52;  x4=67;  x5=81;

>> disp (x1, x2, x3, x4, x5)  ← this does not work
??? Error using ==> disp
Too many input arguments.

>> disp ([x1, x2, x3, x4, x5])          ← this does
12  23     52     67     81
```

numerical example

```
>> s1='12';  s2='23';  s3='52';  s4='67';  s5='81';

>> disp(s1, s2, s3, s4, s5) ← this does not work
Error using disp
Too many input arguments.

>> disp([s1, s2, s3, s4, s5])          ← this does
1223526781
```

string example

More on the disp function

The simplest way **to combine text and numerical variables** is to convert numerical variables to text using the num2str( ) function.

Example: Let x1, x2, x3, x4 and x5 represent the winning numbers in a lottery. To display the following sentence:

The winning numbers are 12 23 52 67 and 81

we can use the code:

```
>> disp(['The winning numbers are ' num2str([x1, x2, x3, x4]) ' and ' num2str(x5)])
The winning numbers are 12  23  52  67 and 81
```

Without the extra spaces the output is much harder to read:

```
>> disp(['The winning numbers are' num2str([x1, x2, x3, x4]) 'and' num2str(x5)])
```

```
The winning numbers are12  23  52  67and81
```

More on the disp function

```
>> disp(['The winning numbers are ' num2str([x1, x2, x3, x4]) ' and ' num2str(x5)])
```

Here the argument of disp( ) is a character array which was created by merging 4 strings into a single string of length 45:

1) 'The winning numbers are ' (length 24)
2) num2str([x1, x2, x3, x4]) (length 14)
3) ' and ' (length 5)
4) num2str(x5) (length 2)

The format of the output can be controlled much better with the fprintf function (but it is less straightforward to use).

```
>> nx1=str2double(num2str(x1)); disp(nx1)
    12

>> disp(class(nx1))
double
```

use str2double to convert from string to numerical

Loops and Conditional Statements: Example 1

Write a script to replace each element of a given array (A) with

+1 if the element is even

-1 if the element is odd

0 otherwise

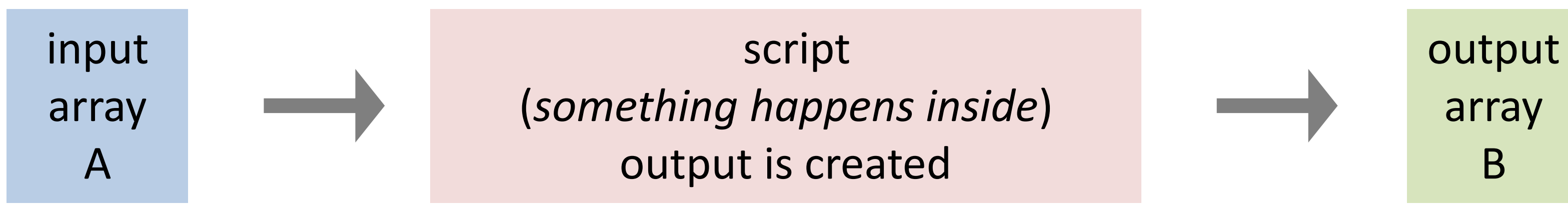
Example structure plan

1) Take the input (save as *A*)
2) Determine the row & column number of the input (save as *n* and *m*).
3) Initialise output variable *B* (must have *n* rows and *m* columns!)
4) Create a nested loop which goes through all rows (1:*n*) and all columns (1:*m*)
5) The loop must

- check whether *A*(i, j) is even, odd or neither *for each possible (i, j) pair*
- change each *B*(i, j) to either 1, -1 or 0 accordingly

6) Display the output (variable *B*)

Handling Array Input



- The script in Example 1 must work on an array of *any size* (scalar, vector, matrix)
- The *script does not create the input array* – the input variable should either come from the Workspace or the user can create the input manually (when prompted). Try *not to overwrite the input* inside the script – save it under another name.
- Use the **`size`** function to determine the row and column numbers of the input. The output of **`size`** can either be referenced as a 1 x 2 vector or as 2 scalar variables.

```
% reference the output of size as vector
n = size(A,1)
m = size(A,2)
% reference the output of size as 2 scalar variables
[n, m] = size(A)
```

Example 1 – Sample Script

```
clc
A = input('Enter an array > '); %  use array from Workspace
%  determine input size
[n,m]=size(A);
%  initialise output variable
B=A;
for i=1:n % in each row
    for j=1:m % and in each column
        if        mod(A(i,j),2)==0, % check condition 1
                        B(i,j)=1;    % action 1
        elseif    mod(A(i,j),2)==1, % check condition 2
                        B(i,j)=-1;   % action 2
        else      % NEVER put a condition after else
                        B(i,j)=0;    % action 3
        end
    end
end
disp(B)

%  no need for else if B is defined as zeros(n, m)
```

The WHILE loop

A **WHILE** loop repeatedly executes a block of statements *as long as* the loop condition remains TRUE. The conditional expression is evaluated in each cycle, before any statement is executed.

```
while  expression (TRUE or FALSE)
        block of statements
end
```

```
% basic example
x=input('enter a number > ');
while  x<20
    x=x+1;
end
disp(x)
```

A badly written WHILE loop may never stop. Use **CTRL-C** to interrupt an infinite loop. Can you modify the above example to create an infinite loop?

Example 2

Given a list of *n* numbers, keep removing the smallest element until the average of the new list exceeds the original average by at least 50%. Display the original and final averages.

Example structure plan

1) Take the input (v) – ask the user to enter a vector!
2) Calculate the average of the input (avg)
3) Sort the input list so that the smallest element comes first
4) Create a while loop where the loop condition is mean(v)<1.5*avg

   (in the 1st iteration, mean(v)=avg so the condition is TRUE and the loop will start running)
5) Remove the 1st (i.e. smallest) element of v and close the loop

(the condition mean(v)<1.5*avg will be evaluated again with the new v, and the loop will keep on running until the average becomes large enough or until all elements are removed)

6) Display avg and the final mean(v)

Example 2 – Sample Script

```
clc
% take the input - do not create a list inside the script!
v=input('input list(vector): > ');
% sort the input
v=sort(v);
% original average
avg=mean(v);

% keep removing the smallest element
while mean(v)<1.5*avg
   v(1)=[];
end

% display output variables with some text
disp(['original average: ' num2str(avg)])
disp(['final average: ' num2str(mean(v))])
```

Can you modify this script so that it calculates the number of elements removed and the % increase in the average? Can you also check if the input is correct (vector)?

Example 3 (more complicated)

Consider the unit square as shown below. If we select a large number of points randomly inside the square, the proportion of points inside the unit circle will tend to $\pi/4$ as $n \to \infty$. Use this to estimate π to 3 decimal places.

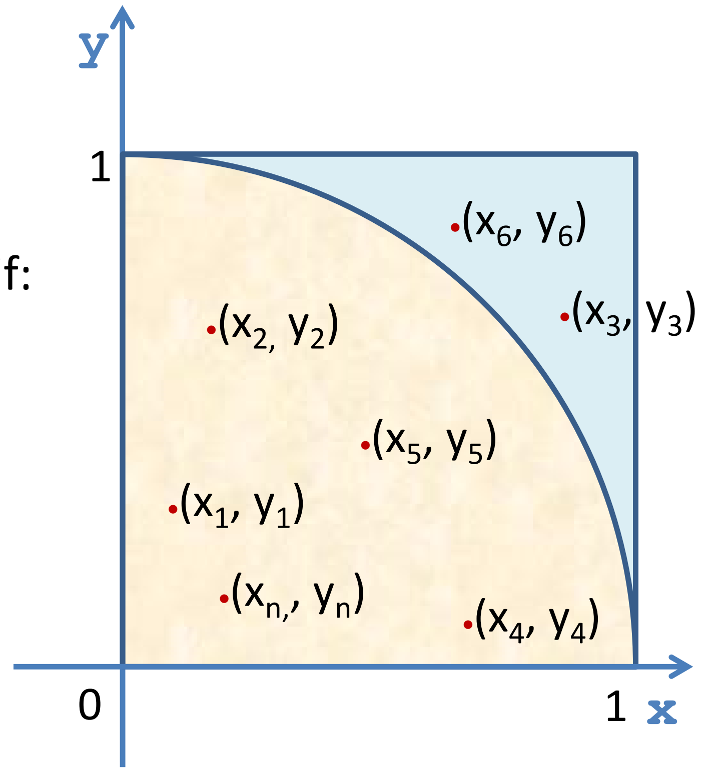
Points inside the unit square are represented by pairs of uniformly distributed random numbers: **`x,y ~ U(0,1).`**

Point **`(x,y)`** is inside the (sector of) the unit circle if:

$$\mathbf{x^2 + y^2 < 1}$$

Given a large number (**n**) of randomly generated points, we expect

$$\frac{\text{No. of points inside sector}}{\text{Total number inside square}} \approx \frac{\text{area of sector}}{\text{area of square}} = \frac{\pi}{4}$$



Example 3 – Sample Script

```
clc
n = 100;  % number of points
% abs_error will be defined as the difference between the estimated
and true values of pi - it needs an initial value before the first
estimate is calculated

abs_error = 1;  % choose a value so that the initial condition is TRUE

while abs_error > 0.0005,

    n = n*2;           % double the number of points
    x = rand(n,1);  % x-coordinates of random points
    y = rand(n,1);  % y-coordinates of random points

    % number of points inside the circle
    m = sum(ceil( 1-sqrt(x.^2 + y.^2) ));
    % sqrt(x.^2 + y.^2))  is a vector including all distances from the origin
    % ceil(1-sqrt(x.^2 + y.^2))  is a vector filled with 0s and 1s

    pi_est = 4*m/n;
    abs_error = abs(pi-pi_est);

end

disp(['The estimated value is ' num2str(pi_est) ' using '...
num2str(n) ' random numbers.']);
```