Flow Control

Spring 2019

Relational Expressions

Conditions in `if` statements use expressions that are conceptually either `true` or `false`. These expressions are called **relational expressions** or **Boolean expressions**

Operator	**Meaning**
>	greater than
<	less than
>=	greater than or equals
<=	less than or equals
==	equality
∼=	inequality

Logical Operators

- The result of a logical operation is `1` if it is `true` and `0` if it is `false`.
- Logical operators can be used to create compound statements that evaluate to `true` or `false`

`&`	logical AND
`|`	logical OR
∼	complements elements of A
`xor`	exclusive OR
`all`	TRUE if all elements of an array are TRUE
`any`	TRUE if any elements of an array are TRUE

Logical Operators

- The result of a logical operation is `1` if it is `true` and `0` if it is `false`.
- Logical operators can be used to create compound statements that evaluate to `true` or `false`

`&`	logical AND
|	logical OR
∼	complements elements of A
`xor`	exclusive OR
`all`	TRUE if all elements of an array are TRUE
`any`	TRUE if any elements of an array are TRUE

The following short circuit operators **only work with scalars**

- `&&` : `(exprA && exprB)` - `exprB` is only evaluated if `exprA` is `true`.
- `||` : `(exprA || exprB)` - `exprB` is not evaluated if `exprA` is `true`

Truth tables

A	B	A & B	A | B	∼A	xor(A,B)
0	0	0	0	1	0
0	1	0	1	1	1
1	0	0	1	0	1
1	1	1	1	0	0

Precidence rules

Operator	Operation	Priority
∼	`NOT`	Highest
`&`	`AND`	
`|`	`OR`	
`&&`	`short circuit AND`	
`||`	`short circuit OR`	Lowest

- It is a good idea to use paranthesis on long expressions

Precidence rules

<table>
<tr><th>Operator</th><th>Operation</th><th>Priority</th></tr>
<tr><td>~</td><td>NOT</td><td>Highest</td></tr>
<tr><td>&</td><td>AND</td><td></td></tr>
<tr><td>|</td><td>OR</td><td></td></tr>
<tr><td>&&</td><td>short circuit AND</td><td></td></tr>
<tr><td>||</td><td>short circuit OR</td><td>Lowest</td></tr>
</table>

- It is a good idea to use paranthesis on long expressions
- `a|b&c` is evaluated as `a|(b&c)`

Precidence rules (always use parenthesis)

```
>> b=10;
>> 1|b>0 &0
ans =
logical
1
```

Precidence rules (always use parenthesis)

```
>> b=10;
>> 1|b>0 &0
ans =
logical
1
```

```
>> (1|b>0) &0
ans =
logical
  0
```

Precidence rules (always use parenthesis)

```
>> b=10;
>> 1|b>0 &0
ans =
logical
1
```

```
>> (1|b>0) &0
ans =
logical
  0
```

```
>> 1|(b>0 &0)
ans =
logical
1
```

Logical data types

- logical data types can be used to as indices for to extract specific elements of vectors

```
>> x=1:10
x =
  1    2    3   4   5   6   7   8   9   10
>> y=[3  5  6  1  8  2  9  4  0  7]
y =
   3   5   6   1   8   2   9   4   0   7
>> y(x<4)
ans =
   3     5     6
```

Logical data types

- However, not 0−1 vectors are **logical data types**

```
>> rem(y,2)
ans =
    1   1   0   1   0   0   1   0   0   1
```

Logical data types

- However, not 0−1 vectors are **logical data types**

```
>> rem(y,2)
ans =
     1    1    0    1    0    0    1    0    0    1
```

- If we try to extract the odd entries as:

```
>> y(rem(y,2))
Array indices must be positive integers or logical ...
    values.
```

- This is due to the fact that `rem` does not return a logical data type

Logical data types

- We can use the `logical` function to fix this problem

```
>> y=[3  5  6  1  8  2  9  4  0  7]
y =
    3    5    6    1    8    2    9    4    0    7
>> y(logical(rem(y,2)))
ans =
       3       5       1       9       7
```

find command

- We can extract elements from a vector satisfying a certain condition.

```
>> x=[1 1 1 4 5 2 1]
x =
     1     1     1     4     5     2     1
>> find(x==1)
ans =
     1     2     3     7
>> x(find(x==1))
ans =
   1     1     1     1
```

find command

- We can extract elements from a vector satisfying a certain condition.

```
>> x=[1 1 1 4 5 2 1]
x =
     1     1     1     4     5     2     1
>> find(x==1)
ans =
     1     2     3     7
>> x(find(x==1))
ans =
  1     1     1     1
```

- `find` also works for matrices, check the documentation for usage

`find` command

- We can extract elements from a vector satisfying a certain condition.

```
>> x=[1 1 1 4 5 2 1]
x =
     1     1     1     4     5     2     1
>> find(x==1)
ans =
     1     2     3     7
>> x(find(x==1))
ans =
  1     1     1     1
```

- `find` also works for matrices, check the documentation for usage
- Other useful commands: `any` and `all`

Selection control – `if` statements

`if-blocks` are used to decide which instruction to execute next depending on wheather an *expression* is `true` or not.

- `if ...end`

```
if logical_expression
    statement1
    statement2
end
```

Selection control – `if` statements

`if-blocks` are used to decide which instruction to execute next depending on wheather an *expression* is `true` or not.

- `if ...end`

```
if logical_expression
    statement1
    statement2
end
```

- `if ...else ...end`

```
if logical_expression
    statements evaluated if TRUE
else
    statements evaluated if FALSE
end
```

Selection control – if statements

if-blocks are used to decide which instruction to execute next depending on wheather an *expression* is true or not.

- if ...elseif ...else ...end

```
if logical_expression1
    block of statements evaluated
    if logical_expression1 is TRUE
elseif logical_expression2
    block of statements evaluated
    if logical_expression2 is TRUE
else
    block of statements evaluated
    if no other expression is TRUE
end
```

Selection control – `if` statements - Examples

```
%selection statements
%if ...end
a=input('Enter an integer:');
if(mod(a,2)==0)
    fprintf('Your integer %d is even\n',a);
end

%if...else...end
%we can add more feedback
if(mod(a,2)==0)
    fprintf('Your integer %d is even\n',a);
else
    fprintf('Your integer %d is odd \n',a);
end
```

Selection control – `if` statements - Examples

```matlab
%a code segment that categories height
height = input('Enter your feet:');
if (height > 7)
    disp ('very tall');
elseif (height > 6)
    disp ('tall');
elseif (height < 5)
    disp ('short');
else
    disp ('average');
end
```

Selection control – `Switch/Case` statements

- *Switches* between several cases depending on an expression, which is either a scalar or a string.

```
a=input('Enter an integer:');
switch(mod(a,2))
    case 0
        fprintf('Your integer %d is even\n',a);
    case 1
        fprintf('Your integer %d is odd \n',a);
    otherwise
        fprintf('The number %f is not an integer\n',a);
end
```

Selection control – `Switch/Case` statements

- *Switches* between several cases depending on an expression, which is either a scalar or a string.

```
a=input('Enter an integer:');
switch(mod(a,2))
    case 0
        fprintf('Your integer %d is even\n',a);
    case 1
        fprintf('Your integer %d is odd \n',a);
    otherwise
        fprintf('The number %f is not an integer\n',a);
end
```

- Handy for avoiding tedious `if..elseif..` statements

Iteration control – `for` Loop

- the `for` loop repeats a block of statements a **fixed** number of times.
- Usage:

```
for index = first:step:last
    block of statements
end
```

Iteration control – `for` Loop

Example

Computing the sum of a geometric series with N terms, first term a and common ration r.

$$S_N = a + ar + ar^2 + ar^3 + \ldots + ar^{N-1}$$

Iteration control – `for` Loop

Example

Computing the sum of a geometric series with N terms, first term a and common ration r.

$$S_N = a + ar + ar^2 + ar^3 + \ldots + ar^{N-1}$$

Exercise

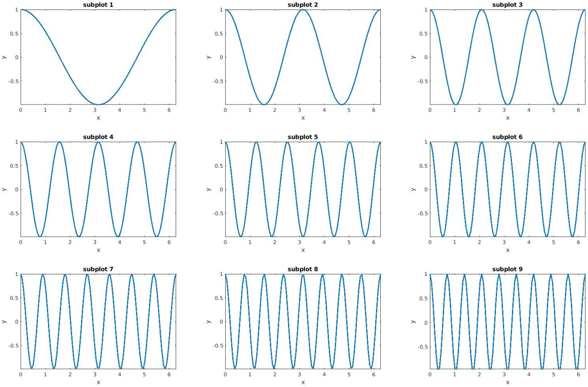
Write a `for` loop to compute the sum

$$1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots + \frac{x^N}{N!}$$

for any integer N

Iteration control – `for` Loop (Example)

Use a `for` loop to plot $cos(nx)$ using subplots for $n = 1 - 9$ on $[0, 2\pi]$



Iteration control – `for` Loops (Example)

Use a `for` loop to plot $cos(nx)$ using subplots for $n = 1 - 9$ on $[0, 2\pi]$

```
%using for loops with subplot
clc
clf
x = linspace(0,2*pi); %default 100 pts
for n=1:9
subplot(3,3,n);
y = cos(n*x);
plot(x,y,'LineWidth',2);
xlabel('x');
ylabel('y');
str =['subplot ',num2str(n)];
title(str);
axis tight
end
```

Iteration control – while Loop

- while loop evaluates a block of statements as long as the logical_expression is TRUE
- Usage

```
while logical_expression
    block of statements
    ...
end
```

- Convert the geom_series.m code to work with a while loop

double for loop

```
%double for loop
%PC MA302
clc
clear
x=[1 2 -1 5 7 2 4];
y=[ 3 1 5 7];

m = length(x);
n = length(y);
A=zeros(m,n); %preallocate memory
for i_index =1:m
    for j_index =1:n
        A(i_index,j_index) = x(i_index)*y(j_index);
    end
end
disp(A);
```

double while loop

```
%while loop
clc
clear
x=[1 2 -1 5 7 2 4];
y=[ 3 1 5 7];
m = length(x);
n = length(y);
A=zeros(m,n); %preallocate memory
i_index=1;

while(i_index ≤m)
    j_index=1;
    while(j_index ≤n)
        A(i_index,j_index) = x(i_index)*y(j_index);
        j_index = j_index+1; %increment j_index
    end %i_index
    i_index = i_index +1; %increament i_index
end %i_index
disp(A);
```

while – prompting user for better input

```
%prompting user for better input
a = input('Enter a non-zero integer:');
while((a==0)||(round(a)≠a))
    fprintf('Your input is not a nonzero integer \n');
    a=input('Enter a non-zero integer:');
end

%%
%stop if input format is incorrect
%%
a = input('Enter a non-zero integer:');
while((a==0)||(round(a)≠a))
    error('Your input is not a nonzero integer, Try again');
end
```

Other useful commands

- `break` - terminates the execution of a `for` or `while` loop.
- `continue` - passes control to the next iteration of a `for` or `while` loop
- `return` - stops execution of the function or script before the end
- `error` - throws an error exception and displays a message

nargin, nargout, varargin, varargout

- nargin - returns the number of function **input** arguments given in the call to the current function.
- nargout - returns the number of function **output** arguments given in the call to the current function.

nargin, nargout, varargin, varargout

- `nargin` - returns the number of function **input** arguments given in the call to the current function.
- `nargout` - returns the number of function **output** arguments given in the call to the current function.
- `varargin` - an input variable that enables a function to **accept** any number of variables.
- `varargout` - output variable that allows a function to **return** any number of variables.

nargin, nargout, varargin, varargout

- `nargin` - returns the number of function **input** arguments given in the call to the current function.
- `nargout` - returns the number of function **output** arguments given in the call to the current function.
- `varargin` - an input variable that enables a function to **accept** any number of variables.
- `varargout` - output variable that allows a function to **return** any number of variables.

The MATLAB function `size` is a good example to illustrate multiple output options

nargout

If the user does not call `quadratic_solve` with 2 arguments output a vector

```
function [r1, r2] = quadratic_solve(a,b,c)
%solves ax^2+bx+c using quadratic formula

d = b^2-4*a*c;
r1 = -b + sqrt(d)/(2*a);
r2 = -b - sqrt(d)/(2*a);

%if the user enters 1 output argument
if nargout <2
    r1 = [r1,r2];
end
end
```

- Variable number of inputs see `quadratic_solve2.m`