The Department of Engineering Science
The University of Auckland

Chapter 3

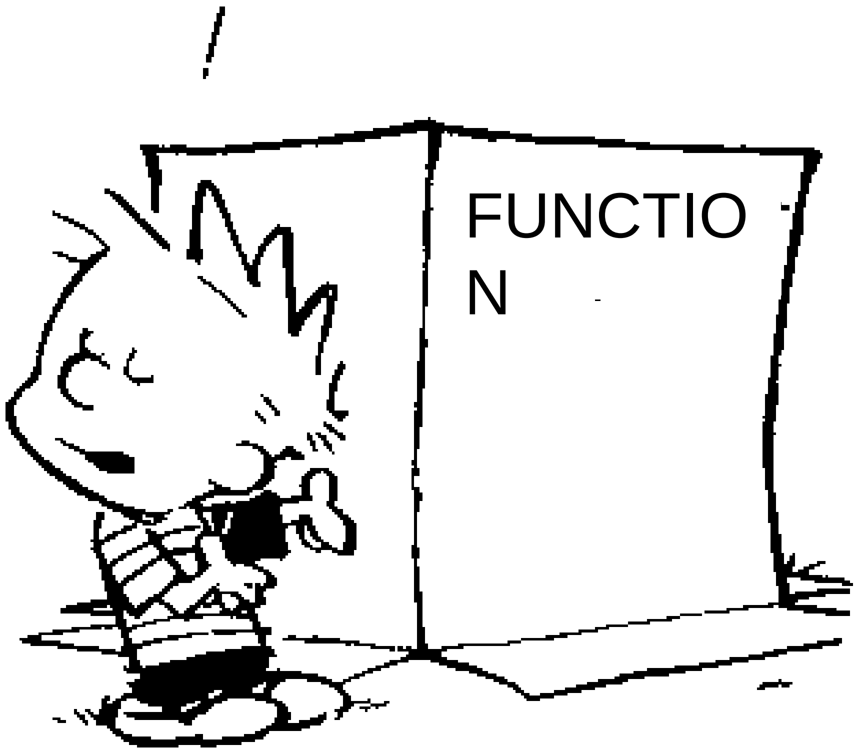
Functions and Debugging

Learning Outcomes

- Explain what a function is
- Call functions from your own programs
- Define your own functions
- Examine the function and command workspaces
- Debug script files and functions

What is a Function?

YOU PLACE INPUTS INTO THIS FUNCTION, WRITE THE APPROPRIATE CODE AND IT SPITS OUT WHATEVER OUTPUTS YOU WANT



What is a Function?

- Mathematical Function

function output → $y = f(x)$ ← function input (argument)

  - Takes input and transforms it into output

- MATLAB Function

```
y = func(x)
```

  - Examples

```
x = linspace(-pi, pi, 10);
length(x)
y = sin(x)
```

Why Use Functions?

- Enables "divide and conquer" strategy
  - Programming task broken into smaller tasks
- Code reuse
  - Same function useful for many problems
- Easier to debug
  - Check right outputs returned for all possible inputs
- Hide implementation

Black-Box

  - Only interaction via inputs/outputs, how it is done (implementation) hidden in function

Behaviour of a Function

- Functions should be well commented
  - Users must be able to find out how a function works
- Functions should be well defined
  - Given inputs should give known outputs
- Functions should be well tested
  - Inputs should always give correct outputs

Calling Functions

Functions can be called from command line or a script

To call a function we need to know:

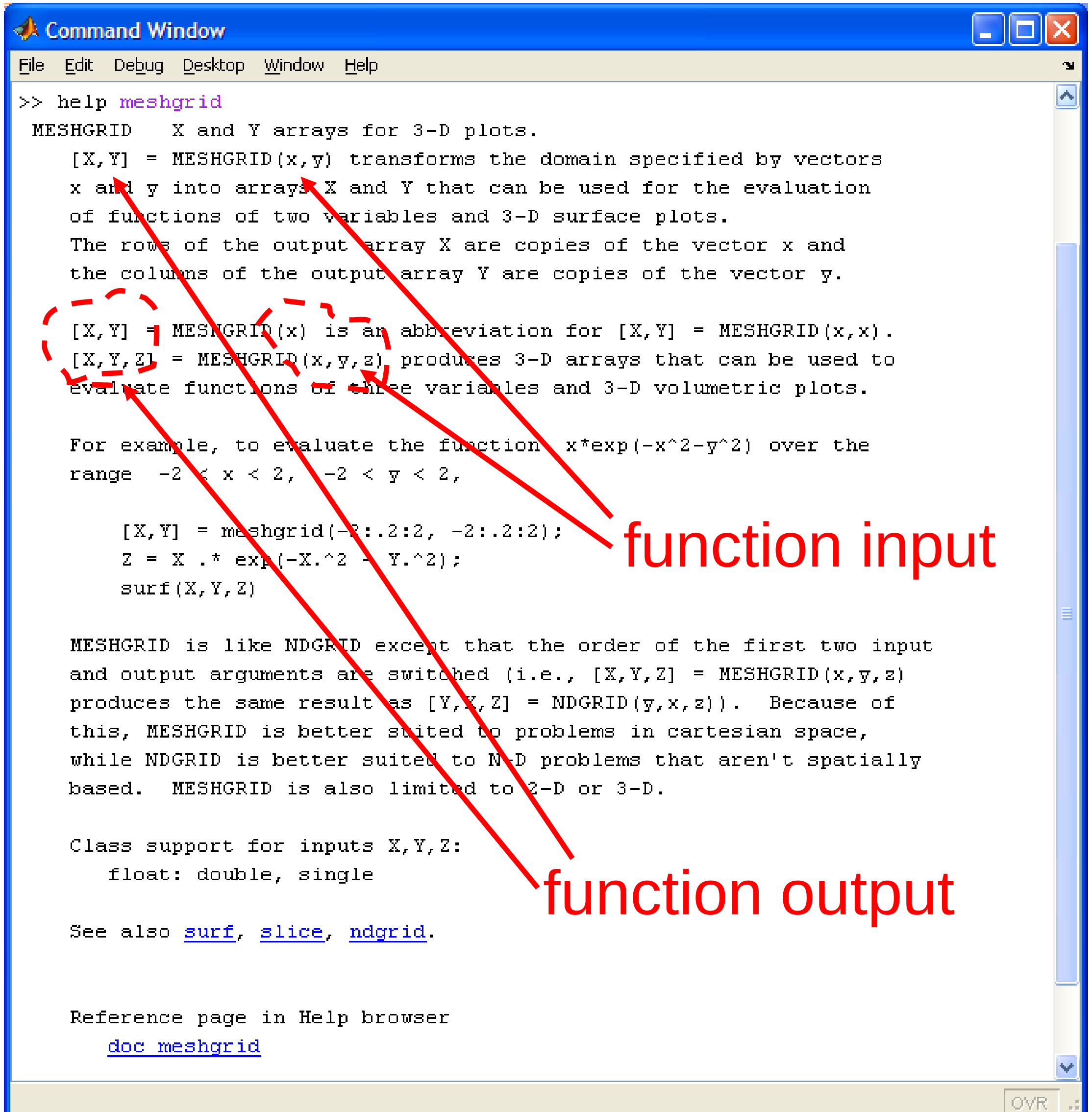
- The name of the function
- The function input(s)
- The function output(s)

Calling Functions

- Function names are case sensitive
  (meshgrid, meshGrid and MESHGRID are interpreted as different functioins)
- Inputs can be either numbers or variables

`y = sin(3);`
 	
`x = 3;`
`y = sin(x);`
	`y = min([3, 5, 1])`
	
	`a = [3, 5, 1]`
	`y = min(a)`

Calling Functions




```
>> help meshgrid
 MESHGRID   X and Y arrays for 3-D plots.
    [X,Y] = MESHGRID(x,y) transforms the domain specified by vectors
    x and y into arrays X and Y that can be used for the evaluation
    of functions of two variables and 3-D surface plots.
    The rows of the output array X are copies of the vector x and
    the columns of the output array Y are copies of the vector y.

    [X,Y] = MESHGRID(x) is an abbreviation for [X,Y] = MESHGRID(x,x).
    [X,Y,Z] = MESHGRID(x,y,z) produces 3-D arrays that can be used to
    evaluate functions of three variables and 3-D volumetric plots.

    For example, to evaluate the function  x*exp(-x^2-y^2) over the
    range  -2 < x < 2,  -2 < y < 2,

        [X,Y] = meshgrid(-2:.2:2, -2:.2:2);
        Z = X .* exp(-X.^2 - Y.^2);
        surf(X,Y,Z)

    MESHGRID is like NDGRID except that the order of the first two input
    and output arguments are switched (i.e., [X,Y,Z] = MESHGRID(x,y,z)
    produces the same result as [Y,X,Z] = NDGRID(y,x,z)).  Because of
    this, MESHGRID is better suited to problems in cartesian space,
    while NDGRID is better suited to N-D problems that aren't spatially
    based.  MESHGRID is also limited to 2-D or 3-D.

    Class support for inputs X,Y,Z:
       float: double, single

    See also surf, slice, ndgrid.


    Reference page in Help browser
       doc meshgrid
```

Calling Functions (inputs)

- Inputs are also called arguments
- Inputs are passed into the function inside of parentheses, separated by commas
- Order of input arguments is very important
- Name of input arguments can be anything you like

Calling Functions (outputs)

- The output is usually assigned to variable(s) so that it can be used
- If more than one variable is returned we use an array (square brackets)

```
[rows,cols] = size([3 2 1]);
```

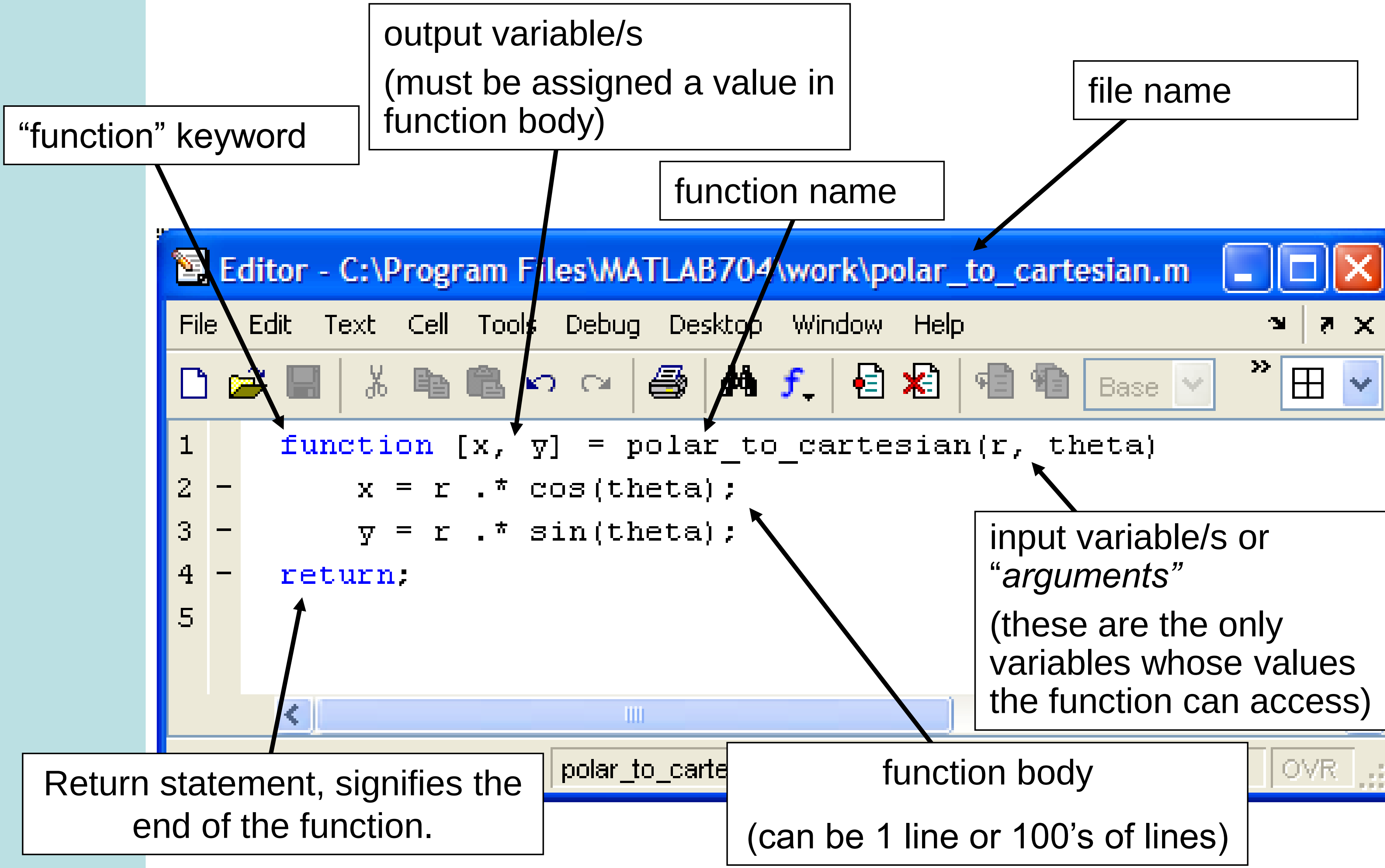
- If only one variable is returned we do not need an array

```
y=atan(0.5)
```

- Some functions have no outputs

```
plot(x,y)
```

Writing Functions



Different Inputs and Outputs

- Multiple outputs
  - No inputs `function [o1, o2, …] = myfunc()`
  - One input `function [o1, o2, …] = myfunc(i1)`
  - Multiple inputs `function [o1, o2, …] = myfunc(i1, i2, …)`
- One output
  - No inputs `function [o1] = myfunc()`
  - One input `function [o1] = myfunc(i1)`
  - Multiple inputs `function [o1] = myfunc(i1, i2, …)`
- No outputs
  - No inputs `function [] = myfunc()`
  - One input `function [] = myfunc(i1)`
  - Multiple inputs `function [] = myfunc(i1, i2, …)`

Function filenames

- All functions are saved to a file with a .m extension
- The filename (without the .m) must match EXACTLY the function name
- Function names may only use alphanumeric characters and the underscore
- Functions names should NOT:
  - include spaces
  - start with a number
  - use the same as an existing command
- Consider capitalising the first letter of a function name (a common convention)

Function headers

- All functions should have a header comment, just under the function defintion
- Header should describe
  - input(s) and output(s)
  - purpose of the function
  - who wrote it

```
function [f] = ConvertToFarenheit(c)
% ConvertToFarenenheit(c) takes a
temperature value c
% measured in degrees celsius and returns
the equilvalent
% value in farenheit
% Author: Peter Bier

f = 9/5 * c + 32;

return
```

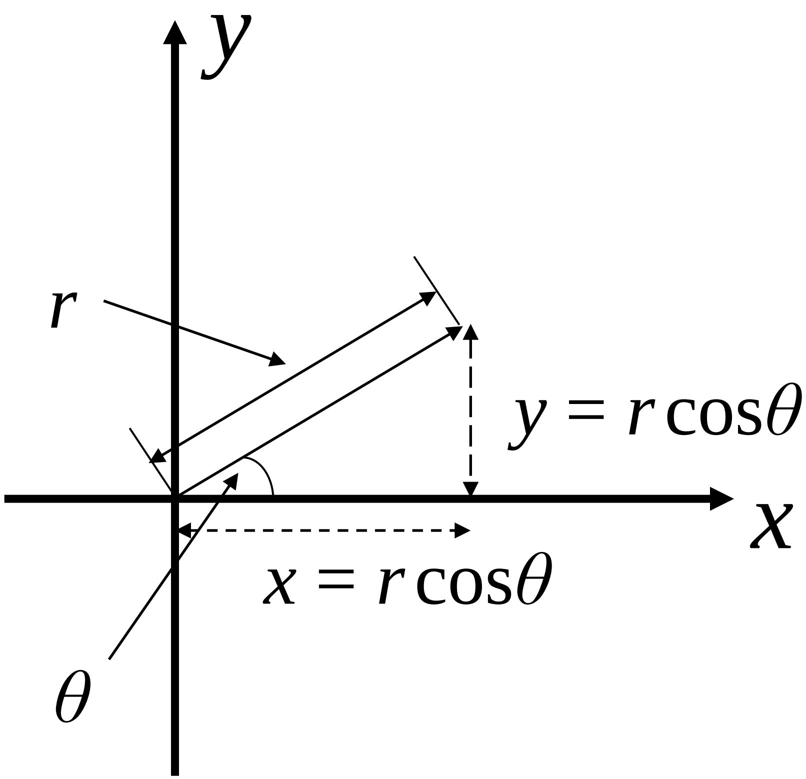
Function headers

- All functions should have a header comment, just under the function defintion
- Header should describe
  - input(s) and output(s)
  - purpose of the function
  - who wrote it
- Header comment becomes the function help

```
>> help ConvertToFarenheit

ConvertToFarenenheit(c) takes a temperature value c
measured in degrees celsius and returns the equilvalent
value in farenheit
Author: Peter Bier
```

Polar to cartesian example

- Polar coordinates useful for describing circular shapes
- Need to convert to Cartesian coordinates for plotting



Pseudocode

INPUTS: r and θ

1. Calculate x value
2. Calculate y value

OUTPUTS: x and y

The `PolarToCartesian` Function

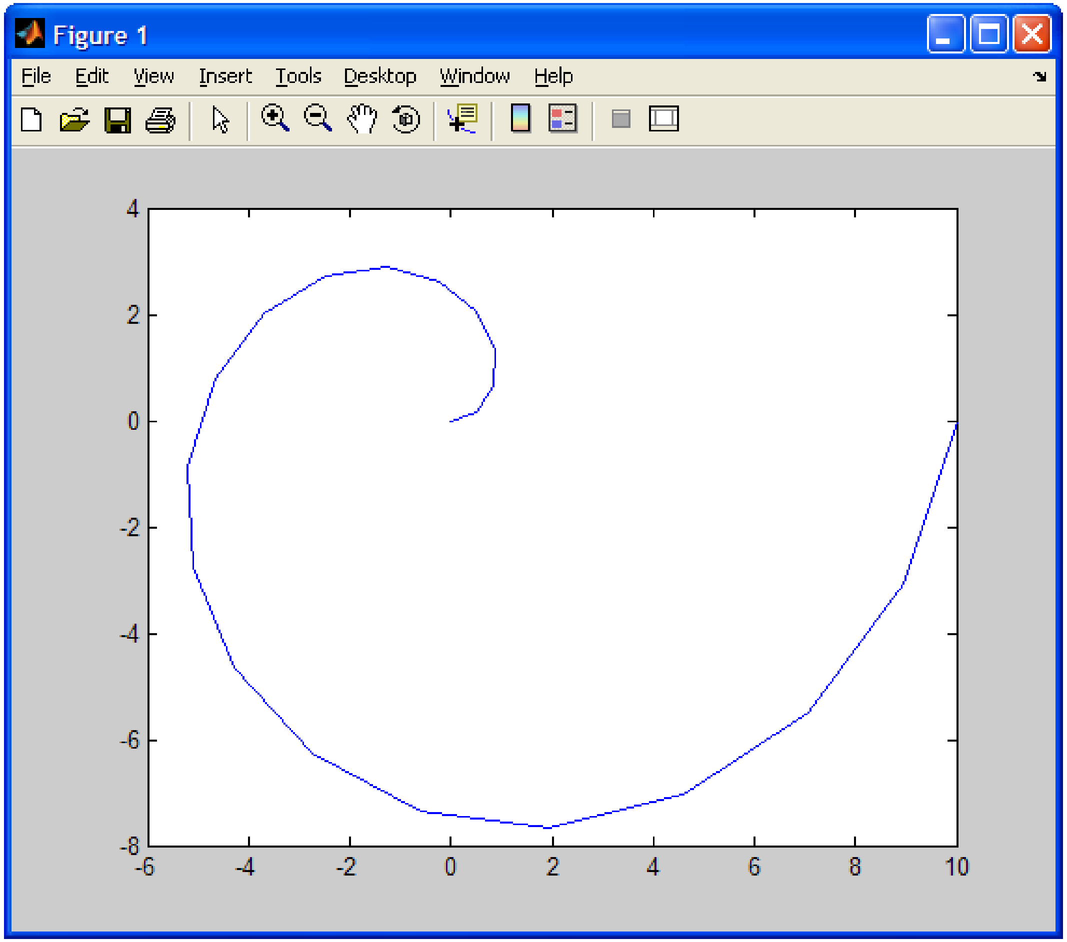
```
function [x, y] = PolarToCartesian(r, theta)
% PolarToCartesian transfroms r and theta from polar
% coordinates into (x,y) cartesian coordinates
% Inputs:  r          = radial distance
%             theta   = radial angle
% Outputs: x          = cartesian x coordinate
%              Y               = carteisan y coordinate
% Author: Peter Bier

% we use the dot operator so that our code will also work
% if r and theta are arrays.
% Note the use of the semi-colon to suppress output,
% otherwise our function will print ou the x and y values
% when calculating them
x = r .* cos(theta);
y = r .* sin(theta);

return;
```

Using our Function

- You use your functions exactly as if they were built-in MATLAB functions

```
% spiral.m draws a spiral using polar coordinates.
% Author: Peter Bier

% our array of 20 radius values will range from 0 to 10
spiralRs = linspace(0,10,20);
% our array of 20 theta values will range from 0 to 2pi,
% ie a full circle
spiralThetas = linspace(0, 2*pi, 20);

[x, y] = PolarToCartesian(spiralRs, spiralThetas);
plot(x,y);
```

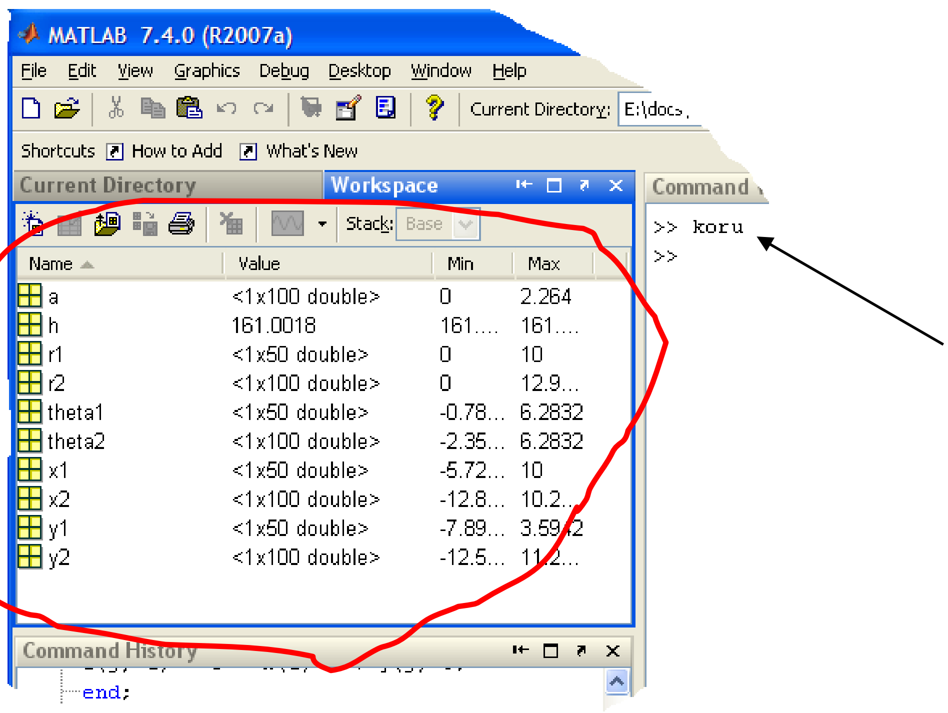
Using our Function



The Matlab Workspace

- When you create variables in Matlab
  - Via the command window
  - In script files

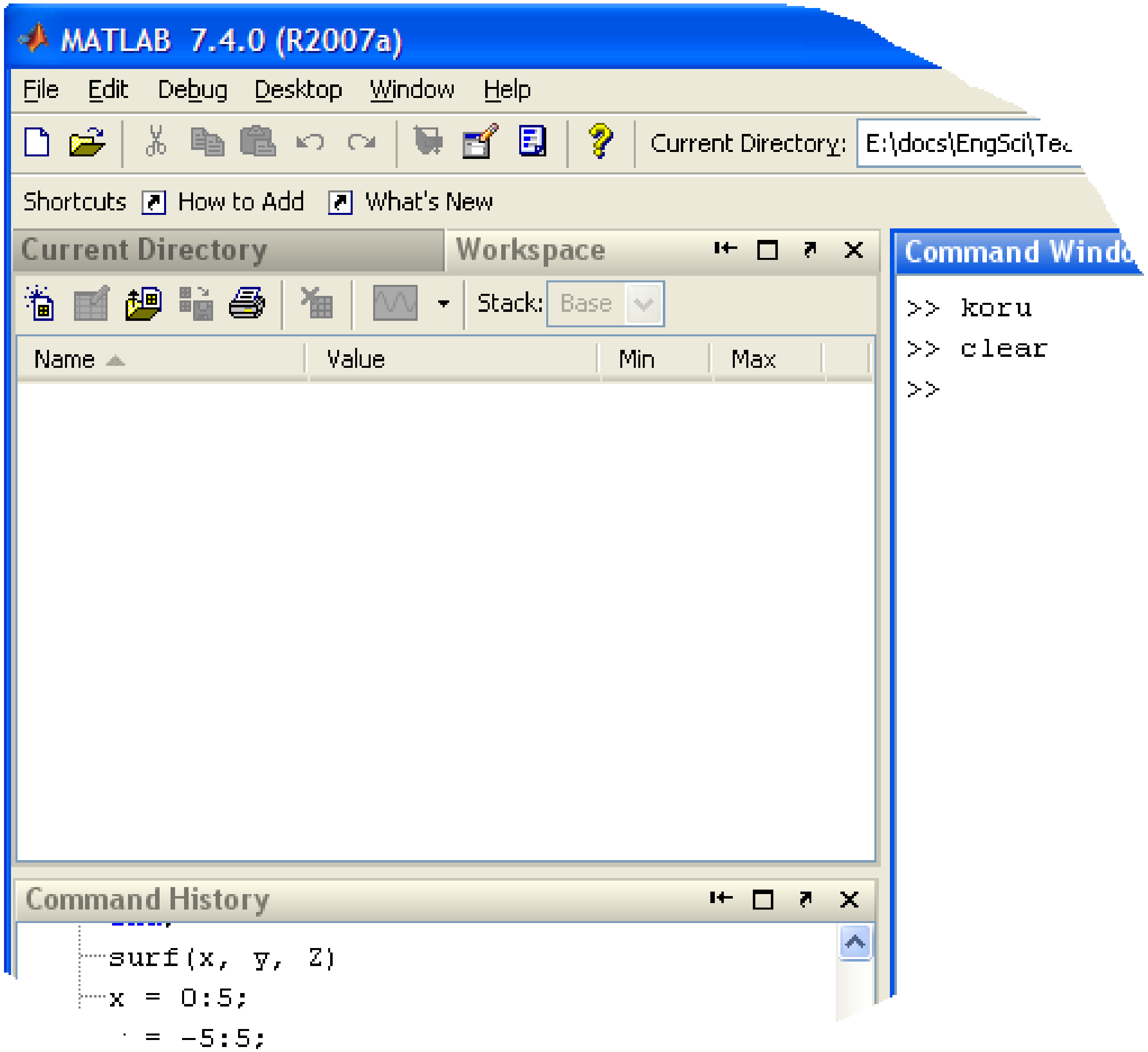
  Matlab stores them in the "workspace"



The `koru.m` script files creates the variables in the workspace

Starting Over

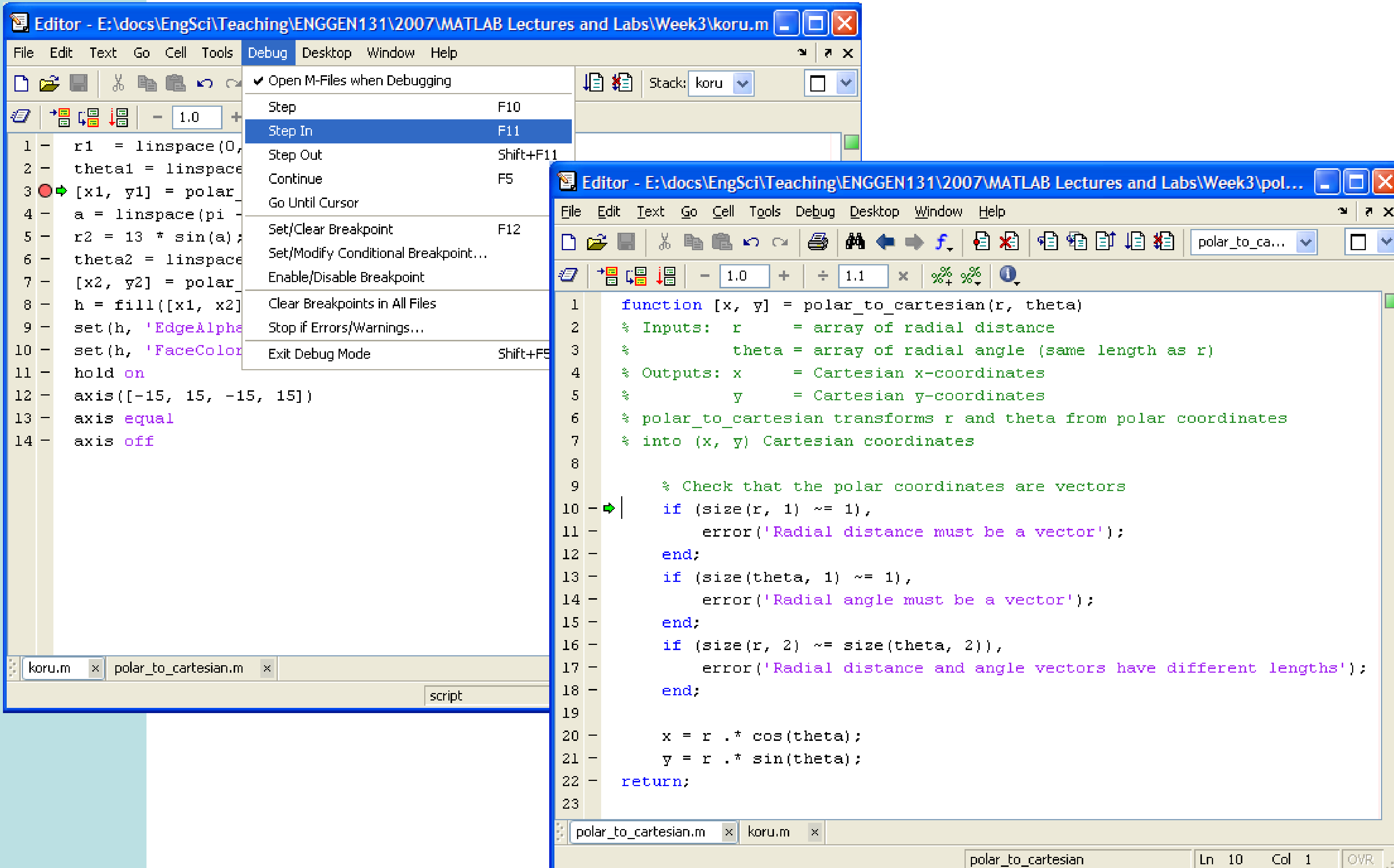
- Matlab can be `clear`ed



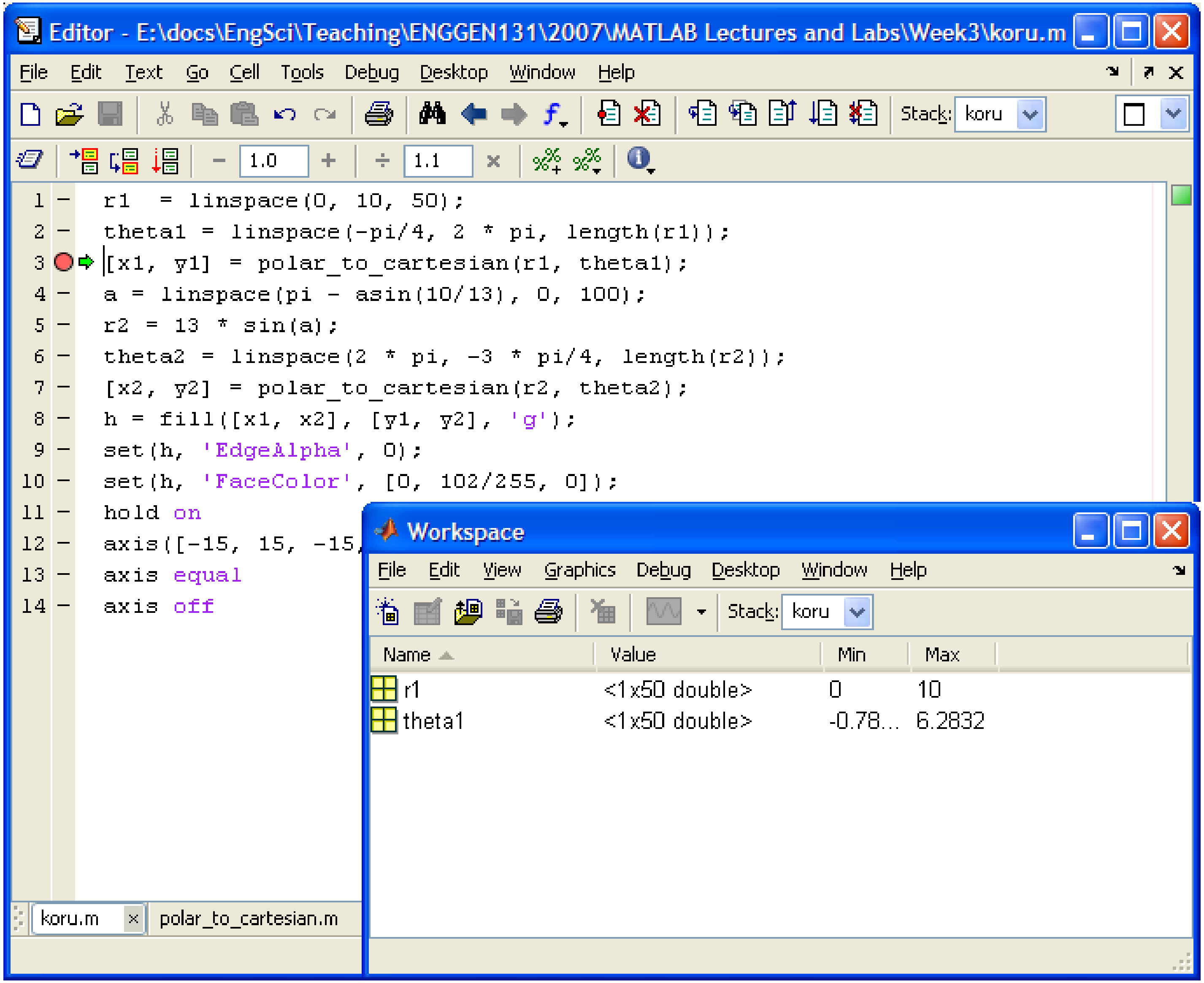
Function Workspaces

- Functions create their own workspaces
- Function inputs are also created in workspace when function starts
- Function doesn't know about any variables in any other workspace
- Function outputs are copied from workspace when function ends
- Function workspaces are destroyed after functions end
  - Any variables created in function "disappear" when function ends

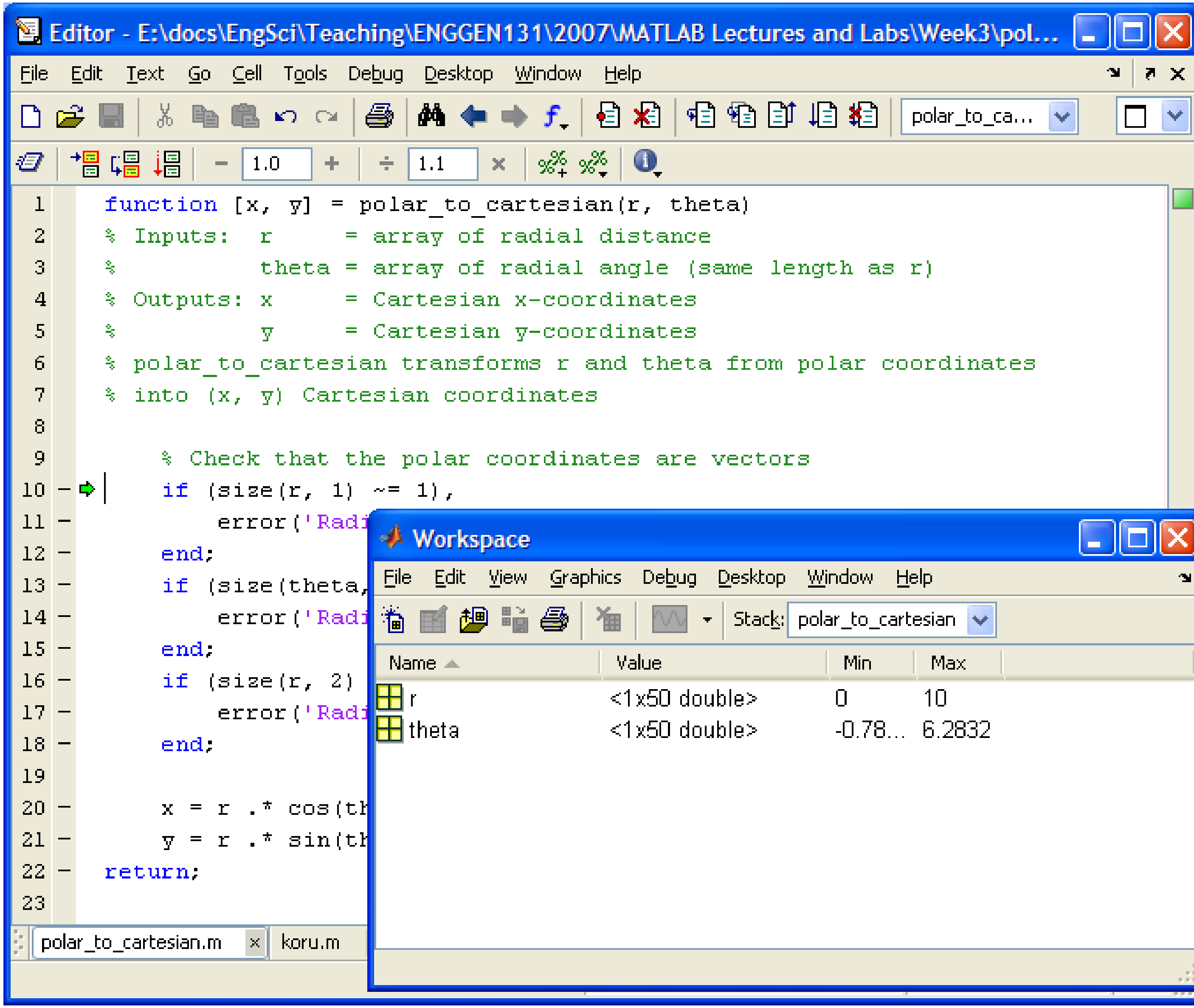
Debugging: Stepping In



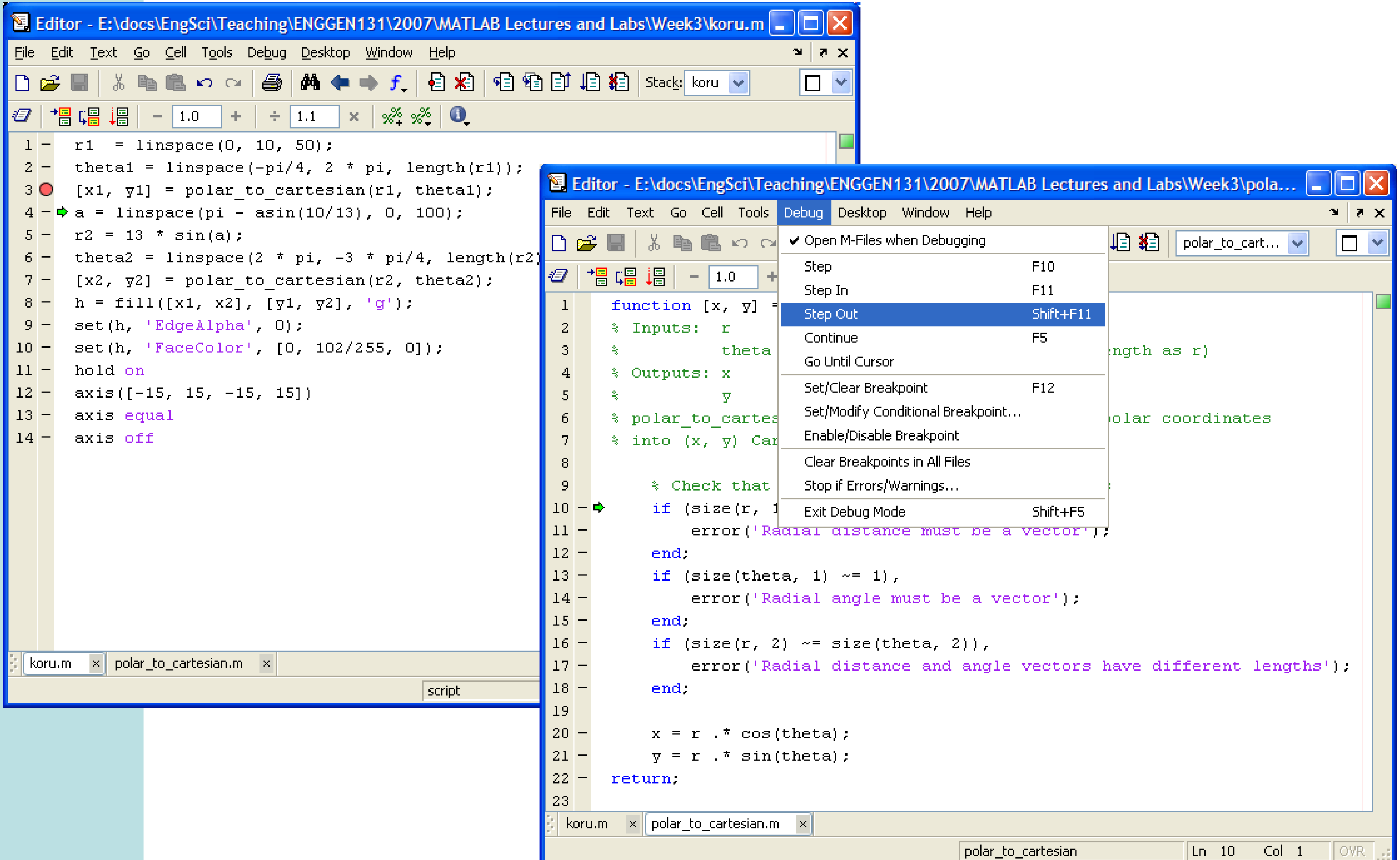
Debugging: Matlab Workspace



Debugging: Function Workspace



Debugging: Stepping Out



Recommended Reading

Chapter 3 Functions, Problem Solving and Debugging	Introduction to Matlab 7 for Engineers (2nd ed)		A Concise Introduction to Matlab (1st ed)	
Topic	**Section**	**Pages**	**Section**	**Pages**
Debugging	1.4	32-34	1.4	25-26
Workspaces	2.1	80-81	1.2	47-48
Debugging	4.7	228-233	4.6	184-188
Writing functions	3.2	148-152	1.4	126-130

The Department of Engineering Science
The University of Auckland

Chapter 4

Logical Operators and Conditional Statements

Learning Outcomes

- Use pseudocode and flow charts to describe programs
- Understand relational and logical operators
- Understand conditional statements
- Create and use boolean variables
- Control program flow with conditional statements and boolean variables

Controlling your Computer

- A computer program is a sequence of simple steps.
- Each step does one thing.

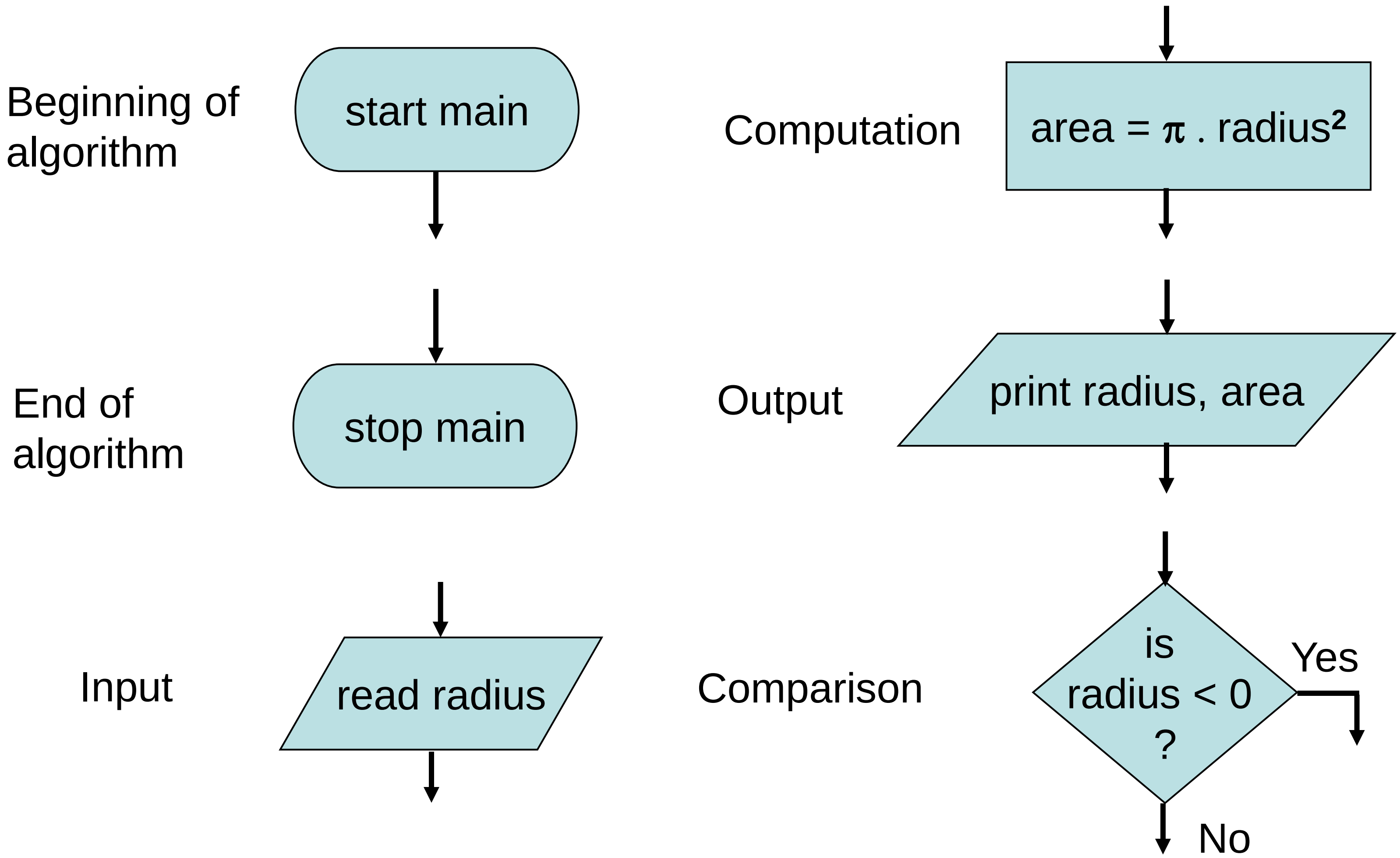
- Before writing a program, we need a plan
- A plan helps us focus on the problem, not the code
- Once we have written a plan, the plan can be implemented in whatever language we want to use (Matlab, C, Java, Perl, Python, etc)

- Two common ways of writing a plan are **pseudocode** and **flowcharts**

Pseudocode and Flowcharts

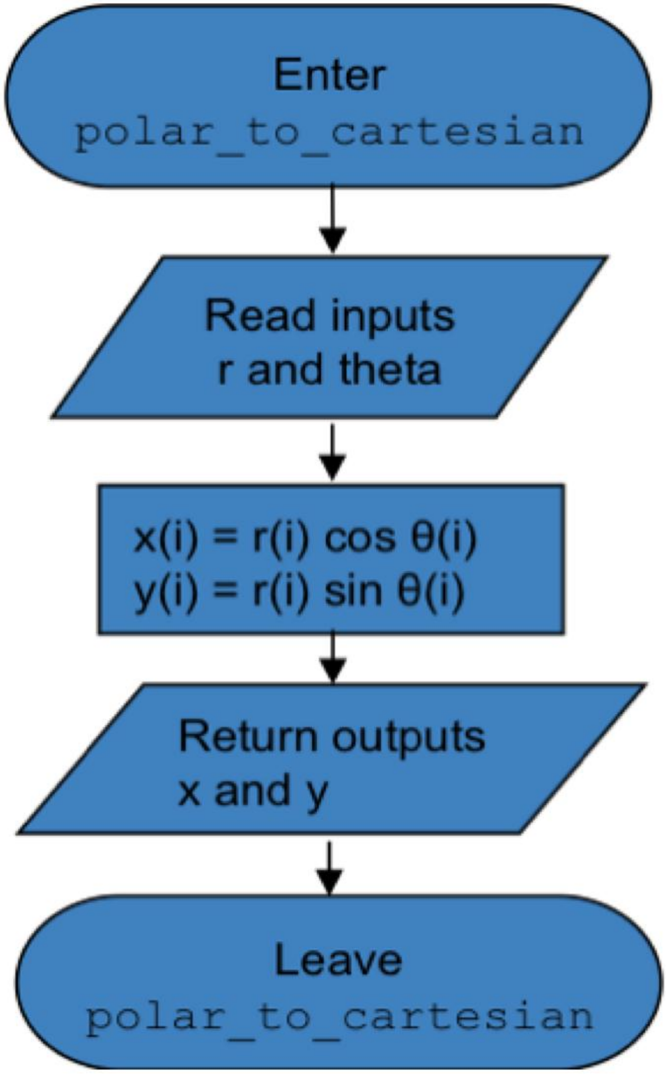
- Pseudocode
  - Text description of program steps
    - May contain fragments of code
    - Doesn’t contain the nitty-gritty details
  - Similar to a recipe
- Flowcharts
  - Geometric symbols to describe program steps
  - Captures “flow” of program
- Both are useful for any programming language.

Flowchart Elements



From Etter Figure 3.1 page 87

Flow charting functions



Programming Example

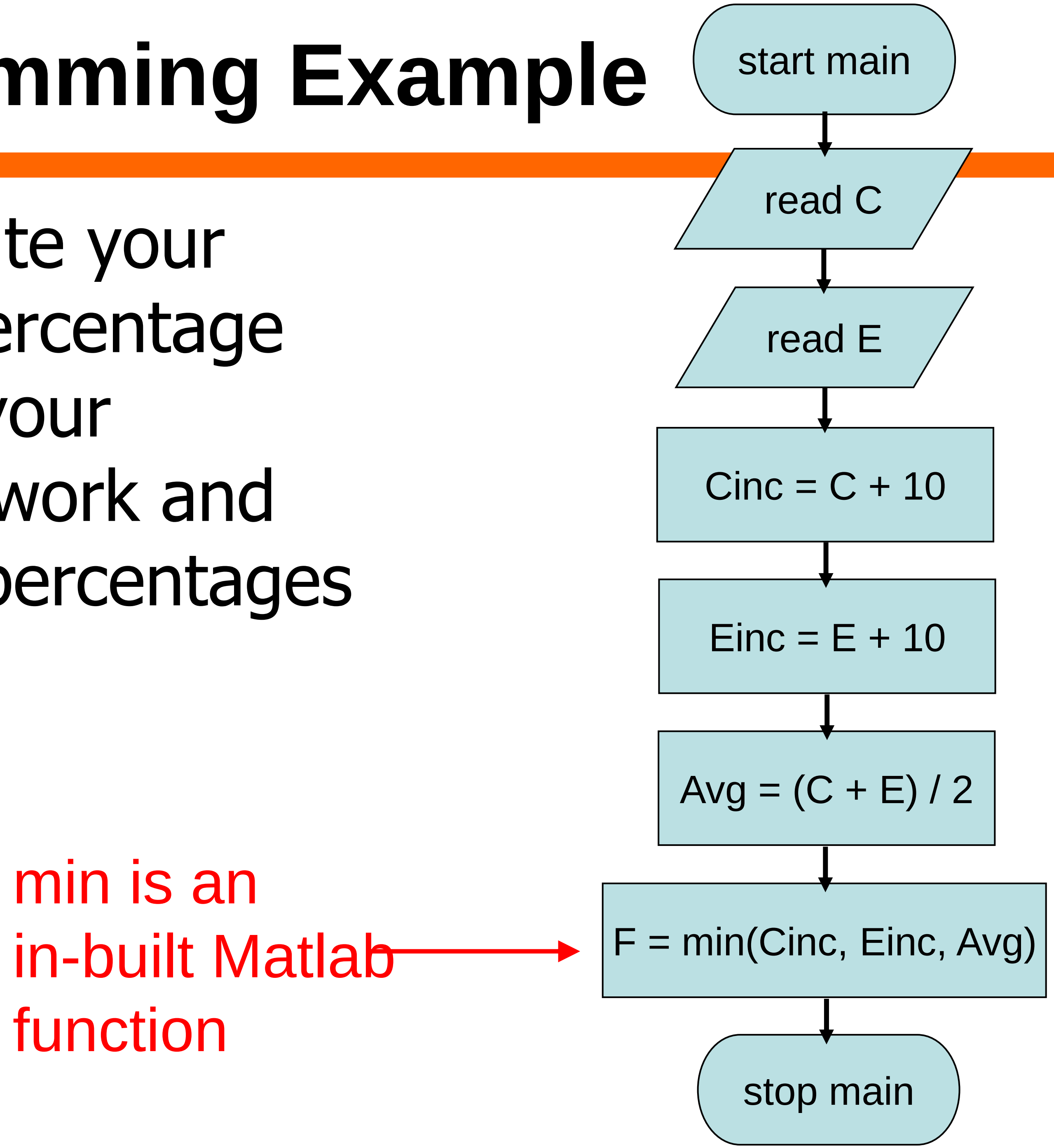
- Calculate your final percentage given your coursework and exam percentages

Pseudocode

1. Get coursework percentage C
2. Get exam percentage E
3. Calculate C + 10
4. Calculate E + 10
5. Calculate (C + E) / 2
6. Set final percentage F to be minimum of C + 10, E + 10, (C + E) / 2

Programming Example

- Calculate your final percentage given your coursework and exam percentages



MATLAB Program

```matlab
% This script file calculates your 131 final percentage
% from your coursework percentage and your exam percentage
% Inputs: C = coursework percentage
%         E = exam percentage
% Output: F = final percentage
clear;

% Get coursework percentage C
C = input('Enter coursework percentage > ');

% Get exam percentage E
E = input('Enter exam percentage > ');

% Calculate C + 10
Cinc = C + 10;

% Calculate E + 10
Einc = E + 10;

% Calculate (C + E) / 2
Avg = (C + E) / 2;

% Set final percentage F to be minimum of C + 10, E + 10, (C + E) / 2
F = min([Cinc, Einc, Avg])
```

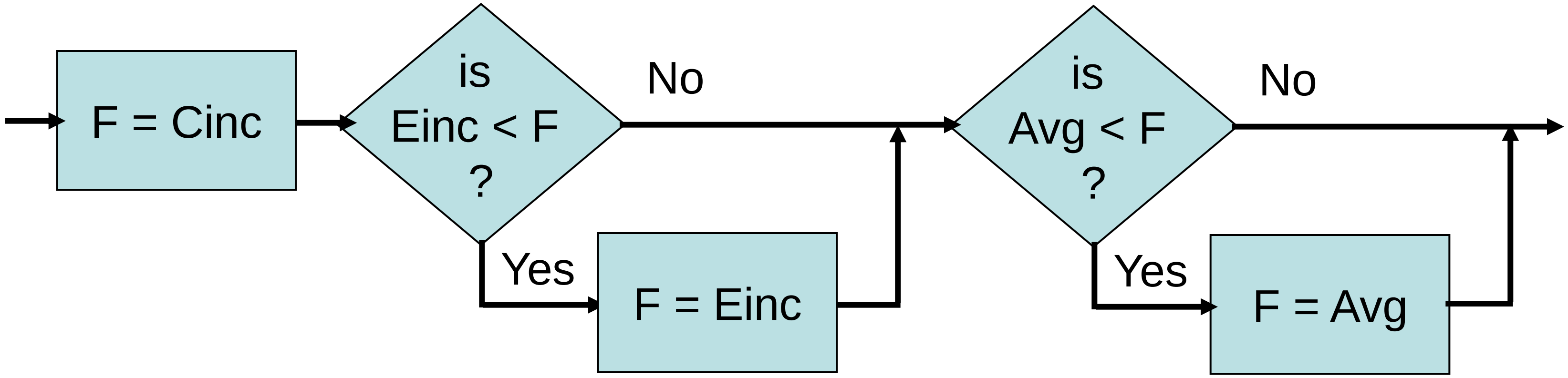
Note that pseudocode makes good comments

C Program

```c
#include <stdio.h>

#define min(X,Y) ((X) < (Y) ? (X) : (Y)) // Define the min function

int main(int argc, char* argv[]) {
/** This script file calculates your 131 final percentage
 *  from your coursework percentage and your exam percentage
 */
    double C, E, Cinc, Einc, Avg, F;

    // Get coursework percentage C
    printf("Enter coursework percentage > ");
    scanf("%lf", &C);

    // Get exam percentage E
    printf("Enter exam percentage > ");
    scanf("%lf", &E);

    // Calculate C + 10
    Cinc = C + 10;

    // Calculate E + 10
    Einc = E + 10;

    // Calculate (C + E) / 2
    Avg = (C + E) / 2;

    // Set final percentage F to be minimum of C + 10, E + 10, (C + E) / 2
    F = min(Cinc, min(Einc, Avg));
    printf("Final percentage = %g\n", F);

    return 0;
}
```

Implementing Min

- What if min was not an in-built function?
  - Need comparisons and logic
- Pseudocode
  1. F = Cinc
  2. If Einc < F, set F = Einc
  3. If Avg < F, set F = Avg
- Flowchart



Relational Operators

- Relational operators test relationships between variables

`A == B`	tests whether A equals B
`A ~= B`	tests whether A does not equal B
`A < B`	tests whether A is less than B
`A > B`	tests whether A is greater than B
`A <= B`	tests whether A is less than or equal to B
`A >= B`	tests whether A is greater than or equal to B

Using Relational Operators

```
>> a=3;
>> b=4;
>> a==b
ans =
     0   ← false
>> a~=b
ans =
     1   ← true
>> a<b
ans =
     1
```

```
>> a>b
ans =
     0
>> a<=b
ans =
     1
>> a>=b
ans =
     0
```

Logical Operators

- Logical operators are test conditions
  - expressed as relationships between variables
  - 0 is false, everything else is treated as true

- Common logical operators

~p	true if p is not true
p & q	true if both p and q are true
p | q	true if either p or q are true

Using Logical Operators

```
>> a=3;
>> b=4;
>> ~(a==b)
ans =
     1
>> c=5;
>> (a<b) & (b<c)
ans  =
     1
```

```
>> (a>b) | (a>c)
ans =
     0
>> (a>b) | (b<c)
ans =
     1
```

Conditional Statements

- Dissecting the conditional:
  - "If it is sunny outside then I will cycle to uni."

    condition **dependent**

  - "If I pass the exam then I will be happy."

    condition **dependent**

  – Using MATLAB
  - "If `e > 50` then `disp('happy')`"

    condition **dependent**

`if … end`

- Syntax

```
if condition
    some commands
end
```

dependent

`end` lets MATLAB know when conditional statement is finished

`if … end`

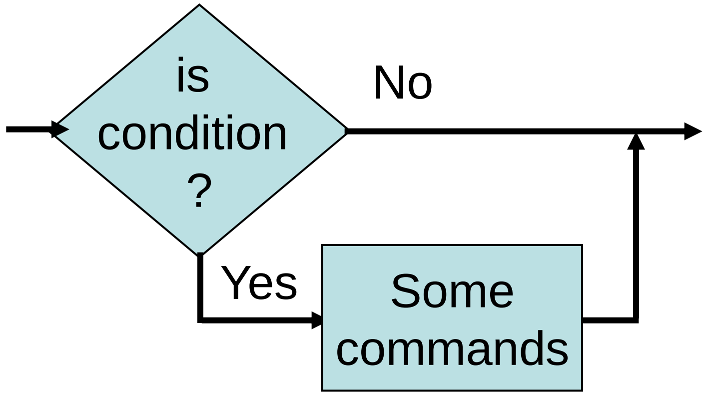
Pseudocode

1. If *conditon, some commands*

Alternative Pseudocode

1. If *condition*
    a) *Some commands*

Flowchart



`if ... end` Example

File: myif.m

```
a=2;
b=5;
if a<b
    disp(a)
end
```

Matlab command prompt

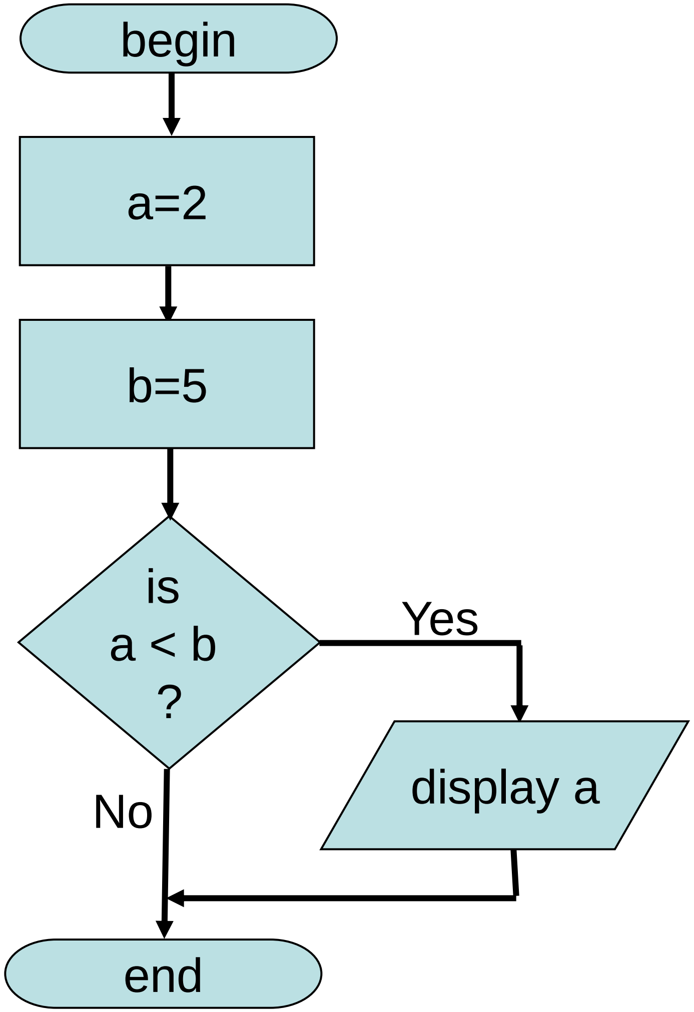
```
>> myif
     2
>>
```

Describing myif.m

<u>Pseudocode</u>

1. Set a = 2
2. Set b = 5
3. If a < b
   a) Display a

<u>Flowchart</u>



`if … else … end`

- Syntax

```
if condition
   some commands
else
   some other commands
end
```

This section is OPTIONAL

`end` lets MATLAB know when the conditional statement is finished

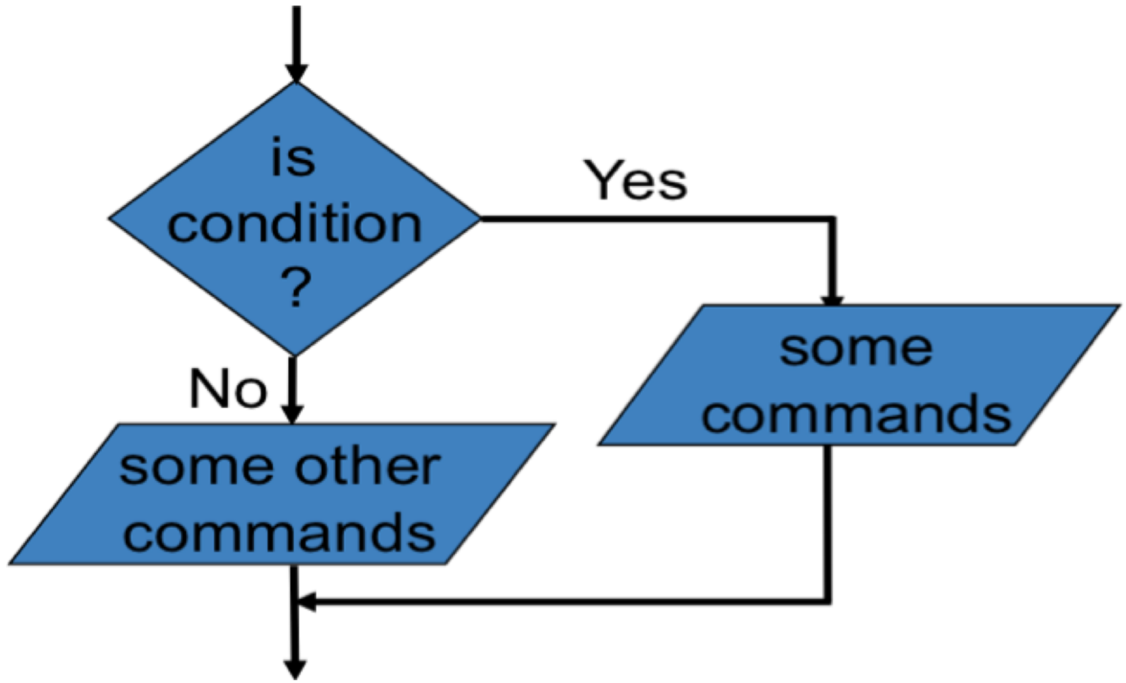
`if ... else ... end`

- Pseudocode

1. If *condition*
   a) *Some commands*
2. *Else*
   a) Some other commands

`if … else … end`

- Flowchart



`if ... else ... end` Example

File: myifelse.m

```
a=5;
b=4;
if a<b
    disp(a)
else
    disp(b)
end
```

Matlab command prompt

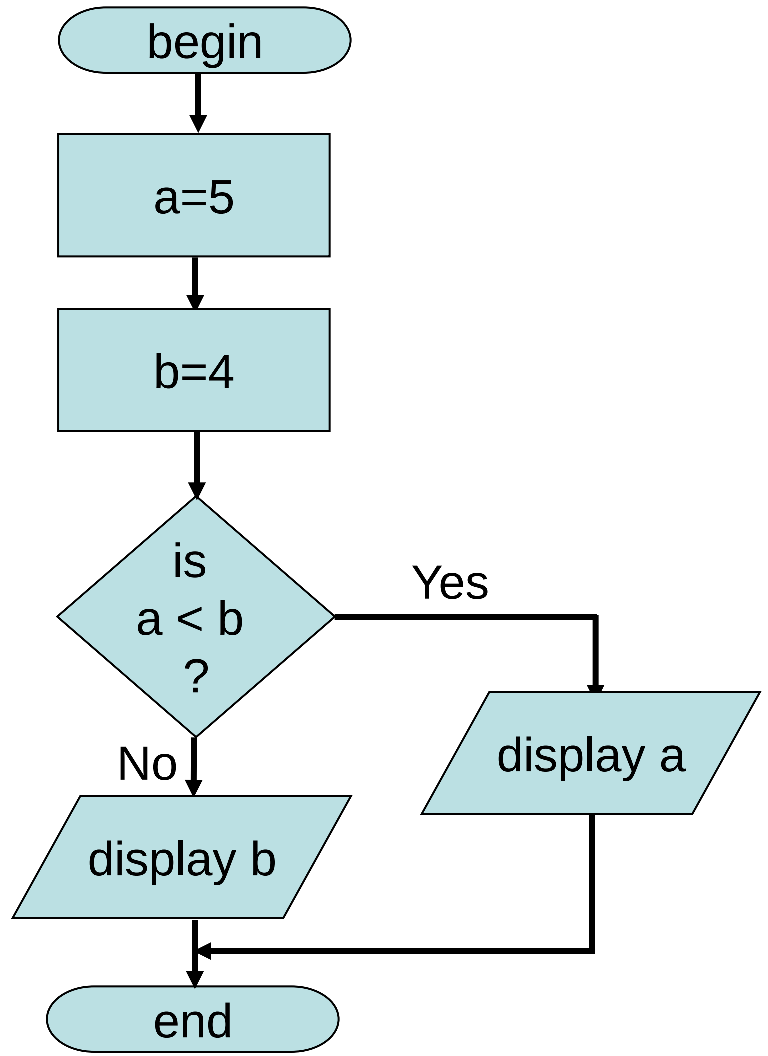
```
>> myifelse
     4
>>
```

Describing myifelse.m

Pseudocode

1. Set a = 5
2. Set b = 4
3. If a < b
    a) Display a
4. Else
    a) Display b

Flowchart



`if ... elseif ... else ... end`

- Syntax

```
if condition
  some commands
elseif another condition
  some different commands
else
  some other commands
end
```

This section is OPTIONAL

`end` lets MATLAB know when the conditional statement is finished

`if … elseif … else … end`

- Psuedocode

1. If *condition*
   a) *Some commands*
2. *Else If another condition*
   a) Some different commands
3. Else
   a) some other commands

`if` … `elseif` … `else` … `end` Example

File: myelseif.m

```
a=5;
b=4;
if a==b,
        disp(a)
        disp(b)
elseif a<b,
        disp(a)
else
        disp(a)
end
```

Matlab command prompt

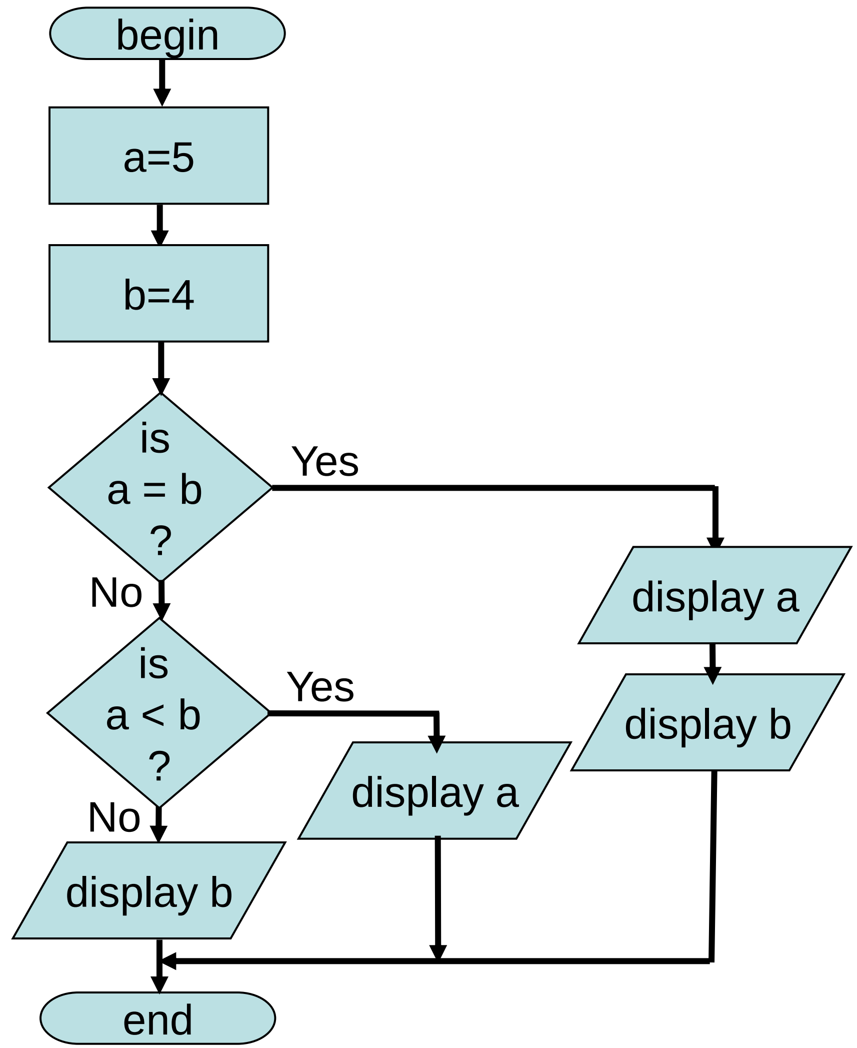
```
>> myelseif
     4
>>
```

Describing myIfElseIfElse.m

Pseudocode

1. Set a = 5
2. Set b = 4
3. If a == b
   a) Display a and b
4. Else if a < b
   a) Display a
5. Else
   a) Display b

Flowchart



Testing Multiple Conditions

- Suppose we want to know if `a < b < c`

  `if (a < b < c)` ✗ Two relationships!!

- `a < b` and `b < c`
- Test whether both are true.

```
if (a<b) & (b<c)
  disp('b is between a and c')
end
```

Note use of "&" to test whether both conditions are true

More on precedence

- Suppose we want to check if a is less than b and c
- This statement is ambiguous and can be interpreted in a few ways
- Use brackets to clarify which meaning

check if a is less than b and c

- test that the value of a is less than b AND that the value of a is less than c

  (a < b) & (a < c)

- find the value of b & c and test to see if a is less than this

  a < (b & c)

- find the value of a<b and see if both this value and c are true

  (a < b) & c

Implementing Min Again

```
% myMin - finds min(Cinc, Einc, Avg)
function [F] = myMin(Cinc, Einc, Avg)

% If Cinc < Einc and Avg, set F = Cinc
if (Cinc < Einc) & (Cinc < Avg),
    F = Cinc;
% If Einc < Cinc and Avg, set F = Einc
elseif (Einc < Cinc) & (Einc < Avg),
    F = Einc;
% If Avg < Cinc and Einc, set F = Avg
else % Must be true by default
    F = Avg;
end;

return;
```

Boolean Variables

- Boolean variables used store “true” and “false” values.
- Very useful with relational operators and conditional statements.
- MATLAB uses 1 to represent true
  - Actually any NONZERO

  and 0 to represent false.

Boolean Variables

- Create boolean variables just like other variables:

```
isFinished = 1 % true

isFound    = 0 % false
```

Variable name

Variable value

- Use boolean variables to store answer to some "question" that controls a conditional statement or *while loop.*

Using Boolean Variables

- Good programming practice to choose a variable name which indicates what kind of value is stored in the variable
- For booleans it is common practice to start the variable name with the word 'is'
- Use meaningful boolean names, eg `isSuccessful` rather than `status`
- Try to write statements which read well.

```
if isSuccessful        ✓
    do something
end
```

```
if ~isFailure          ✗
    do something
end
```

Using Boolean Variables

- Choose a name that is a question with a yes/no or true/false answer

```
if ( atUniversity & stillAStudent)
  needMoreMoney = 1;
end
```

Recommended Reading

Chapter 4 Logical operators and conditional statements	Introduction to Matlab 7 for Engineers (2nd ed)		A Concise Introduction to Matlab (1st ed)	
Topic	**Section**	**Pages**	**Section**	**Pages**
Relational operators and conditional statements	1.6	44-48		
Relational operators	4.2	191-192	4.1	153-155
Logical operators	4.3	194-197	4.2	156-160
Conditionals	4.4	201-208	4.3	163-167