

Today's Objectives

- reference material on I/O and plotting
- Program logic
- Download the files from github
- We will repeat the accessing and plotting the sounding data

`disp` (display) command displays variable values or text on screen

- Displays each time on new line
- Doesn't print variable name

`disp(variable_name)` or

`disp(variable_name, 'text string')`

fprintf


- Means file print formatted
 - *formatted text* is text that can be read by people
 - *unformatted text* looks random to people but computers can read it
- Can write to screen or to a file
- Can mix numbers and text in output
- Have full control of output display
- Complicated to use

`\n` is an *escape character*, a special combination of two characters that makes `fprintf` do something instead of print the two characters

`\n` – makes following text come out at start of next line

`\t` – horizontal tab

```
>> fprintf( 'Joe is %d weighs %f kilos', age, weight )
```

A diagram with two red curved arrows. The first arrow starts under the format specifier '%d' in the string and points to the variable 'age'. The second arrow starts under the format specifier '%f' and points to the variable 'weight'.

Arguments

- Number of arguments and conversion specifiers must be the same
- Leftmost conversion specifier formats leftmost argument, 2nd to left specifier formats 2nd to left argument, etc.

Conversion specifier

```
>> fprintf( 'Joe weighs %f kilos', n1 )
```



Common conversion specifiers

- %f fixed point (decimal always between 1's and 0.1's place, e.g., 3.14, 56.8)
- %e scientific notation, e.g, 2.99e+008
- %d integers (no decimal point shown)
- %s string of characters

Conversion specifier

```
>> fprintf( 'Joe weighs %6.2f kilos', n1 )
```

To control display in fixed or scientific,
use `%w.pf` or `%w.pe`

- `w` = width: the minimum number of characters to be displayed
- `p` = “precision”: the number of digits to the right of the decimal point

If you omit “`w`”, MATLAB will display correct precision and just the right length

Example

```
>> weight = 178.3;
```

```
>> age = 17;
```

```
>> fprintf( ['Tim weighs %.1f lbs'...  
' and is %d years old'], weight, age )
```

```
Tim weighs 178.3 lbs and is 17 years old
```


Using the `fprintf` command to save output to a file:

Takes three steps to write to a file

Step a: – open file

```
fid=fopen('file_name','permission')
```

`fid` – *file identifier*, lets `fprintf` know what file to write its output in

`permission` – tells how file will be used, e.g., for reading, writing, both, etc.

Some common permissions

- `r` - open file for reading
- `w` - open file for writing. If file exists, content deleted. If file doesn't exist, new file created
- `a` - same as `w` except if file exists the written data is appended to the end of the file
- If no permission code specified, `fopen` uses `r`

See Help on `fopen` for all permission codes

Step b:

Write to file with `fprintf`. Use it exactly as before but insert `fid` before the format string, i.e.,

```
fprintf(fid, 'format string', variables)
```

The passed `fid` is how `fprintf` knows to write to the file instead of display on the screen

Step c:

When you're done writing to the file, close it with the command

```
fclose(fid)
```

- Once you close it, you can't use that `fid` anymore until you get a new one by calling `fopen`
- Be sure to close every file you open.

```
output_file = input('enter output file name:  
\n','s')
```

```
id = fopen(output_file,'w');
```

```
%use the transpose of the winds array to  
plot out the rows, rather than all
```

```
%one variable then the other
```

```
fprintf(id,'%.2f\t %.2f\n',winds');
```

```
fclose(id);
```

- Look at file you created using text editor

`fprintf` is *vectorized*, i.e., when vector or matrix in arguments, command repeats until all elements displayed

- Displays matrix data column by column and row by row

```
0.58  0.61  0.54
0.48  0.45  0.68
0.56  0.60  0.64
0.51  0.44  0.51
0.48  0.58  0.51
0.41  0.49  0.62
0.52  0.56  0.58
-0.66 0.59  0.41
-0.64 -0.34      -0.57
0.63  -0.51      0.55
-0.39 0.36  -0.53
0.39  -0.76      0.73
-0.42 -0.47      0.62
-0.51 0.56  -0.44
-0.54 0.46  0.36
-0.63 -0.36      -0.40
0.54  -0.41      0.41
-0.37 0.40  -0.50
0.39  -0.63      0.69
-0.49 -0.46      0.48
-0.47 0.48  -0.36
```

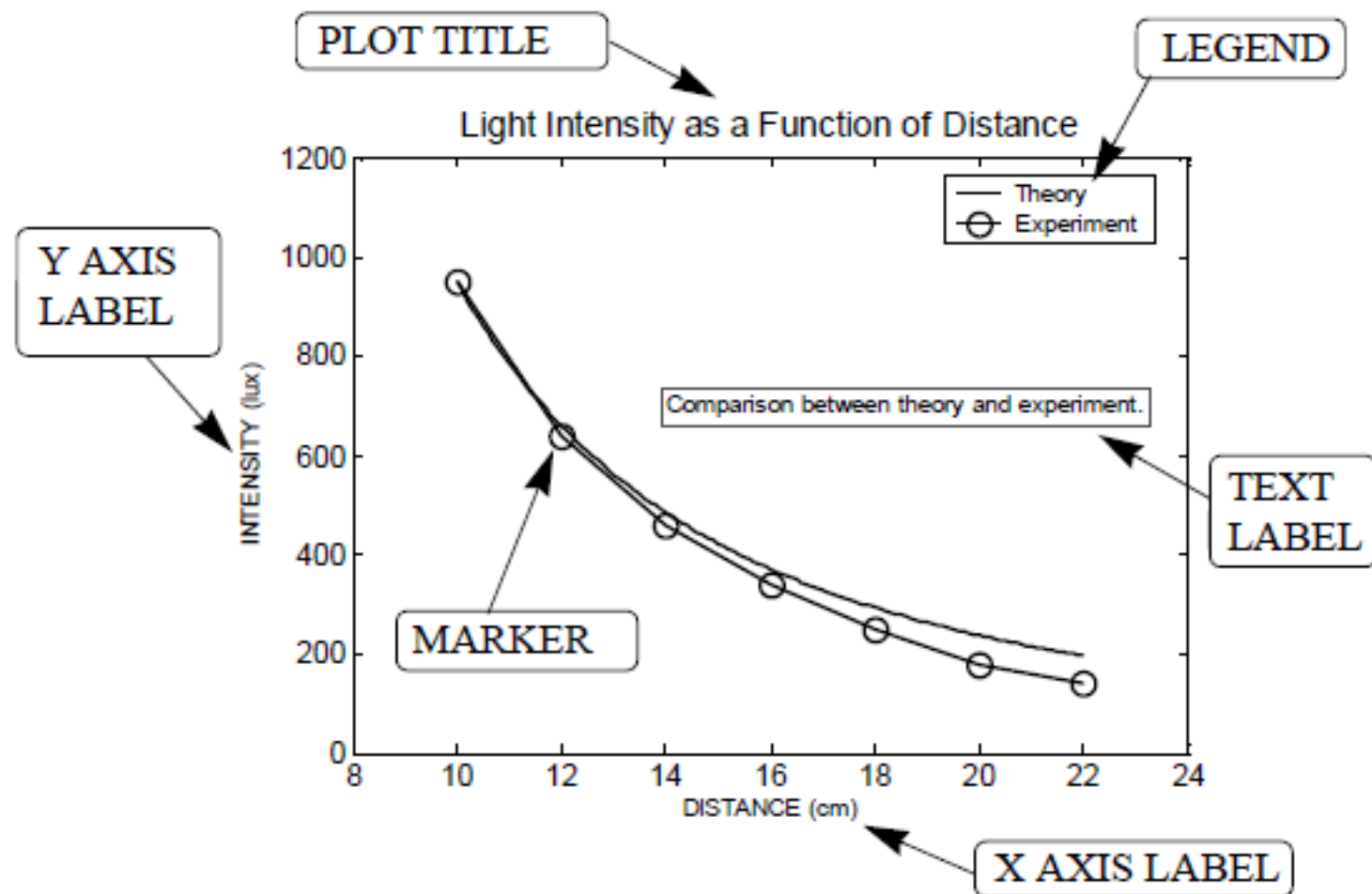


Figure 5-1: Example of a formatted two-dimensional plot.

On execution, any plotting command

1. Creates a Figure Window (if none exists)
2. Erases any plot in that window
3. Draws new plot

Can be useful though to have plots in multiple windows. `figure` command lets you do this

`figure (n)`

- If Figure Window `n` exists, makes it the active window
- If Figure Window `n` doesn't exist, creates it and makes it the active window
- `figure (n)` useful in scripts, e.g., scripts in which data set 1 is displayed in Figure 1, data set 2 is displayed in Figure 2, etc.

Use `close` command to close figure windows

- `close` closes active Figure Window
- `close (n)` closes Figure Window `n`
- `close all` closes all open Figure Windows

`plot` command used to make basic 2D plots. Simplest form is

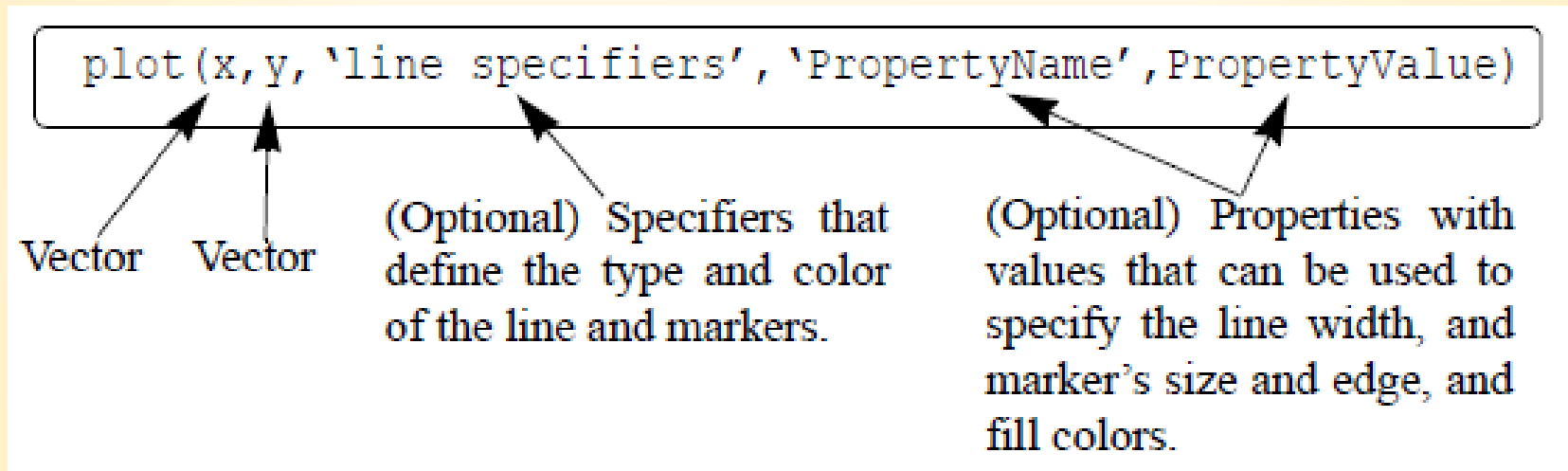
```
plot (y)
```

- Plots vector `y` on vertical axis, numbers 1 through `N` on horizontal axis (`N` = number of points in `y`)
- If there's a Figure Window, draws in it. Otherwise, creates a new Figure Window and draws in that

`plot(y)` default values

- Both axes linear
 - MATLAB chooses axis ranges so that end values are nice
- Points connected by straight lines
- No point markers
- Points and lines in blue

To use values other than defaults,



Line specifiers define style and color of lines, and marker types

Line Styles

Line Style	Specifier
solid (default)	-
dashed	--

Line Style	Specifier
dotted	:
dash-dot	-.

Line Colors

Line Color	Specifier
red	r
green	g
blue	b
cyan	c

Line Color	Specifier
magenta	m
yellow	y
black	k
white	w

Marker Types

Marker Type	Specifier		Marker Type	Specifier
plus sign	+		square	s
circle	o		diamond	d
asterisk	*		five-pointed star	p
point	.		six-pointed star	h
cross	x		triangle (pointed left)	<
triangle (pointed up)	^		triangle (pointed right)	>
triangle (pointed down)	v			

Property Name and Property Value:

- In `plot` command, type property name in quote marks, then comma, then value

Property Name	Description	Possible Property Values
<code>LineWidth</code> (or <code>linewidth</code>)	Specifies the width of the line.	A number in units of points (default 0.5).
<code>MarkerSize</code> (or <code>markersize</code>)	Specifies the size of the marker.	A number in units of points.
<code>MarkerEdgeColor</code> (or <code>markeredgecolor</code>)	Specifies the color of the marker, or the color of the edge line for filled markers.	Color specifiers from the table above, typed as a string.
<code>MarkerFaceColor</code> (or <code>markerfacecolor</code>)	Specifies the color of the filling for filled markers.	Color specifiers from the table above, typed as a string.

For example, the command:

```
plot(x,y, '-mo', 'LineWidth', 2, 'markersize', 12,
      'MarkerEdgeColor', 'g', 'markerfacecolor', 'y')
```

creates a plot that connects the points with a magenta solid line and circles as markers at the points. The line width is two points and the size of the circle markers is 12 points. The markers have a green edge line and yellow filling.

Plot two or more graphs on same plot as follows (example for three graphs)

```
plot(x, y, u, v, t, h)
```

- Plots y vs. x , v vs. u , h vs. t
- Vectors of each pair must be same size
 - Can be different than sizes in other pairs
- Can use line specifiers by putting in triplets (x-data, y-data, specifier), e.g.,

```
plot(x, y, '-b', u, v, '--r', 't, h, 'g:')
```

Normally, each time you execute `plot` it erases previous plot and draws new one. To change this behavior:

- Draw the first graph with `plot`
- Issue the command `hold on`
- Call `plot` for each of the remaining graphs
- Issue the command `hold off`

Graphs drawn after `hold on` are added to plot. Graphs drawn after `hold off` erase plot

`line` command adds additional graphs to an existing plot

```
line(x,y,'PropertyName','PropertyValue')
```

Example

```
line(x,y,'linestyle','--','color','r','marker','o')
```

adds graph drawn with dashed red line and circular markers to current plot

`plot` makes basic plot. After issuing that command, can use

- `xlabel('some text')` writes label below horizontal axis
 - Example: `xlabel('Time (sec)')`
- `ylabel('some text')` writes label to left of vertical axis
 - Example: `ylabel('Current (mA)')`

- `title('Some text')` writes title above plot
 - Example: `title('Diode Current')`
- `text(x, y, 'Some text')` places text in figure with first character at (x, y)
 - Example:
`text(x, y, 'Peak 3.5 sec after first')`
- `gtext('Some text')` – figure window opens, user clicks on graph where she wants text to appear

`legend('text1','text2',...,pos)`
writes legend

- For each graph (data set) displays short line in same style as graph line and adds specified text
 - First string goes with first graph plotted, second string goes with second graph plotted, etc.
- Most useful for plots having at least two graphs
- `pos` values in book are obsolete as of MATLAB 7.0 (R14). Type `help legend` or see documentation for new values

Formatting the text within the `xlabel`, `ylabel`, `title`, `text` **and** `legend` **commands:**

Can format text displayed by above commands

- Can set font, size, character color, background color, sub/superscript, style (bold, italic, etc.)
- Can display Greek letters
- Can format using modifiers within text string or by adding property names and values to command

Text modifiers are placed inside text string and affect appearance of text

- All text following modifier gets modified
- To only modify some text put open brace (`{`), modifier, text-to-be-modified, close brace (`}`)

Example titles

```
title('\it What You Should Never See')
```

makes

What You Should Never See

```
title('What You Should{\it Never} See')
```

makes

What You Should *Never* See

Some common modifiers

- `\bf` – **bold face**
- `\it` – *italic*
- `\rm` – normal font
- `\fontname{fontname}` – font name
- `\fontsize{fontsize}` – font size

Subscript and superscript:

To make single character

- Subscript – precede it by underscore (`_`)
- Superscript – precede it by caret (`^`)

For multiple characters, same as above but enclose characters in (curly) braces

- `xlabel('H_2O')` makes H₂O
- `ylabel('e^{ -k*sin(x) }')` makes $e^{-k\sin(x)}$

Greek characters:

To make a Greek letter, follow
backslash with letter name
(in English!)

- Name in lowercase makes
lowercase Greek letter
- Name with capitalized first letter
makes uppercase Greek letter

`ylabel('Standard deviation (\sigma) of resistance in M\Omega')`

makes

Standard deviation (σ) of resistance in $M\Omega$

Some Greek characters

Characters in the string	Greek Letter
<code>\alpha</code>	α
<code>\beta</code>	β
<code>\gamma</code>	γ
<code>\theta</code>	θ
<code>\pi</code>	π
<code>\sigma</code>	σ

Characters in the string	Greek Letter
<code>\Phi</code>	Φ
<code>\Delta</code>	Δ
<code>\Gamma</code>	Γ
<code>\Lambda</code>	Λ
<code>\Omega</code>	Ω
<code>\Sigma</code>	Σ

For `xlabel`, `ylabel`, `title`, `text`, can also change display of entire text string by using property name – property value pairs, e.g.,

```
text(x,y,'Some text',PropertyName,PropertyValue)
```

- `PropertyName` is text string
- `PropertyValue` is number if value is number or text string if value is letter or word

Example

```
text(x,y,'Depth','Rotation',45)
```

 makes

Depth

Some property-name property-value pairs

Property Name	Description	Possible Property Values
Rotation	Specifies the orientation of the text.	Scalar (degrees) Default: 0
FontAngle	Specifies italic or normal style characters.	normal, italic Default: normal
FontName	Specifies the font for the text.	Font name that is available in the system.
FontSize	Specifies the size of the font.	Scalar (points) Default: 10
FontWeight	Specifies the weight of the characters.	light, normal, bold Default: normal
Color	Specifies the color of the text.	Color specifiers (See Section 5.1).
Background-Color	Specifies the background color (rectangular area).	Color specifiers (See Section 5.1).
EdgeColor	Specifies the color of the edge of a rectangular box around the text.	Color specifiers (See Section 5.1). Default: none.
LineWidth	Specifies the width of the edge of a rectangular box around the text.	Scalar (points) Default: 0.5

The `axis` command:

MATLAB makes axes limits in `plot` command so that all data appears and limits are nice numbers. Can change that with `axis` command

Common axis variations are:

```
axis([xmin xmax ymin ymax])
```

- Sets limits of both axes

```
axis equal
```

- Sets same scale for both axes

```
axis square
```

- Sets axis region to be square

```
axis tight
```

- Sets axes limits to range of data
(not usually nice numbers!)

The `grid` command:

`grid on`

- Adds grid lines to plot

`grid off`

- Removes grid lines from plot

A *histogram* is a plot of the distribution of data. Entire range of data broken into consecutive subranges or *bins*.

In histogram plot

- Each bin represented by vertical bar
- Left and right of vertical bar show range of data in bin
- Height of vertical bar is number of data points in bin

MATLAB command `hist` makes histogram. Simplest form is `hist(y)`

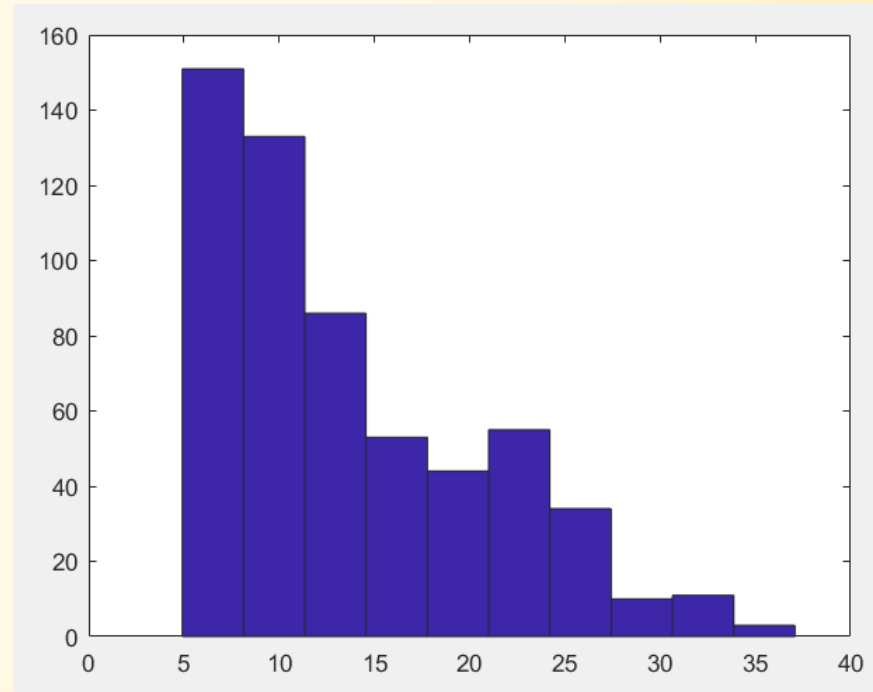
- `y` is vector of data points
- `hist` divides range into ten equal bins, then plots result

`figure(14)`

`hist(gust);`

`[N,XO] = hist(gust);`

See next slide for the
Meaning of `N`, `XO`



Can get histogram heights and center of bins if desired

```
n=hist(y)  n=hist(y,nbins)  n=hist(y,x)
```

- Output `n` is a vector
 - Size of `n` is number of bins
 - Value of element of `n` is number of data points in corresponding bin

```
[n xout]=hist(y)
```

- `n` same as before
- `xout(i)` is center of *i*th bin

Additional forms of `hist`

`hist(y, nbins)`

- MATLAB divides range into `nbins` (scalar) bins of equal size

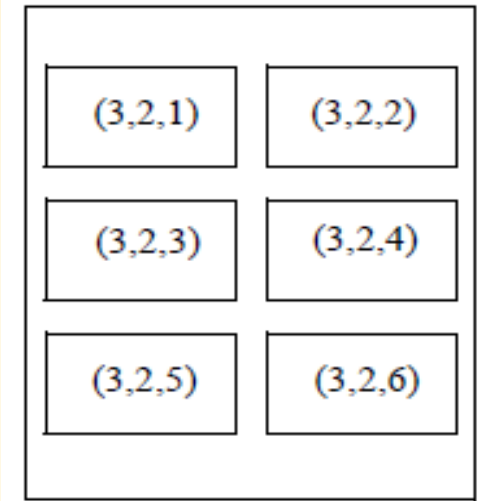
`hist(y, x)`

- `x` (vector) specifies midpoint of each bin
 - Spacing between consecutive elements of `x` can be different
 - Bin edges are midpoints of consecutive bins

`subplot(m,n,p)`

divides Figure Window into m rows and n columns of subplots

- Subplots numbered from left to right and top to bottom, with upper left being subplot 1 and lower right subplot $m \times n$. p in subplot command refers to this numbering



Subplot numbers for
3x2 set of subplots

```
subplot (m,n,p)
```

m- # rows, n- # columns, p-pos

- If subplots don't exist, `subplot` creates them and makes subplot `p` the current subplot
- If subplots exist, `subplot` makes subplot `p` the current one
- When `subplot` defines current subplot, next `plot` and formatting commands draw in current subplot

The logic of programming

- Logic- use the correct reasoning to solve problems
- Flow charts- describing what you want to do
 - What do you do first, next, last
- Are there possible choices depending on the situation that require decisions?
 - Are they relational ($>$, $=$, $<$) or logical (and/or)?
 - Do you need to evaluate the decision many times?

A *flowchart* is a diagram that shows the code flow. It is particularly useful for showing how conditional statements work. Some common flowchart symbols are



- represents a sequence of commands



- represents an if-statement



- shows the direction of code execution

Relational operators:

<u>Relational operator</u>	<u>Description</u>
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
~=	Not Equal to

- Can't put space between operators that have two characters
- "Not equal to" is "~= ", not "!=" as in C or C++
- "Equal to" comparison is two equal signs (==), not one.
 - Remember, "=" means "assign to" or "put into"
- Result of comparing with a relational operator is always "true" or "false"
 - If "true", MATLAB gives the comparison a value of one (1)
 - If "false", MATLAB gives the comparison a value of zero (0)

When comparing array to scalar

- MATLAB compares scalar to every member of array
- Result is an array that has same dimensions as original but only contains 1's and 0's

When comparing arrays

- They must be the same dimensions
- MATLAB does an elementwise comparison
- Result is an array that has same dimensions as other two but only contains 1's and 0's

A logical vector or logical array is a vector/array that has only logical 1's and 0's

- 1's and 0's from mathematical operations don't count
- 1's and 0's from relational comparisons do work
- First time a logical vector/array used in arithmetic, MATLAB changes it to a numerical vector/array

Can use logical vector to get actual values that satisfy relation, not just whether or not relation satisfied.

Doing this is called *logical indexing* or *logical subscripting*

- Do this by using logical vector as index in vector of values. Result is values that satisfy relation, i.e., values for which relationship are 1

Logical operators:

Boolean logic is a system for combining expressions that are either true or false.

- MATLAB has operators and commands to do many Boolean operations
- Boolean operations in combination with relational commands let you perform certain types of computations clearly and efficiently

A *truth table* defines the laws of Boolean logic. It gives the output of a logical operation for every possible combination of inputs. The truth table relevant to MATLAB is

INPUT		OUTPUT				
A	B	AND A&B	OR A B	XOR (A,B)	NOT ~A	NOT ~B
false	false	false	false	false	true	true
false	true	false	true	true	true	false
true	false	false	true	true	false	true
true	true	true	true	false	false	false

In words, the truth table says

- AND is true if both inputs are true, otherwise it is false
- OR is true if at least one input is true, otherwise it is false
- XOR (exclusive OR) is true if exactly one input is true, otherwise it is false
- NOT is true if the input is false, otherwise it is false

An arithmetic operator, e.g., $+$ or $-$, is a symbol that causes MATLAB to perform an arithmetical operation using the numbers or expressions on either side of the symbol

Similarly, a *logical operator* is a character that makes MATLAB perform a logical operation on one or two numbers or expressions

MATLAB has three logical operators: $\&$, $|$, \sim

- $a \& b$ does the logical AND operation on a and b
- $a | b$ does the logical OR operation on a or b
- $\sim a$ does the logical NOT operation on a
- Arguments to all logical operators are numbers
 - Zero is "false"
 - Any non-zero number is "true"
- Result (output) of logical operator is a logical one (true) or zero (false)

When operating with array and scalar

- MATLAB does element-wise operation on each array element with scalar
- Result is an array that has same dimensions as original but only contains 1's and 0's

When using logical operator on arrays

- They must be the same dimensions
- MATLAB does an element-wise evaluation of operator
- Result is an array that has same dimensions as other two but only contains 1's and 0's

(not only operates on one array so the first point is irrelevant)

Can combine arithmetic, relational operators, and logical operators.
Order of precedence is

<u>Precedence</u>	<u>Operation</u>
1 (highest)	Parentheses (if nested parentheses exist, inner ones have precedence)
2	Exponentiation
3	Logical NOT (~)
4	Multiplication, division
5	Addition, subtraction
6	Relational operators (>, <, >=, <=, ==, ~=)
7	Logical AND (&)
8 (lowest)	Logical OR ()

Built-in logical functions:

MATLAB has some built-in functions or commands for doing logical operations and related calculations. Three are equivalent to the logical operators

- `and (A, B)` – same as `A & B`
- `or (A, B)` – same as `A | B`
- `not (A)` – same as `~A`

MATLAB also has other Boolean functions

Function	Description	Example
<code>xor(a,b)</code>	Exclusive or. Returns true (1) if one operand is true and the other is false.	<pre>>> xor(7,0) ans = 1 >> xor(7,-5) ans = 0</pre>
<code>all(A)</code>	Returns 1 (true) if all elements in a vector A are true (nonzero). Returns 0 (false) if one or more elements are false (zero). If A is a matrix, treats columns of A as vectors, and returns a vector with 1s and 0s.	<pre>>> A=[6 2 15 9 7 11]; >> all(A) ans = 1 >> B=[6 2 15 9 0 11]; >> all(B) ans = 0</pre>
<code>any(A)</code>	Returns 1 (true) if any element in a vector A is true (nonzero). Returns 0 (false) if all elements are false (zero). If A is a matrix, treats columns of A as vectors, and returns a vector with 1s and 0s.	<pre>>> A=[6 0 15 0 0 11]; >> any(A) ans = 1 >> B = [0 0 0 0 0 0]; >> any(B) ans = 0</pre>
<code>find(A)</code> <code>find(A>d)</code>	If A is a vector, returns the indices of the nonzero elements. If A is a vector, returns the address of the elements that are larger than d (any relational operator can be used).	<pre>>> A=[0 9 4 3 7 0 0 1 8]; >> find(A) ans = 2 3 4 5 8 9 >> find(A>4) ans = 2 5 9</pre>

find- powerful tool

- $I = \text{find}(X)$ returns a vector containing the linear indices of each nonzero element in array X
- Very helpful for handling logical checks
- *in Python, is 'where' the functional equivalent of find?*

Cell array

- matlab arrays are a fixed size $m \times n$ of one type (string, float, etc.)
- How to handle mixture of arrays with different sizes and types: cell array
- Access the content of cell arrays using the cell index be accessed with `{ }` brackets, e.g., `C{1,2}`
- Access an element within the array embedded with a cell array by `C{1,2}(10,1)`
- Convert one cell array e.g., `C{1,2}` to matrix by using `cell2mat`

Cell array

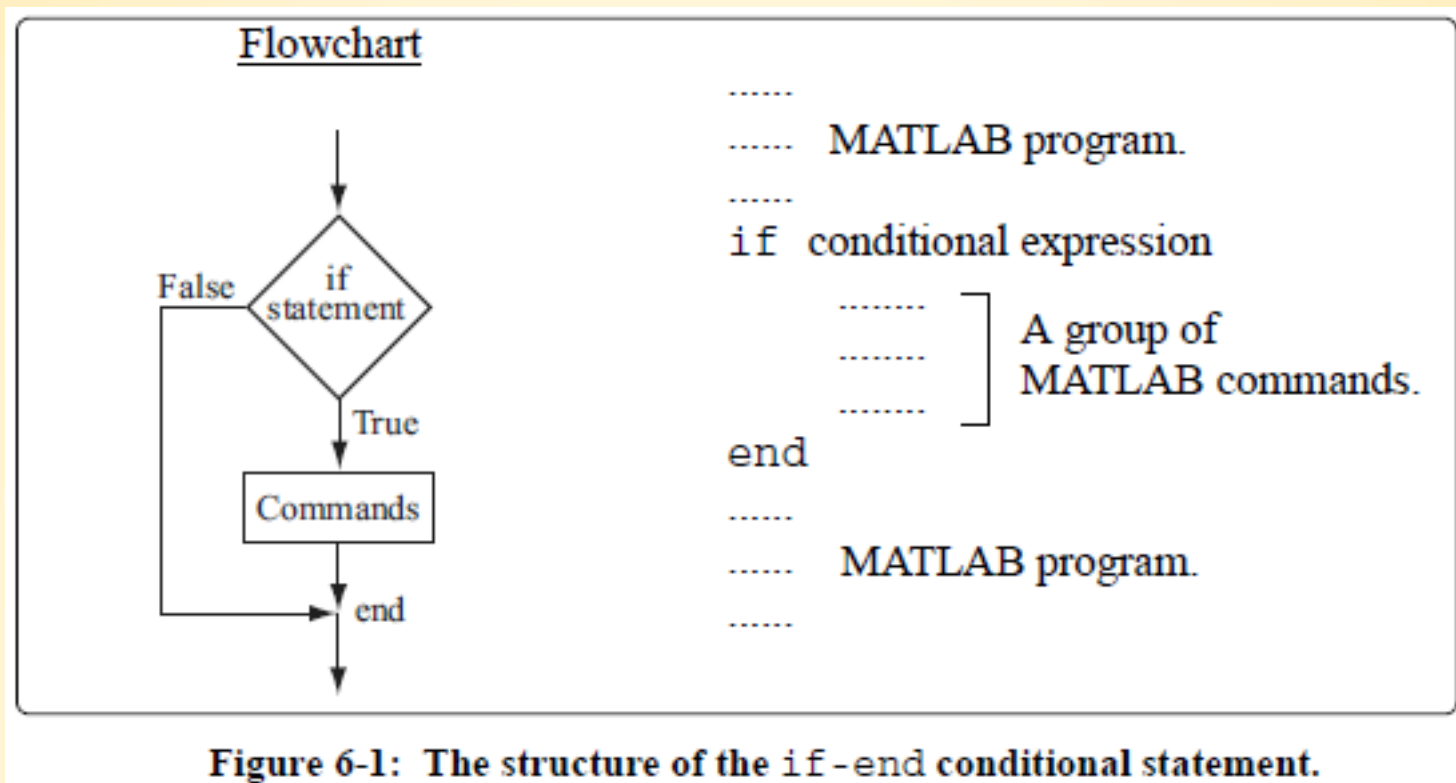
- matlab arrays are a fixed size $m \times n$ of one type (string, float, etc.)
- How to handle mixture of arrays with different sizes and types: cell array
- Access the content of cell arrays using the cell index be accessed with { } brackets, e.g., `C{1,2}`
- Access an element within the array embedded with a cell array by `C{1,2}(10,1)`

DateNumber = datenum(t) converts the datetime values in datetime array **t** to serial date numbers

- Critical to handle time properly
- %convert time to numerical time
- `time = datenum(M{1,2},'mm/dd/yyyy HH:MM');`
- %convert numerical time to vector
- `time_vec = datevec(time);`

A conditional statement is a command that allows MATLAB to decide whether or not to execute some code that follows the statement

- Conditional statements almost always part of scripts or functions
- They have three general forms
 - `if-end`
 - `if-else-end`
 - `if-elseif-else-end`



If the conditional expression is true, MATLAB runs the lines of code that are between the line with `if` and the line with `end`. Then it continues with the code after the `end`-line

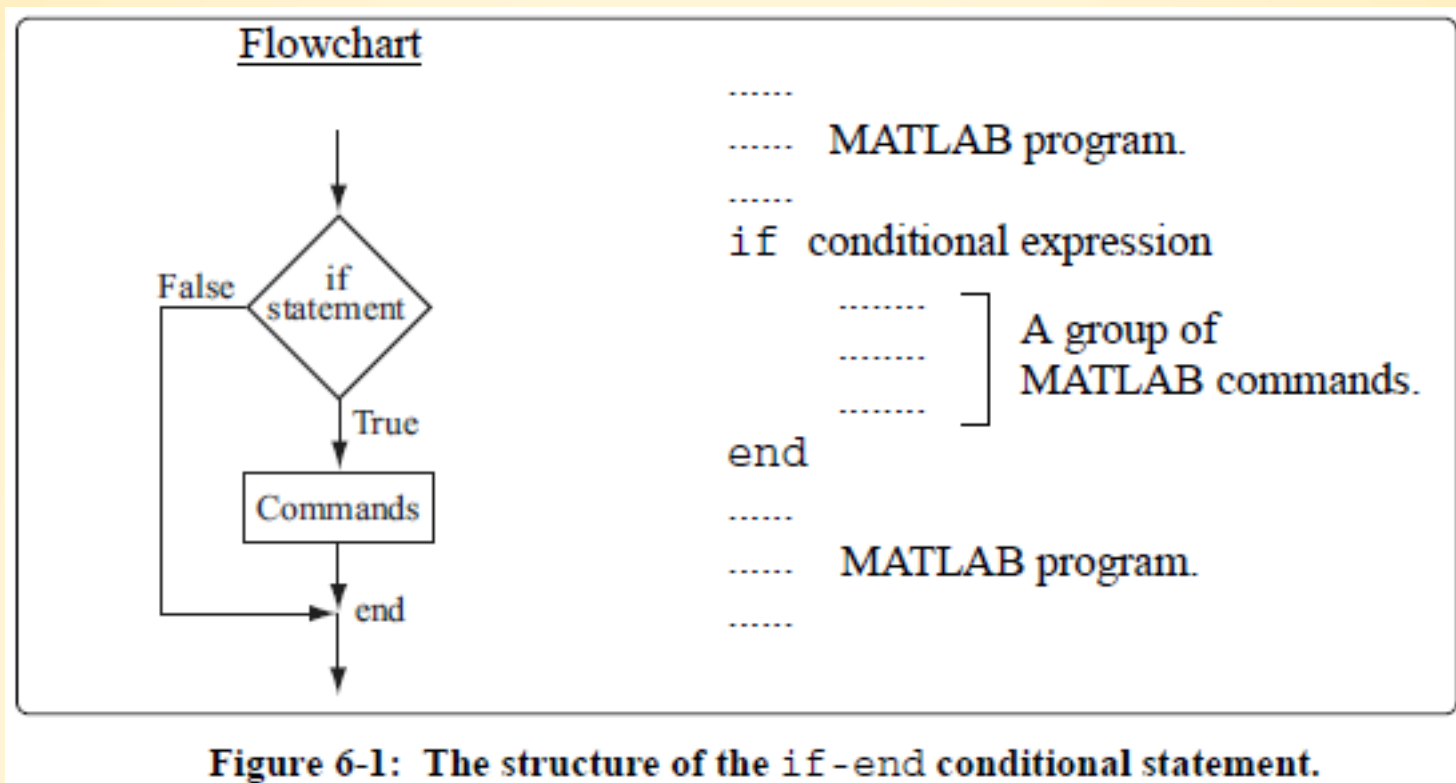


Figure 6-1: The structure of the `if-end` conditional statement.

If the conditional expression is false, MATLAB skips the lines of code that are between the line with `if` and the line with `end`. Then it continues with the code after the `end`-line

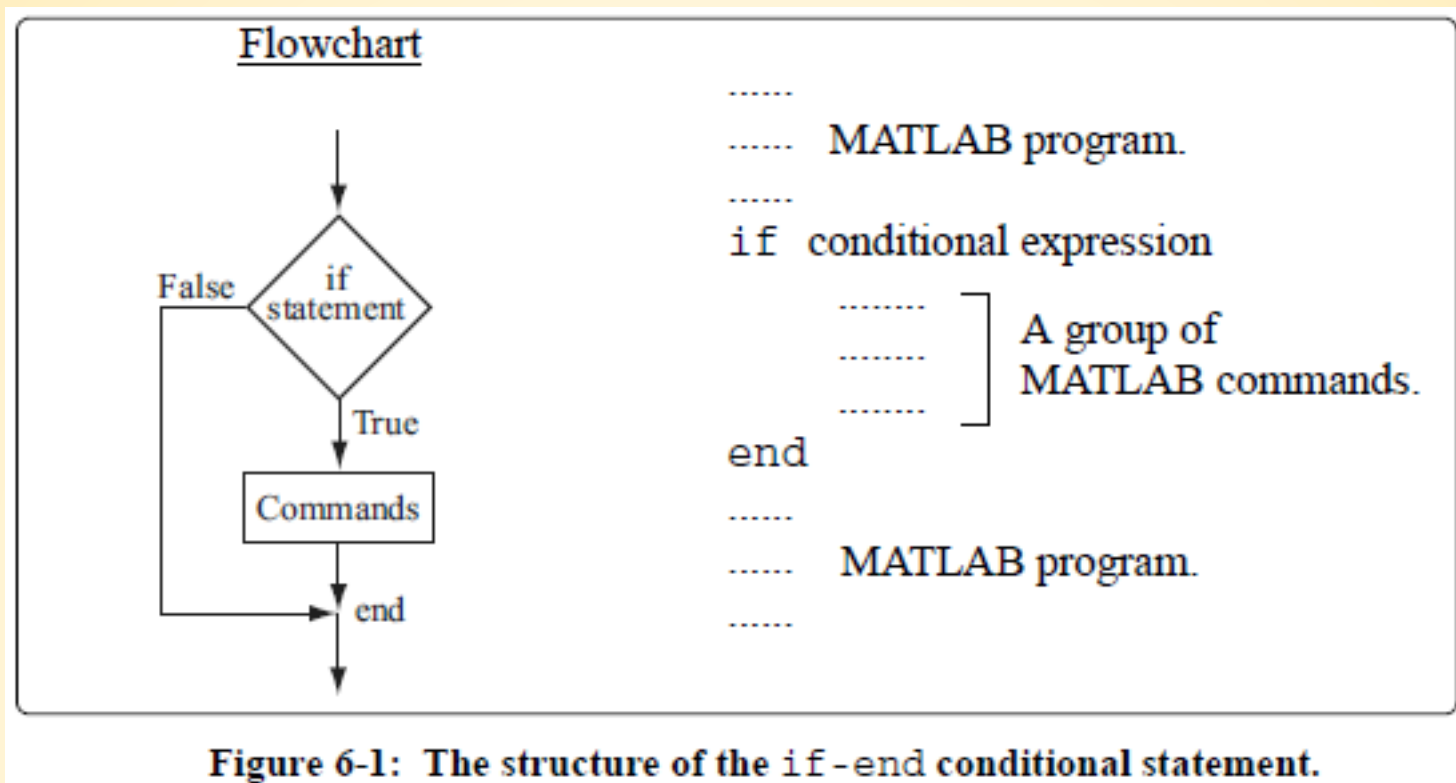


Figure 6-1: The structure of the if-end conditional statement.

The conditional expression is true if it evaluates to a logical 1 or to a non-zero number. The conditional expression is false if it evaluates to a logical 0 or to a numerical zero

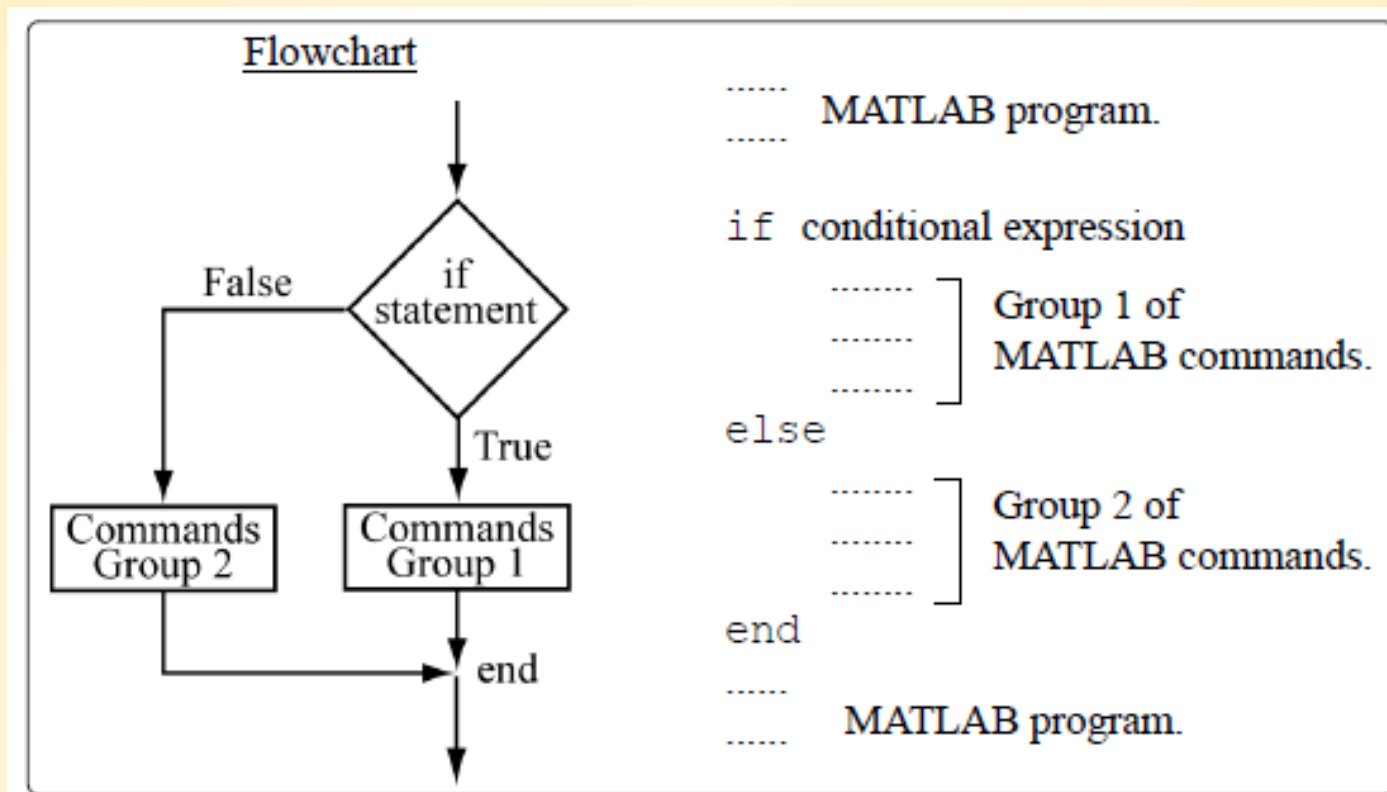


Figure 6-2: The structure of the if-else-end conditional statement.

Fig. 6-2 shows the code and the flowchart for the if-else-end structure

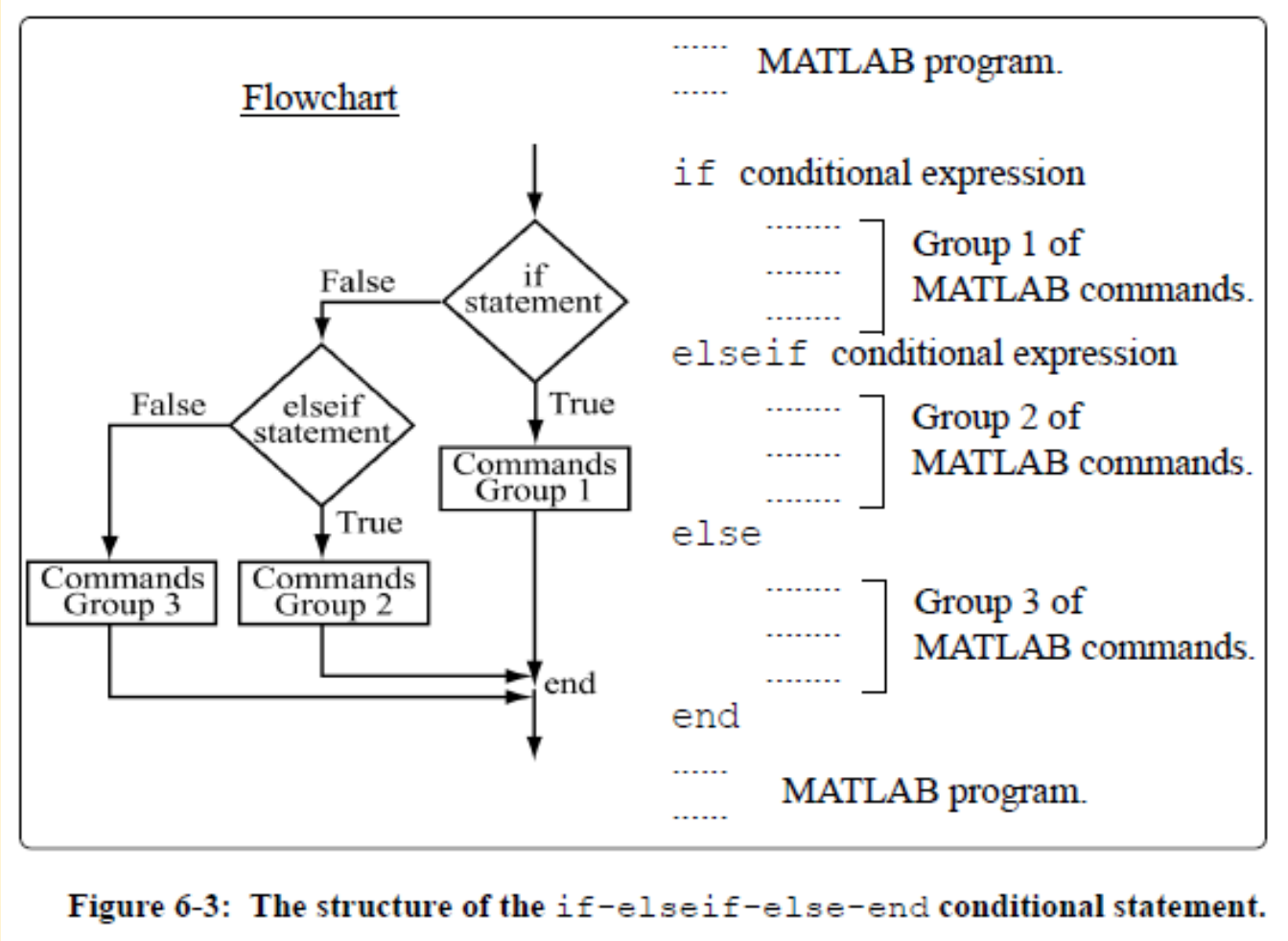


Fig. 6-3 shows the code and the flowchart for the if-elseif-else-end structure

`if-elseif-else-end` structure gets hard to read if more than a few `elseif` statements. A clearer alternative is the `switch-case` structure

- `switch-case` slightly different because choose code to execute based on value of scalar or string, not just true/false

A for-loop executes set of commands a specified number of times. The set of commands is called the *body* of the loop

MATLAB has two ways to control number of times loop executes commands

- Method 1 – loop executes commands a specified number of times
- Method 2 – loop executes commands as long as a specified expression is true

Can often evaluate using either a for-loop or elementwise operations.

Elementwise operations are:

- faster
- easier to read
- ADVICE – try to use elementwise operations; use for-loops when elementwise operations become too abstract

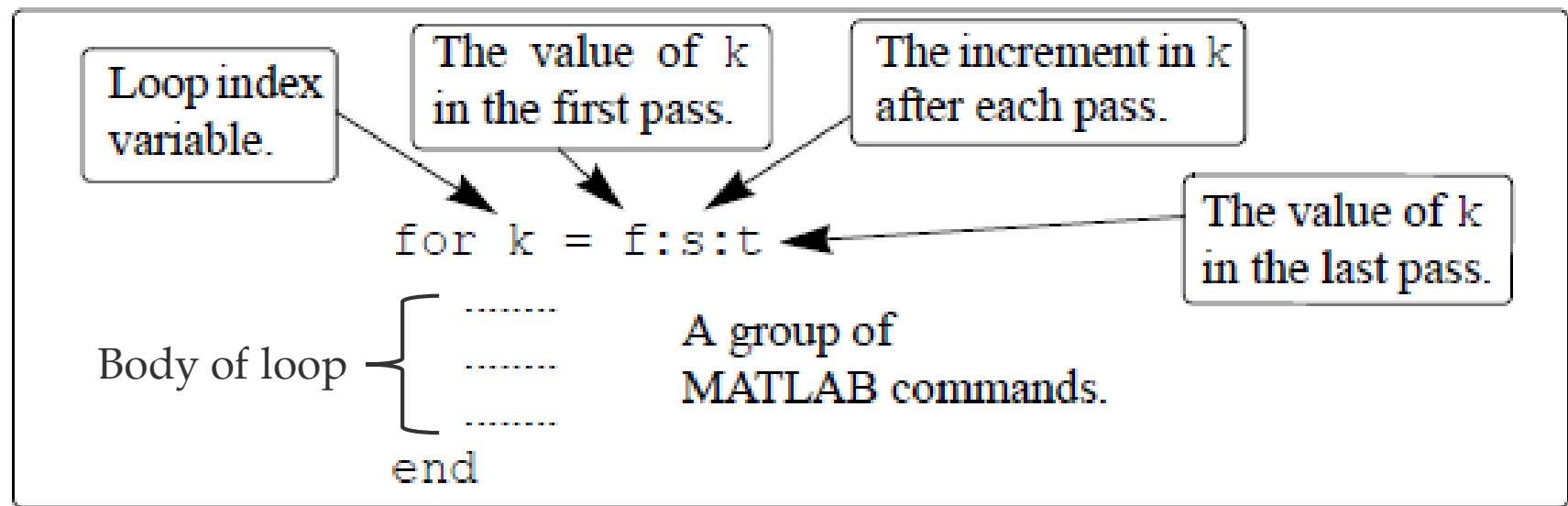


Figure 6-5: The structure of a for-end loop.

- The loop index variable can have any variable name (usually `i`, `j`, `k`, `m`, and `n` are used)
 - `i` and `j` should not be used when working with complex numbers. (`ii` and `jj` are good alternative names)

1. Loop sets k to f , and executes commands between `for` and the `end` commands, i.e., executes body of loop
2. Loop sets k to $f+s$, executes body
3. Process repeats itself until $k > t$
4. Program then continues with commands that follow `end` command
 - f and t are usually integers
 - s usually omitted. If so, loop uses increment of 1

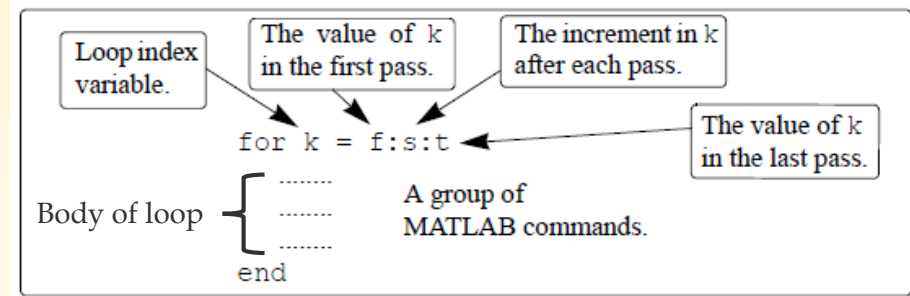
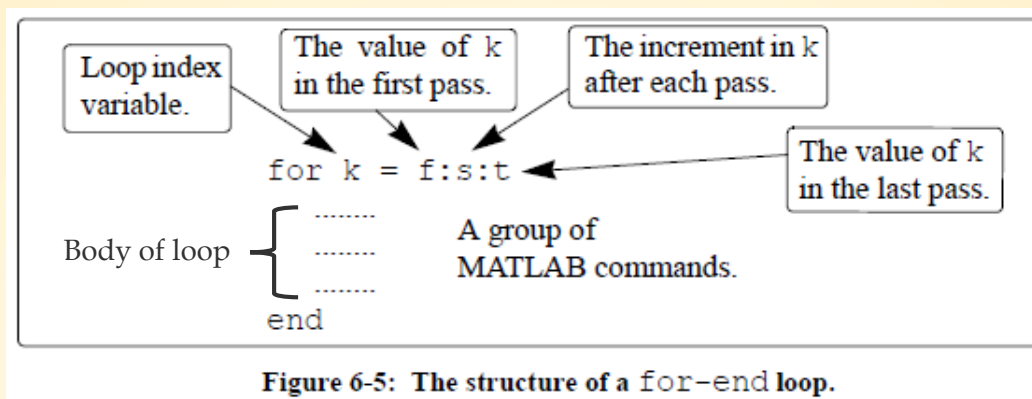


Figure 6-5: The structure of a for-end loop.



- Increment `s` can be negative
 - For example, `k = 25:-5:10` produces four passes with `k = 25, 20, 15, 10`
- If `f = t`, loop executes once
- If `f > t` and `s > 0`, or if `f < t` and `s < 0`, loop not executed

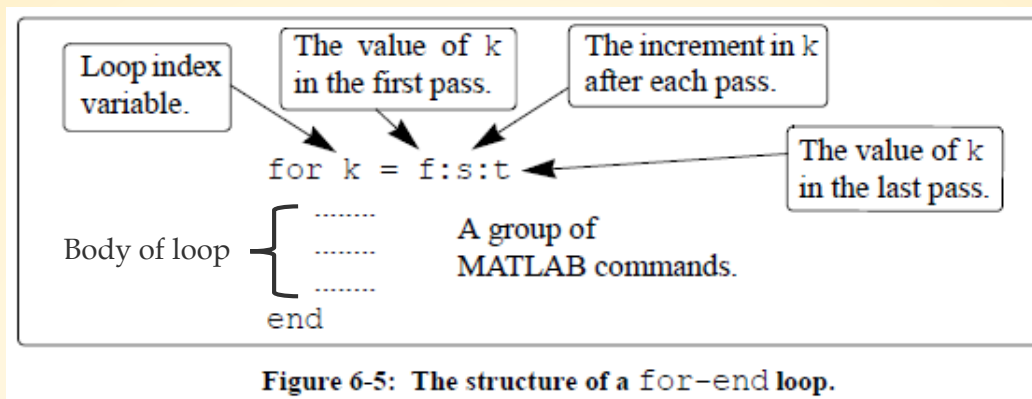
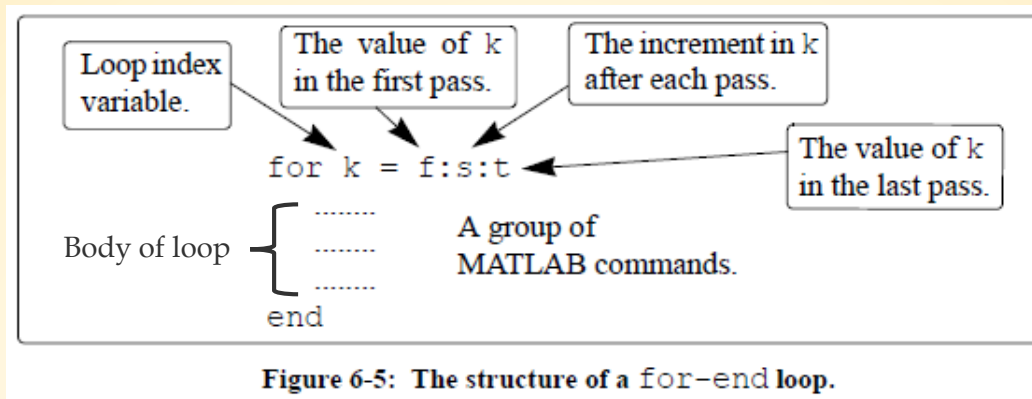
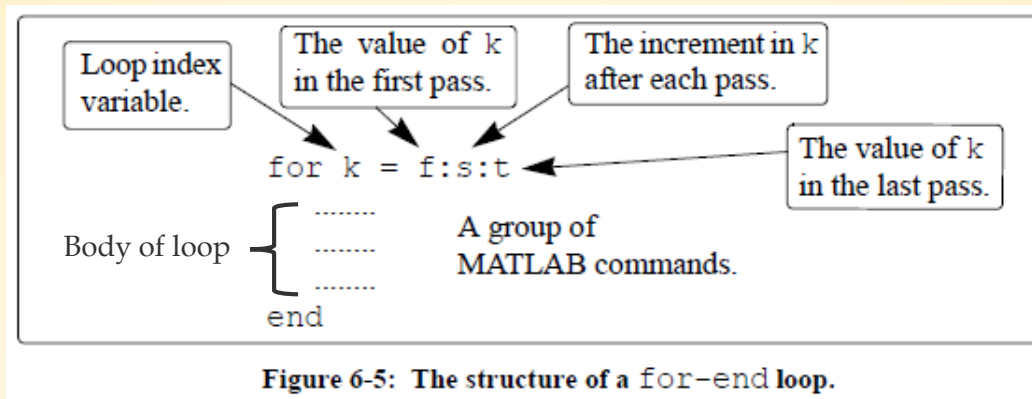


Figure 6-5: The structure of a for-end loop.

- If values of k , s , and t are such that k cannot be equal to t , then
 - If s positive, last pass is one where k has largest value smaller than t
 - For example, $k = 8:10:50$ produces five passes with $k = 8, 18, 28, 38, 48$
 - If s is negative, last pass is one where k has smallest value larger than t



- In the `for` command `k` can also be assigned specific value (typed in as a vector)
 - For example: `for k = [7 9 -1 3 3 5]`
- In general, loop body should not change value of `k`
- Each `for` command in a program must have an `end` command



- Value of loop index variable (k) not displayed automatically
 - Can display value in each pass (sometimes useful for debugging) by typing k as one of commands in loop
- When loop ends, loop index variable (k) has value last assigned to it

while-end loop used when

- You don't know number of loop iterations
- You do have a condition that you can test and stop looping when it is false. For example,
 - Keep reading data from a file until you reach the end of the file
 - Keep adding terms to a sum until the difference of the last two terms is less than a certain amount

1. Loop evaluates conditional-expression

2. If conditional-expression is true, executes code in body, then goes back to Step 1

3. If conditional-expression is false, skips code in body and goes to code after end-statement

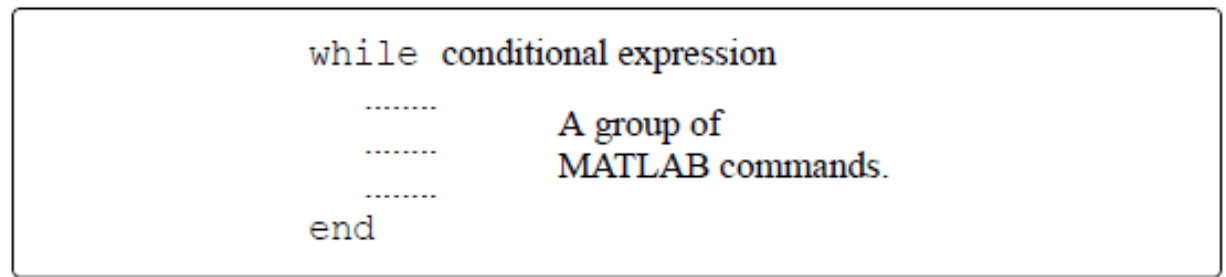


Figure 6-6: The structure of a while-end loop.

The conditional expression of a while-end loop

- Has a variable in it
 - Body of loop must change value of variable
 - There must be some value of the variable that makes the conditional expression be false

- If the conditional expression never becomes false, the loop will keep executing... Forever as an *infinite loop*. Your program will just keep running, and if there is no output from the loop (as is often the case), it will look like MATLAB has stopped responding
- If your program gets caught in an indefinite loop,
 - Put the cursor in the Command Window
 - Press CTRL+C

Common causes of infinite loops:

- No variable in conditional expression

```
distance1 = 1;
```

```
distance2 = 10;
```

```
distance3 = 0;
```

```
while distance1 < distance2
```

distance1 and distance2
never change

```
    fprintf('Distance = %d\n', distance3);
```

```
end
```

Common causes of infinite loops:

- Variable in conditional expression never changes

```
minDistance = 42;
```

```
distanceIncrement = 0; ← Typo - should be 10
```

```
distance = 0;
```

```
while distance < minDistance
```

```
    distance=distance+distanceIncrement;
```

```
end
```

Common causes of infinite loops:

- Wrong variable in conditional expression changed

```
minDistance = 42;
```

```
delta = 10;
```

```
distance = 0;
```

```
while distance < minDistance
```

```
    minDistance = minDistance + delta;
```

```
end
```



Typo - should be distance

If a loop or conditional statement is placed inside another loop or conditional statement, the former are said to be *nested* in the latter.

- Most common to hear of a *nested loop*, i.e., a loop within a loop
 - Often occur when working with two-dimensional problems
- Each loop and conditional statement must have an end statement

The `break` command:

- When inside a loop (`for` and `while`), `break` terminates execution of loop
 - MATLAB jumps from `break` to end command of loop, then continues with next command (does not go back to the `for` or `while` command of that loop).
 - `break` ends whole loop, not just last pass
- If `break` inside nested loop, only nested loop terminated (not any outer loops)
 - `break` command in script or function file but not in a loop terminates execution of file
 - `break` command usually used within a conditional statement.
 - In loops provides way to end looping if some condition is met

The continue command:

Use `continue` inside a loop (for- and while-) to stop current iteration and start next iteration

- `continue` usually part of a conditional statement. When MATLAB reaches `continue` it does not execute remaining commands in loop but skips to the `end` command of loop and then starts a new iteration

EXAMPLE

```
for ii=1:100
```

```
    if rem( ii, 8 ) == 0
```

```
        count = 0;
```

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

```
        continue;
```

```
    end
```

```
    % code
```

```
    % more code
```

```
end
```

Every eight iteration
reset count to zero,
print the iteration
number, and skip the
remaining computations
in the loop

Summary

- There are many ways to muck up “for,while” loops and if-else logic.
- Try to use elementwise calculations whenever possible
- Check that you are getting the correct answers