MATLAB Basics and More

Appendix A

Scalars, Vectors, and Matrices

- MATLAB was originally developed to be a *matrix laboratory*
- In its present form it has been greatly expanded to include many other features, but still maintains inherent matrix and vector based arithmetic at its core
- The basic *data structure* of MATLAB is the matrix

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}, B = \begin{bmatrix} b_{11} & b_{12} \end{bmatrix}$$

$$C = \begin{bmatrix} c_{11} \\ c_{21} \end{bmatrix}, D = \begin{bmatrix} d_{11} \end{bmatrix}$$

  – A matrix with a single row is also known as a *row vector*

  – A matrix with a single column is also known as a *column vector*

  – A matrix with just one row and one column (a single element) is simply a scalar

Variable Initialization

- Variable names in MATLAB
  - Must be begin with a letter
  - May contain digits and the underscore character
  - May be any length, but must be unique in the first 19 characters
- A common way to create a matrix instance is by simply assigning a list of numbers to it, e.g.

```
A = [3.1415]; % A 1x1 matrix
B = [23, 35.3]; % A 1x2 matrix
C = [1, 2, 3; 4, 5, 6; 7, 8, 9]; % a 3x3 matrix
```

  - Note: a comma or a blank is used to separate elements in the same row, while a semicolon is used to separate rows
  - The *assignment statements* listed above are terminated with a semicolon to suppress MATLAB from echoing the variables value
- To continue one line into another we can break a line where a comma occurs, and then follow the comma with an ellipsis (three periods in a row), e.g.,

```
M = [.1, .2, .3, .4, .5, .6, .7, .8, .9, 1.0];
% or break the line
M = [.1, .2, .3, .4, .5, ...
.6, .7, .8, .9, 1.0];
```

- One matrix may be used to define another matrix, e.g.,

```
A = [4, 5, 6];
B = [1, 2, 3, A]; % creates
B = [1, 2, 3 ,4 ,5 ,6];
```

The Colon Operator

- To make matrix generation and addressing easier we use the colon operator
- The colon operator is indeed powerful, and mastering it is essential to becoming a MATLAB expert
  - Here we use it to generate row vectors

```
k = 0:6; % creates
k = [0, 1, 2, 3, 4, 5, 6];
t = 0:.25:2; % creates
t = [0, .25, .5, .75, 1, 1.25, 1.5, 1.75, 2];
s = -10:2:0; % creates
s = [-10, -8, -6, -4, -2, 0];
```

  - Here we use to generate **row and column slices** of a matrix

```
A = [1, 2, 3; 4, 5, 6; 7, 8, 9];
```

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

```
A_col2 = A(:,2); % Span all rows with the column
                 % index fixed at 2.
A_row1 = A(1,:); % With row index fixed at 1,
                 % span all columns.
A_11 = A(2,2);   % Produces the scalar A_11 = 5
```

- We can combine the techniques to extract a **submatrix** of *A*

```
A_sub = A(2:3,2:3) % Extract a sub matrix
                   % consisting of rows 2-3 and
                    % columns 2-3
```

$$A_{\text{sub}} = \begin{bmatrix} 5 & 6 \\ 8 & 9 \end{bmatrix}$$

- We can swap the rows and columns of a matrix using the **transpose operator**, e.g.,

```
      A = [1, 2, 3];
      A_transpose = A′; % produces
      A_transpose = [1; 2; 3];
```

- A simple formatting scheme to have MATLAB display the values of several equal length vectors side-by-side is the following (actual MATLAB command line interaction)

```
>> A = [0, 1, 2, 3, 4];
>> B = [0, 1, 4, 9, 16];
>> C = [0, 1, 8, 27, 64];
>> % Display in side-by-side columns:
>> [A' B' C'] % commas may be included but not needed
ans =
     0     0     0
     1     1     1
     2     4     8
     3     9    27
     4    16    64
```

Special Values and Matrices

- To make life easier, MATLAB comes with several predefined values and matrix generators
  - `pi` represents π in MATLAB floating point precision, e.g.,

```
    >> pi
        ans =    3.1416
```

  - `i, j` represents the value $\sqrt{-1}$
  - `Inf` is the MATLAB notation for infinity, i.e., 1/0
  - `Nan` is the MATLAB representation for *not-a-number*; often a result of a 0/0 operation
  - clock displays the current time, e.g.,

```
    >> clock
ans =
  1.0e+003 *
  1.9970    0.0080    0.0270    0.0230    0.0160   0.0508
```

  - `date` is the current date in a string format

```
>> date
ans = 23-Aug-2006
```

  - `eps` is the smallest floating-point number by which two numbers can differ, e.g.,

```
>> eps
ans =  2.2204e-016
```

- A matrix of zeros can be generated with

```
A_3x3 = zeros(3,3); % or
A_3x3 = zeros(3);
B_3x2 = ones(3,2);
C_3x3 = ones(size(A_3x3));
```

$$A_{3x3} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, B_{3x2} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}$$

$$C_{3x3} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

- In linear algebra the **identity matrix** is often needed

```
I_3x3 = eye(3);
I_3x2 = eye(3,2);
```

$$I_{3x3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, I_{3x2} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}$$

User Prompting

- In the future we will write MATLAB programs or **script files** (saved as *.m files)
- Short of making a full graphical user interface (GUI) MATLAB program, which MATLAB can also do, we may simply wish to prompt the user to supply some input data
- When the following is encountered in a MATLAB program, the variable on the left is assigned the value the user enters at the MATLAB prompt

```
x = input(‘Please enter your height and weight: ’);
```

- We create a simple script file (more on this in the next chapter) `user_prompt`

```
% the *.m file user_prompt.m

x = input(‘Please enter your height and weight: ’);
```

- To run the script we type the file name at the MATLAB prompt (making sure the file is in the MATLAB path)

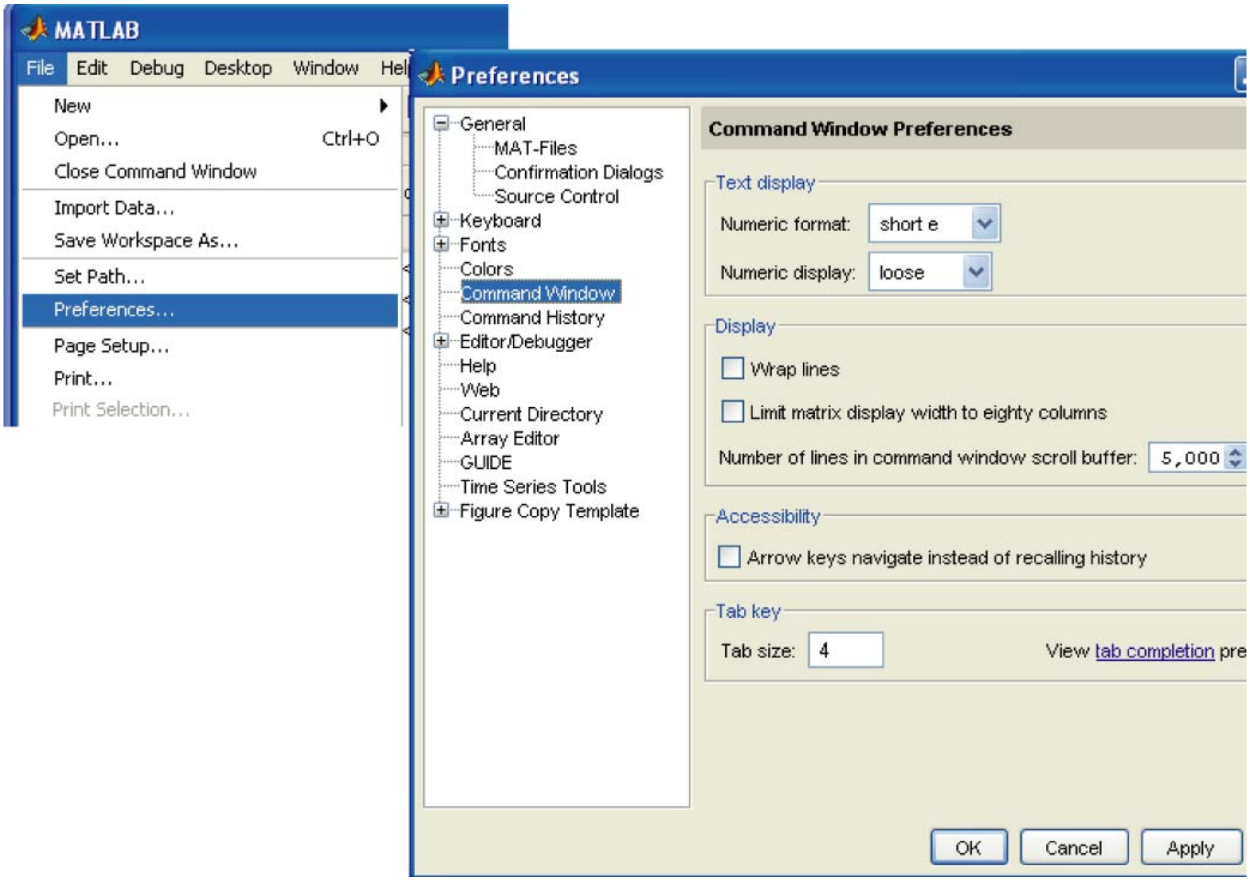
```
>> clear % clear all variables form the workspace
>> user_prompt % run the script file user_prompt.m
Please enter your height and weight: [100 200]
>> x
x =
   100   200
```

Output Display/Print Formatting

- Globally the display format for the command window can be changed by:
  - Typing commands directly, e.g.,

```
format short % four decimal digits
format long % 14 decimal digits
format short e % short with scientific notation
format long e % long with scientific notation
% others available
```

- Selecting displays formats from the command window pull down menus, e.g.,



Command Window Preferences

Text display

Numeric format: short e

Numeric display: loose

Display

☐ Wrap lines

☐ Limit matrix display width to eighty columns

Number of lines in command window scroll buffer: 5,000

Accessibility

☐ Arrow keys navigate instead of recalling history

Tab key

Tab size: 4

View tab completion pre

- Formatted output from script files is also possible, that is the format of variables printed to the screen can be made unique for each variable you want the program to print, and custom text be distributed around the numeric values

- The command `disp()` is used to display text and print the contents of a matrix

```
>> disp('A Text String to Display.')
A Text String to Display.
>> M = [1, 2; 3, 4];
>> disp(M)
     1     2
     3     4
```

- A fully customizable way of printing both text and matrices is the `fprintf()` command

  ```
  fprintf(format_string,matrices)
  ```

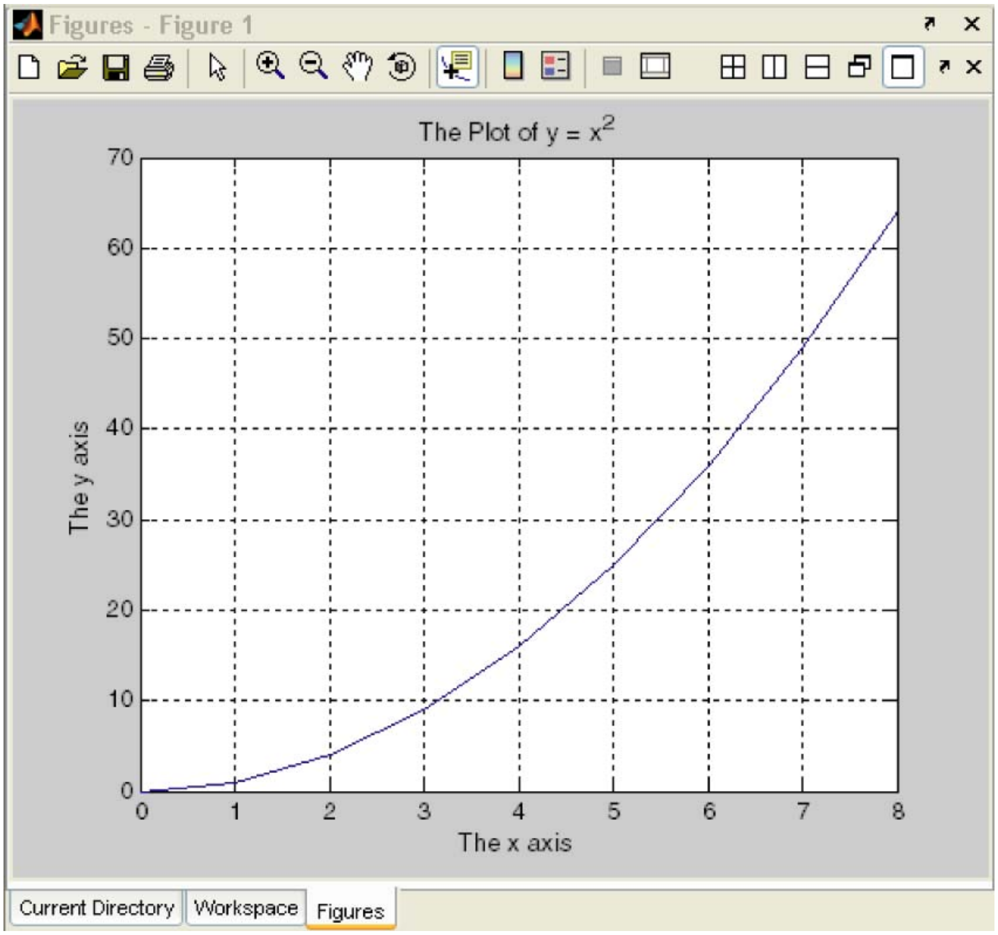
- The format string contains text you wish to display along with *format specifiers*, `%e`, `%f`, and `%g`, which tell `fprintf` how to display respective values in the matrices list
  - `%e`, `%f`, and `%g` select exponential, fixed point, or global (either `%f` or `%e` which ever is smaller)
  - `\n` tells MATLAB to start a new line

```
>> A = 24.567;
>> fprintf('Hello, we have A = %f and pi =%6.4f.\n',...
        A,pi)
Hello, we have A = 24.567000 and pi = 3.1416.
```

The x–y Plot Command

- A simple `y` versus `x` plot is obtained with the command `plot(x,y)`
- To address a specific figure window type `figure(1)`, etc.
- Plots without a title and axis labels are uninformative, so we must also add labels

```
>> x = [0, 1, 2, 3, 4, 5, 6 ,7 ,8];
>> y = [0, 1, 4, 9, 16, 25 ,36 ,49 ,64];
>> plot(x,y)
>> grid
>> title('The Plot of y = x^2')
>> ylabel('The y axis')
>> xlabel('The x axis')
```



Data File Commands

- Data file commands are used to `save` and `load` files in either standard ASCII text or the more compact MATLAB binary format which uses the `*.mat` extension
- The binary `.mat` format is useful for files which will be used with MATLAB, while the ASCII format is useful when working with other programs, or sharing data with others who may not have MATLAB

```
>> x = 0:5;
>> y = rand(size(x));
>> [x' y']
ans =
         0    0.3529
    1.0000    0.8132
    2.0000    0.0099
    3.0000    0.1389
    4.0000    0.2028
    5.0000    0.1987
>> save mat_data x y; % creates the file mat_data.mat
>> save mat_data.dat x y /ascii; % creates the text file
                                % mat_data.dat
```

- To verify that these files are valid we first clear the MATLAB work space using the command `clear`

```
>> clear
>> whos
>>
```

- Next we load the `.mat` file back into the work space

```
>> load mat_data
>> whos
  Name      Size         Bytes  Class
  x         1x6             48  double array
  y         1x6             48  double array
Grand total is 12 elements using 96 bytes
```

- To see how the ASCII file is loaded we again clear the work space

```
>> load mat_data.dat
>> whos
  Name           Size         Bytes  Class
  mat_data       2x6             96  double array
Grand total is 12 elements using 96 bytes
```

- When loading an ASCII file we simply place each row of the text file into a corresponding matrix row
- The variable name that holds the loaded text file is in this case `mat_data`
  - To recover the data vectors `x` and `y` from `mat_data` we must parse them out using the colon operator; left as an exercise

Scalar and Array Operations

Computations in MATLAB typically require wide variety of arithmetic computations between scalars, vectors, and matrices.

Scalar Operations

- Scalar operations are the most obvious if you have programmed in C, Fortran, Basic, etc.

Table A.1: Simple scalar operations

Operation	Algebraic Form	MATLAB syntax
addition	$a + b$	`a + b`
subtraction	$a - b$	`a - b`
multiplication	$a \times b$	`a*b`
division	$a \div b$	`a/b`
exponentiation	a^b	`a^b`

```
>> % Assign constants to a and b:
>> a = 3.1; b = 5.234;
>> c = a + b
c =    8.3340
>> c = a/b
c =    0.5923
>> c = a^b
c =  373.0672
```

Array Operations

- When working with vectors and matrices we have the choice of performing operations **element-by-element** or according to the rules of matrix algebra
- In this section we are only interested in element-by-element operations
- Basically element-by-element operations on vectors and matrices are the same as those of Table A.2, except '.' must be added before the operator

Table A.2: Vector and Matrix element-by-element operations

Operation	Algebraic Form	MATLAB syntax
addition	$a + b$	`a + b`
subtraction	$a - b$	`a - b`
multiplication	$a \times b$	`a.*b`
division	$a \div b$	`a./b`
exponentiation	a^b	`a.^b`

- Another related case is when a scalar operates on a vector or matrix
- In this case the scalar is applied to each vector or matrix element in a like fashion

Examples:

```
>> A = [1 3 5 7]
    A =     1     3     5     7
>> B = 2*A   % Scalar operating on a vector
    B =     2     6    10    14
>> B = 2.*A
    B =     2     6    10    14
>> C = B./A  % Vector-to-vector point wise
    C =     2     2     2     2
>> D = A.^3
    D =     1    27   125   343
```

Operator Precedence

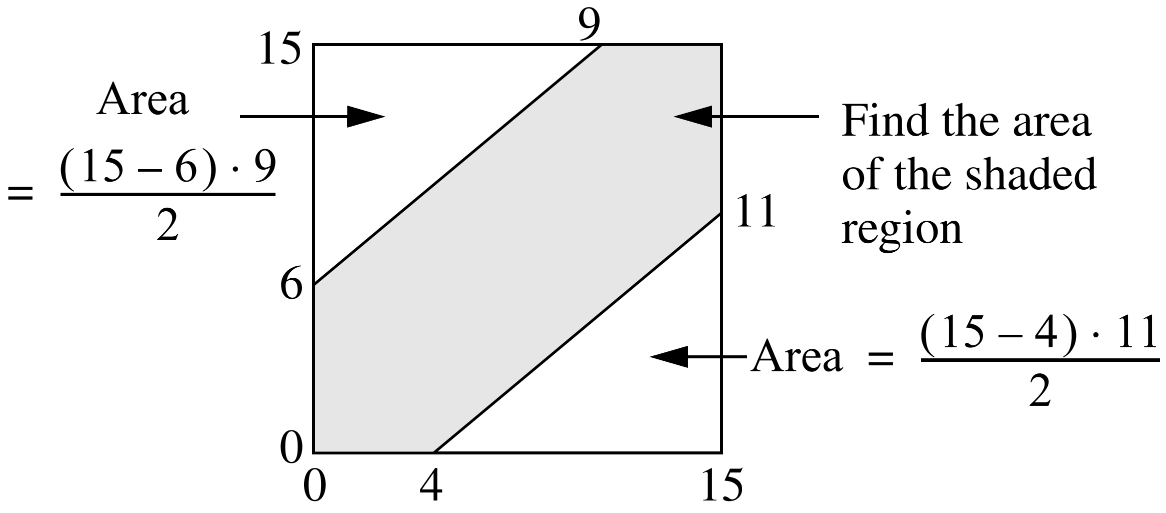
- The order in which expressions are evaluated in MATLAB is fixed according to Table A.3

Table A.3: Operator precedence in MATLAB

Precedence	Operation
1	parenthesis, innermost first
2	exponentiation, left to right
3	multiplication and division, left to right
4	addition and subtraction, left to right

- As long as there are matching left and right parenthesis there is no danger in *over doing it*; the parenthesis help insure that operations are done in the proper order

Example: Find the area of the following shape


15
9
Area
$= \frac{(15-6)\cdot 9}{2}$
Find the area of the shaded region
11
6
Area $= \frac{(15-4)\cdot 11}{2}$
0
0
4
15

- There are several ways to solve this problem, but one that comes to mind is to find the area to the square and then subtract the area of the two right triangles
  - Recall that the area of a triangle is $1/2 \cdot \text{base} \cdot \text{height}$

$$\text{Area} = (15\times 15) - \left[\frac{1}{2}(15-4)11 + \frac{1}{2}(15-6)9\right]$$

  - Code in MATLAB as follows:

```
>> Area = (15*15) - 1/2*((15-4)*11+(15-6)*9)
    Area =   124
```

Numerical Limitations

- The usable range of numbers MATLAB can work with is from 10^{-308} to 10^{308}

- If overflow (number too large) occurs MATLAB indicates a result of `Inf`, meaning infinity
- If underflow (number too small) occurs MATLAB indicates a result of `0`

Example:

```
>> a = 5.00e200; % same as 5*10^200
>> b = 3.00e150; % same as 3*10^150
>> a*b
     ans =   Inf % should be 15.0e350
>> 1/a * 1/b
     ans =     0 % should be (1/5)*(1/3)e-350
```

- Other special cases occur when the conditions of nearly $\text{something}/0 \to \infty$ or $0/0 \to$ occur
- For the first case MATLAB gives an error message of divide by zero and returns `Inf`
- In the second case MATLAB gives an error message of divide by zero and returns `NaN`

Example:

```
>> 1e-30/1e-309 %according to MATLAB like somthing/0
     Warning: Divide by zero.
     ans =   Inf %IEEE notation for ‘infinity’
>> 1e-309/1e-309 %according to MATLAB like 0/0
     Warning: Divide by zero.
     ans =   NaN %IEEE notation for ‘not a number’
```

Additional Plot Types and Plot Formatting

- The *xy* plot is the most commonly used plot type in MATLAB
- Engineers frequently plot either a measured or calculated *dependent* variable, say *y*, versus an *independent* variable, say *x*
- MATLAB has huge array of graphics functions and there are many variations and options associated with `plot()`
  - To find out more type: `>> help plot`

Special Forms of Plot

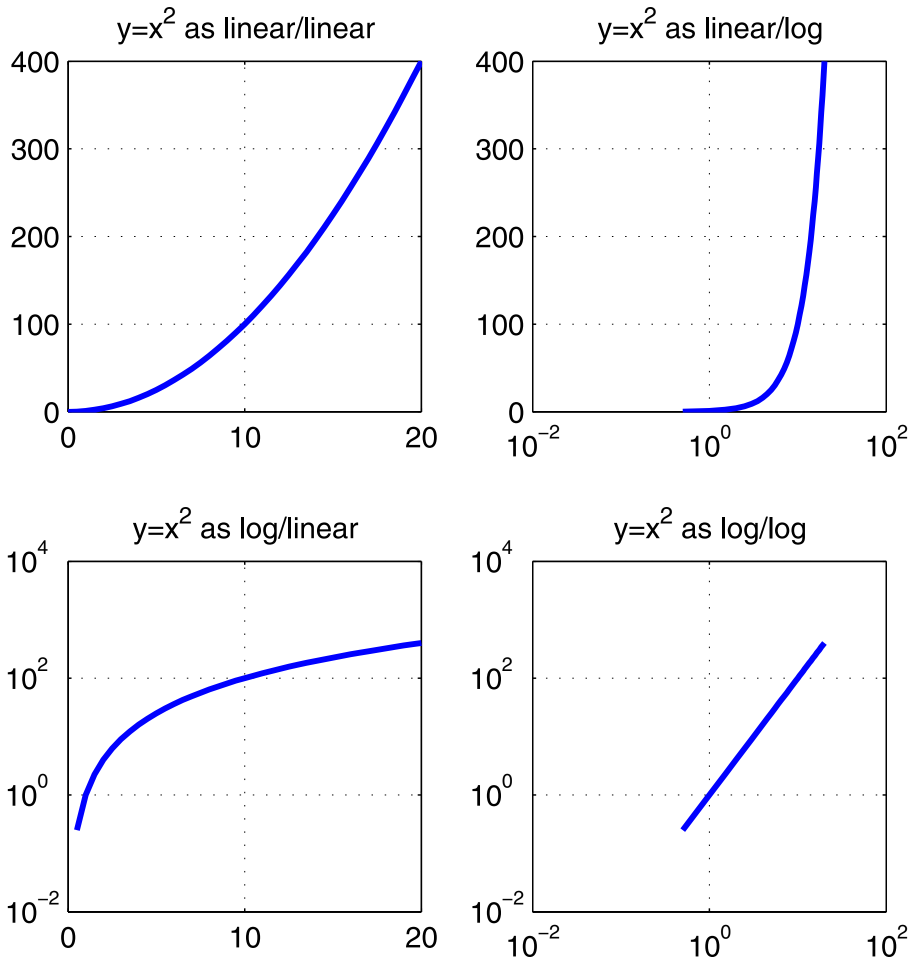
- Linear axes for both *x* and *y* are not always the most useful
- Depending upon the functional form we are plotting we may wish to use a **logarithmic scale** (base 10) for *x* or *y*, or both axes
  - When a log axes is used the variable may range over many orders of magnitude, yet still show detail at the smaller values

Table A.4: Common variations of the plot command

Command	Explanation
`plot(x,y)`	Linear plot of `y` versus `x`
`semilogx(x,y)`	Linear `y` versus log of `x`
`semilogy(x,y)`	Log of `y` versus linear of `x`
`loglog(x,y)`	Log of `y` versus log of `x`

Example: Plotting $y = x^2$ for $0 \le x \le 20$

```
>> x = 0:.5:20;
>> y = x.^2;
>> subplot(2,2,1) % address plot in row 1/column 1
>> plot(x,y),grid,title('y = x^2 as linear/linear')
>> subplot(2,2,2) % address plot in row 1/column 21
>> semilogx(x,y),grid,title('y = x^2 as linear/log')
>> subplot(2,2,3) % address plot in row 2/column 1
>> semilogy(x,y),grid,title('y = x^2 as log/linear')
>> subplot(2,2,4) % address plot in row 2/column 2
>> loglog(x,y),grid,title('y = x^2 as log/log')
>> % Below we export the graphic as encapsulated
>> % postscript with a tiff bitmap overlay
>> print -depsc -tiff quad_plot.eps
```



Multiple (Overlay) Plots

- Using the basic command plot() we can easily plot multiple curves on the same graph
- The most direct way is to simply supply plot() with additional arguments, e.g.,

```
plot(x1,y1,x2,y2,...)
```

- Another way is to use the `hold` command

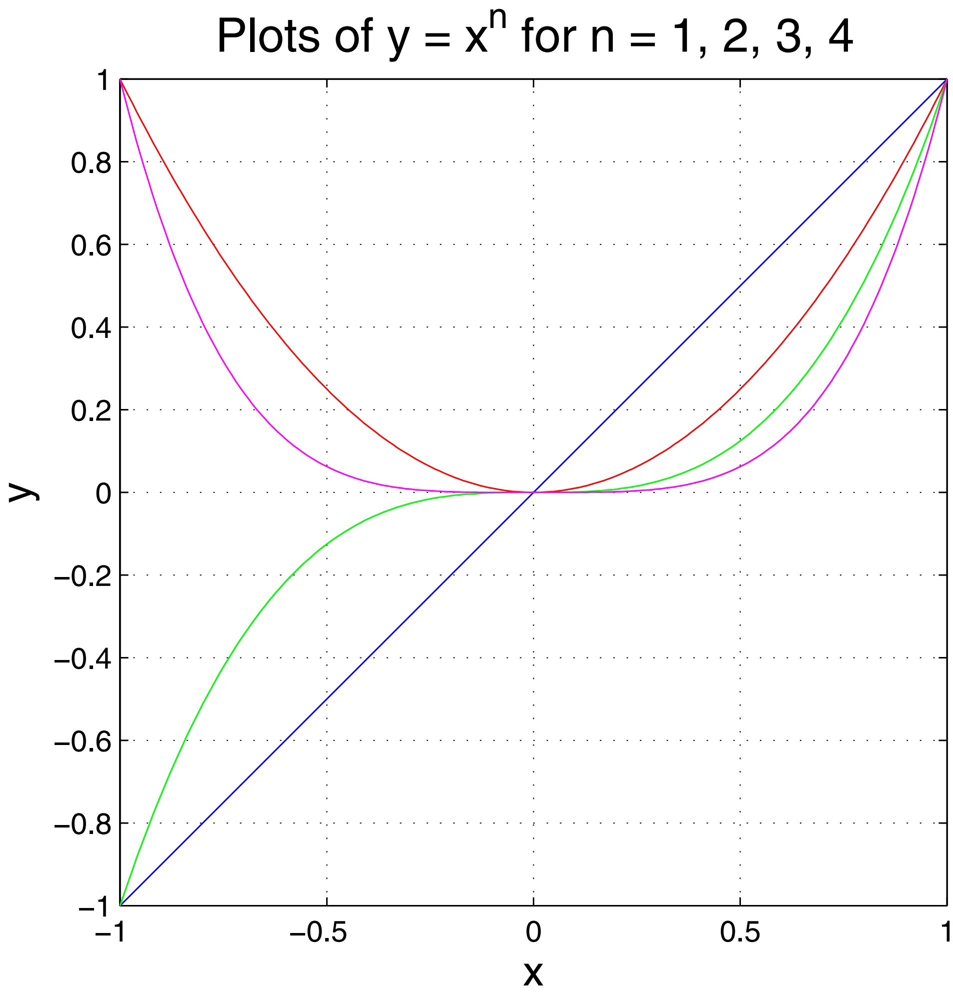
```
plot(x1,y1)
hold %turns plot holding on for the current
    %figure
plot(x2,y2)
hold %turns plot holding off for current figure
```

Example: Plotting a family of polynomials, say

$$y = x^n, -1 \le x \le 1$$

for $n = 1, 2, 3, 4$

```
>> x = -1:.01:1;
>> plot(x,x),axis('square'),grid
>> hold
Current plot held
>> plot(x,x.^2)
>> plot(x,x.^3)
>> plot(x,x.^4)
>> title('Plots of y = x^n for n = 1, 2, 3, & 4',...
        'fontsize',16)
>> ylabel('y','fontsize',14), xlabel('x','fontsize',14)
>> print -depsc -tiff powers.eps
```



Line and Mark Styles

- A wide variety of line and mark styles can be obtained by including a character string argument to `plot()`

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

  where s is a character string (`'a string'`) composed of at most one element from each column of Table A.5
  - The basic variations are line color, line style, and mark styles

Table A.5: Graphics line style, line color, and mark options

Line Color	Marker (symbol) Style	Line Style
`y` yellow	`.` point	`-` solid
`m` magenta	`o` circle	`:` dotted
`c` cyan	`x` x-mark	`-.` dashdot
`r` red	+ plus	`--` dashed
`g` green	`*` star	
`b` blue	`s` square	
`w` white	`d` diamond	
`k` black	`v` triangle (down)	
	^ triangle (up)	
	< triangle (left)	
	> triangle (right)	
	`p` pentagram	
	`h` hexagram	

- As an example,

  ```
  plot(x,y,'c+:')
  ```

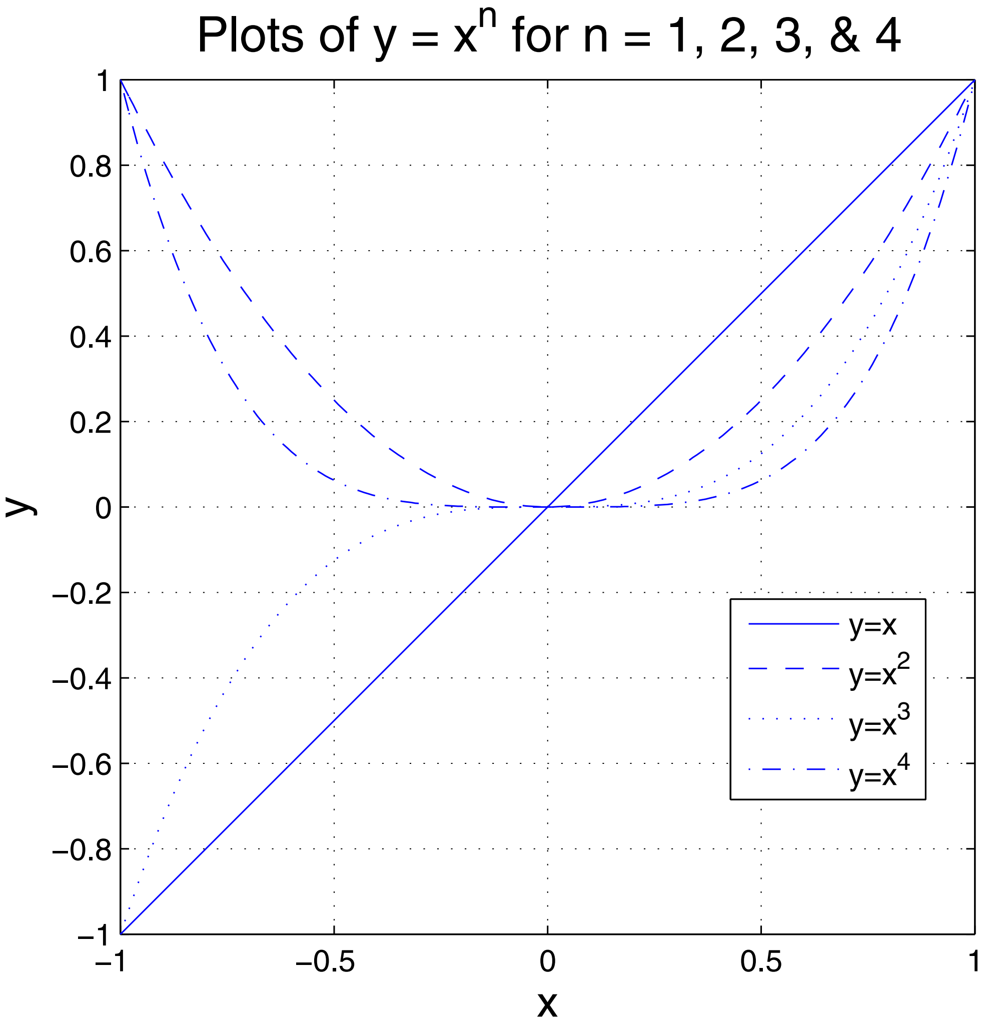
  plots a cyan dotted line with a plus at each data point, while

  ```
  plot(x,y,'bd')
  ```

  plots blue diamond at each data point but does not draw any line

Example: Plotting a family of polynomials (continued). Here we will improve upon a previous multiple plot by coding the line types using markers (symbols) and MATLAB's `legend` command.

```
>> x = -1:.01:1;
>> plot(x,x),axis('square'),grid
>> hold
Current plot held
>> plot(x,x.^2,'--')
>> plot(x,x.^3,':')
>> plot(x,x.^4,'-.')
>> legend('y = x','y = x^2','y = x^3','y = x^4')
>> title('Plots of y = x^n for n = 1, 2, 3, & 4','fon-
tsize',16)
>> ylabel('y','fontsize',14)
>> xlabel('x','fontsize',14)
>> print -depsc -tiff powers_leg.eps
```



Axis Scaling

- When creating plots in MATLAB the axes are initially automatically scaled
- Sometimes custom axis scaling is more desirable
- The `axis` command, which has a large variety of options (type `help axis`), allows the axis scaling to be changed <u>following graph generation</u> via

  ```
  axis([x_min,x_max,y_min,y_max]);
  ```

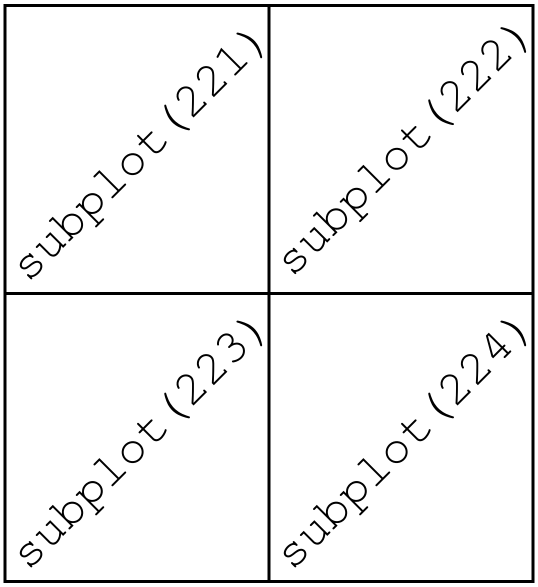
- Another useful command in this area is `zoom` command

  - The zoom command allows the user to zoom into an area of an existing graph by drawing a zoom rectangle with the mouse
  - To turn zoom off for a particular figure window you type `zoom` again

Subplots

- The `subplot` command allows a given figure window in MATLAB to be broken into a rectangular array of plots, each addressed with the standard plot commands
- The command `subplot(m,n,p)` or `subplot(mnp)` splits the graph (figure) window into an array of plots `m`-rows by `n`-columns, and addresses the `p`th subplot
  - The subplot index p counts from left to right beginning with the upper left plot and ending with the lower right plot

A 2x2 Array of Subplots



- The example on page A-18 uses the `subplot` command

Mathematical Functions

Common Math Functions

Table A.6: Common math functions

Function	Description	Function	Description
`abs(x)`	$\lvert x\rvert$	`sqrt(x)`	$\sqrt{x}$
`round(x)`	nearest integer	`fix(x)`	nearest integer
`floor(x)`	nearest integer toward $-\infty$	`ceil(x)`	nearest integer toward ∞
`sign(x)`	$\begin{cases} -1, & x < 0 \\ 0, & x = 0 \\ 1, & x > 0 \end{cases}$	`rem(x,y)`	the remainder of x/y
`exp(x)`	e^x	`log(x)`	natural log $\ln x$
`log10(x)`	log base 10 $\log_{10}x$		

Examples:

```
>> x = [-5.5 5.5];
>> round(x)
ans =    -6     6
>> fix(x)
ans =    -5     5
>> floor(x)
ans =    -6     5
>> ceil(x)
```

```
ans =    -5     6
>> sign(x)
ans =    -1     1
>> rem(23,6)
ans =     5
```

Trigonometric and Hyperbolic Functions

- Unlike pocket calculators, the trigonometric functions always assume the input argument is in radians
- The inverse trigonometric functions produce outputs that are in radians

Table A.7: Trigonometric functions

Function	Description	Function	Description
`sin(x)`	$\sin(x)$	`cos(x)`	$\cos(x)$
`tan(x)`	$\tan(x)$	`asin(x)`	$\sin^{-1}(x)$
`acos(x)`	$\cos^{-1}(x)$	`atan(x)`	$\tan^{-1}(x)$
`atan2(y, x)`	the inverse tangent of y/x including the correct quadrant		

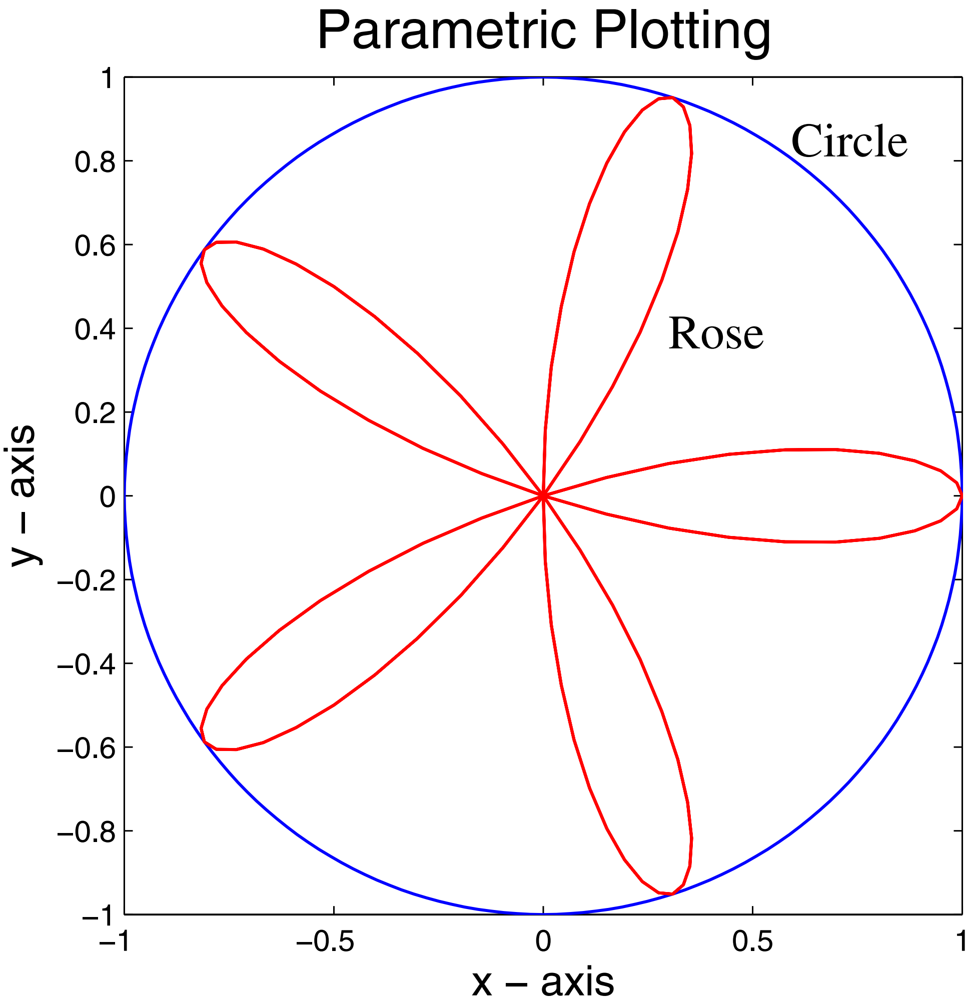
Examples:

- A simple verification that $\sin^2(x) + \cos^2(x) = 1$

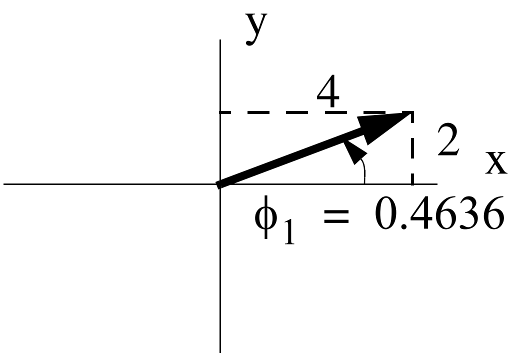
```
>> x = 0:pi/10:pi;
>> [x' sin(x)' cos(x)' (sin(x).^2+cos(x).^2)']
ans =
         0         0    1.0000    1.0000
    0.3142    0.3090    0.9511    1.0000
    0.6283    0.5878    0.8090    1.0000
    0.9425    0.8090    0.5878    1.0000
    1.2566    0.9511    0.3090    1.0000
    1.5708    1.0000    0.0000    1.0000
    1.8850    0.9511   -0.3090    1.0000
    2.1991    0.8090   -0.5878    1.0000
    2.5133    0.5878   -0.8090    1.0000
    2.8274    0.3090   -0.9511    1.0000
    3.1416    0.0000   -1.0000    1.0000
```

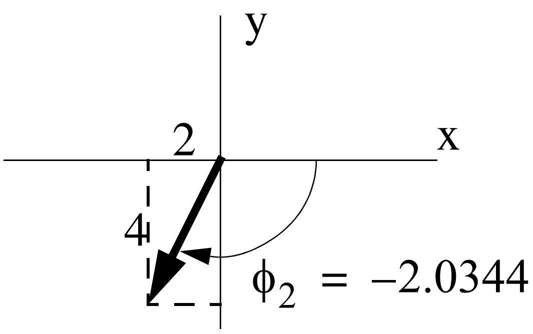
- Parametric plotting:
  - Verify that by plotting $\sin\theta$ versus $\cos\theta$ for $0 \le \theta \le 2\pi$ we obtain a circle

```
>> theta = 0:2*pi/100:2*pi; % Span 0 to 2pi with 100 pts
>> plot(cos(theta),sin(theta)); axis('square')
>> title('Parametric Plotting','fontsize',18)
>> ylabel('y - axis','fontsize',14)
>> xlabel('x - axis','fontsize',14)
>> hold
Current plot held
>> plot(cos(5*theta).*cos(theta), ...
    cos(5*theta).*sin(theta)); % A five-leaved rose
```



- The trig functions are what you would expect, except the features of `atan2(y,x)` may be unfamiliar





```
>> [atan(2/4) atan2(2,4)]
    ans =    0.4636    0.4636 % the same
>> [atan(-4/-2) atan2(-4,-2)]
    ans =    1.1071   -2.0344 % different; why?
```

- The hyperbolic functions are defined in terms of e^x

Table A.8: Hyperbolic functions

Function	Description	Function	Description
`sinh(x)`	$\frac{e^x - e^{-x}}{2}$	`cosh(x)`	$\frac{e^x - e^{-x}}{2}$
`tanh(x)`	$\frac{e^x - e^{-x}}{e^x + e^{-x}}$	`asinh(x)`	$\ln(x + \sqrt{x^2 + 1})$
`acosh(x)`	$\ln(x + \sqrt{x^2 - 1})$	`atanh(x)`	$\ln\sqrt{\frac{1+x}{1-x}}, \lvert x\rvert \le 1$

- There are no special concerns in the use of these functions except that `atanh` requires an argument that must not exceed an absolute value of one

Complex Number Functions

- Before discussing these functions we first review a few facts about complex variable theory
- In electrical engineering complex numbers appear frequently
- A complex number is *an ordered pair of real numbers*[1]

[1] Tom M. Apostle, *Mathematical Analysis*, second edition, Addison Wesley, p. 15, 1974.

denoted (a, b)

- The first number, a, is called the real part, while the second number, b, is called the imaginary part
- For algebraic manipulation purposes we write $(a, b) = a + ib = a + jb$ where $i = j = \sqrt{-1}$; electrical engineers typically use j since i is often used to denote current

Note: $\sqrt{-1} \times \sqrt{-1} = -1 \Rightarrow j \times j = -1$

- For complex numbers $z_1 = a_1 + jb_1$ and $z_2 = a_2 + jb_2$ we define/calculate

$$z_1 + z_2 = (a_1 + a_2) + j(b_1 + b_2) \text{ (sum)}$$

$$z_1 - z_2 = (a_1 - a_2) + j(b_1 - b_2) \text{ (difference)}$$

$$z_1 z_2 = (a_1 a_2 - b_1 b_2) + j(a_1 b_2 + b_1 a_2) \text{ (product)}$$

$$\frac{z_1}{z_2} = \frac{(a_1 a_2 + b_1 b_2) - j(a_1 b_2 - b_1 a_2)}{a_2^2 + b_2^2} \text{ (quotient)}$$

$$|z_1| = \sqrt{a_1^2 + b_1^2} \text{ (magnitude)}$$

$$\angle z_1 = \tan^{-1}(b_1 / a_1) \text{ (angle)}$$

$$z_1{}^* = a_1 - jb_1 \text{ (complex conjugate)}$$

- MATLAB is consistent with all of the above, starting with the fact that `i` and `j` are predefined to be $\sqrt{-1}$

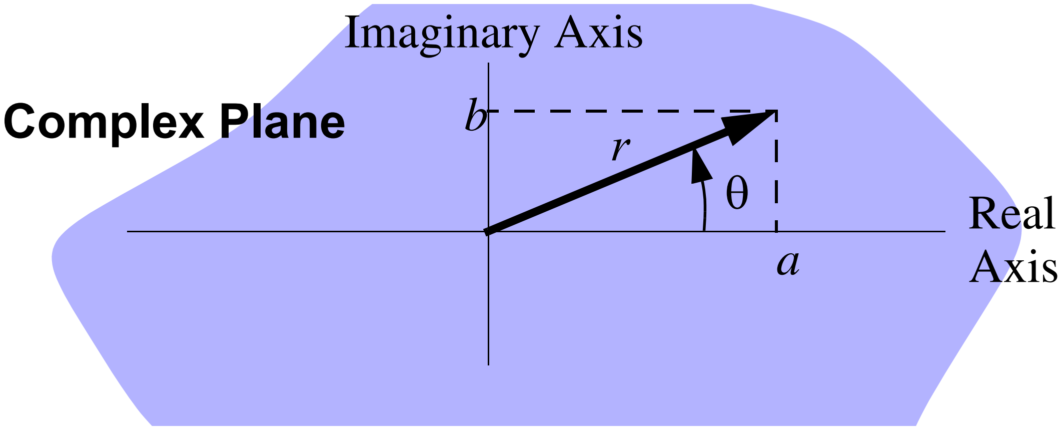
- The *rectangular form* of a complex number is as defined above,

$$z = (a, b) = a + jb$$

- The corresponding *polar form* is

$$z = r\angle\theta$$

where $r = \sqrt{a^2 + b^2}$ and $\theta = \tan^{-1}(b/a)$



- MATLAB has five basic complex functions, but in reality most all of MATLAB's functions accept complex arguments and deal with them correctly

Table A.9: Basic complex functions

Function	Description
`conj(x)`	Computes the conjugate of $z = a + jb$ which is $z^* = a - jb$
`real(x)`	Extracts the real part of $z = a + jb$ which is $\text{real}(z) = a$

Table A.9: Basic complex functions

Function	Description
`imag(x)`	Extracts the imaginary part of $z = a + jb$ which is $\text{imag}(z) = b$
`angle(x)`	computes the angle of $z = a + jb$ using `atan2` which is `atan2(imag(z),real(z))`

Euler's Formula: A special mathematical result, of special importance to electrical engineers, is the fact that

$$e^{jb} = \cos b + j\sin b \tag{A.1}$$

- Turning (3.1) around yields

$$\sin\theta = \frac{e^{j\theta} - e^{-j\theta}}{2j} \tag{A.2}$$

$$\cos\theta = \frac{e^{j\theta} + e^{-j\theta}}{2} \tag{A.3}$$

- It also follows that

$$z = a + jb = re^{j\theta} \tag{A.4}$$

where

$$r = \sqrt{a^2 + b^2},\ \theta = \tan^{-1}\frac{b}{a},\ a = r\cos\theta,\ b = r\sin\theta$$

- Some examples:

```
>>>> z1 = 2+j*4; z2 = -5+j*7;
>> [z1 z2]
ans =
   2.0000 + 4.0000i  -5.0000 + 7.0000i
>> [real(z1) imag(z1) abs(z1) angle(z1)]
ans =
   2.0000    4.0000    4.4721    1.1071
>> [conj(z1) conj(z2)]
ans =
   2.0000 - 4.0000i  -5.0000 - 7.0000i
>> [z1+z2 z1-z2 z1*z2 z1/z2]
ans =
  -3.0000 +11.0000i   7.0000 - 3.0000i
 -38.0000 - 6.0000i   0.2432 - 0.4595i
```

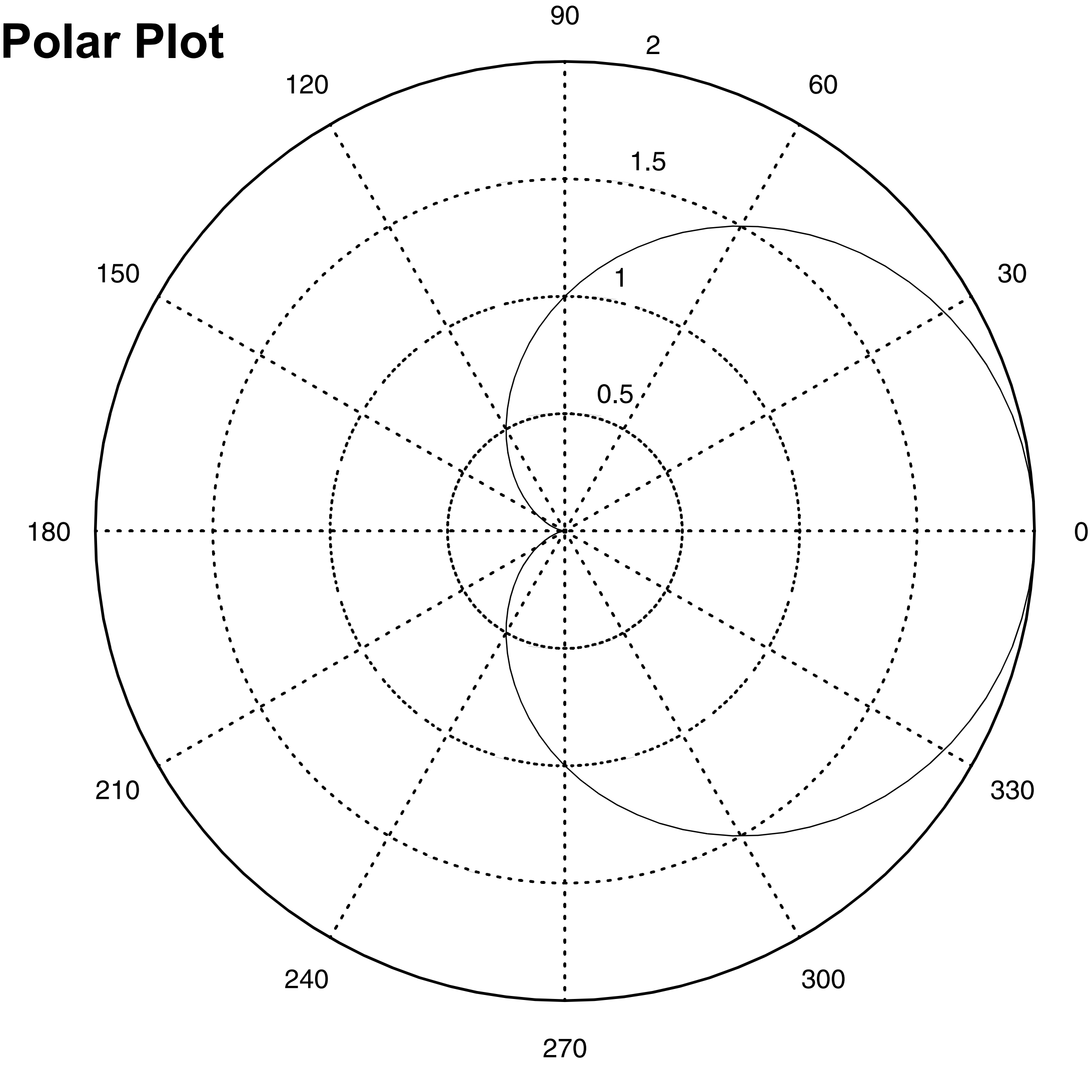
Polar Plots: When dealing with complex numbers we often deal with the polar form. The plot command which directly plots vectors of magnitude and angle is `polar(theta,r)`

- This function is also useful for general plotting
- As an example the equation of a cardioid in polar form, with parameter a, is

$$r = a(1 + \cos\theta), 0 \le \theta \le 2\pi$$

```
>> theta = 0:2*pi/100:2*pi; % Span 0 to 2pi with 100 pts
>> r = 1+cos(theta);
>> polar(theta,r)
```

Polar Plot



Roots of Polynomials

- Recall that the roots of $y = f(x)$ are those values of x where $f(x) = 0$
- For polynomials with real number coefficients, the roots may be real numbers and/or pairs of complex conjugate numbers
  - For a third-order polynomial the roots may be either three real roots or one real root and a complex pair
  - For a second-order polynomial the roots may be either two

real roots or a complex conjugate pair

- If we plot $f(x)$ we find that the real roots correspond to those locations where the polynomial crosses the x-axis
- Recall that for a quadratic $y(x) = a_0x^2 + a_1x + a_2$ the roots are (quadratic formula)

$$r_1, r_2 = \frac{-a_1 \pm \sqrt{a_1^2 - 4a_0a_2}}{2a_0}$$

<u>Note</u>: The roots are complex if $4a_0a_2 > a_1^2$

- Hand calculation of polynomial roots becomes impractical as the order increases, hence we often turn to a numerical solution
- The MATLAB function `roots(a)` finds the roots of a polynomial with coefficient vector `a`

Example: $f(x) = 2x^4 + 5x^3 + 10x^2 + 7x + 6$

```
>> p = [2 5 10 7 6];
>> r = roots(p)
r =
  -1.0000 + 1.4142i
  -1.0000 - 1.4142i
  -0.2500 + 0.9682i
  -0.2500 - 0.9682i
>> polyval(p,r)% Check by evaluating p
             % at the root locations
ans =  1.0e-013 *
  -0.1243 + 0.0089i
  -0.1243 - 0.0089i
```

```
    -0.0533 + 0.0355i
    -0.0533 - 0.0355i % Functional values are small
```

- Given the roots of a polynomial $r_1, r_2, \ldots, r_N$ we know that

$$\begin{aligned} f(x) &= a_0 x^N + a_2 x^{N-1} + \ldots + a_{N-1} x + a_N \\ &= (x - r_1)(x - r_2)\ldots(x - r_N) \end{aligned}$$

- The MATLAB function `poly(r)` effectively reconstructs the polynomial coefficient vector, `a`, given the vector of roots by repeated polynomial multiplication

3-D Plotting Options

- In MATLAB there are a variety of standard 3-D plotting functions; a short table is given below

Table A.10: Plotting functions of two variables

Plot Function	Description
`mesh(x,y,z)`	Plots a mesh type surface defined by matrix `z`. `x` and `y` are either vectors of the *x* and *y* range values or the corresponding grid matrices.
`surf(x,y,z)`	Plots a shaded surface defined by matrix `z`. `x` and `y` are as defined above
`contour(x,y,z)`	A flat map containing contour lines of the surface defined by `z`. x and y are as defined above. The number of contour lines is chosen automatically

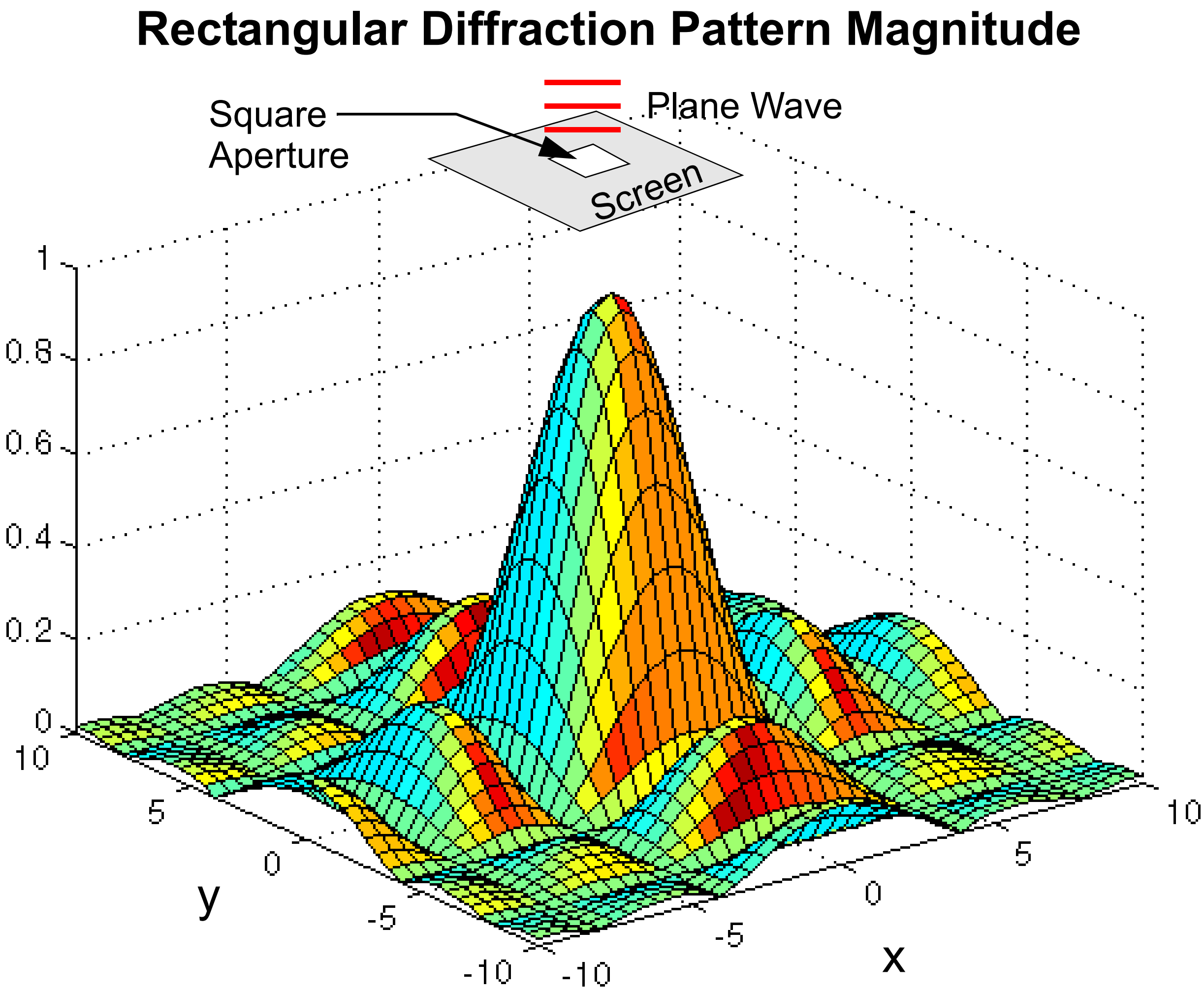
Table A.10: Plotting functions of two variables

Plot Function	Description
`contour(x,y,z,v)`	Same as three argument contour except `v` defines the contour lines
`meshc(x,y,z)`	A mesh plot with a contour plot sitting in the $z = 0$ plane

Example: Evaluate the diffraction pattern of a square aperture

$$z = f(x, y) = \frac{\sin(\pi x/4)}{\pi x/4} \cdot \frac{\sin(\pi y/4)}{\pi y/4}$$

```
>> x = -10:.5:10; y = -10:.5:10;
>> [x_grid,y_grid] = meshgrid(x,y);
>> % MATLAB defines sin(pi*x)/(pi*x) as sinc(x)
>> z = sinc(x_grid/4).*sinc(y_grid/4);
>> surfl(x,y,abs(z)) % Surf. plot with lighting effects
>> title('Rectangular Diffraction Pattern Magnitude',
     ... 'fontsize',16)
>> ylabel('y','fontsize',14)
>> xlabel('x','fontsize',14)
```



Data Analysis Functions

The analysis of data coming from analytical calculations, computer simulations, or actual experimental data, is a requirement in most all engineering projects. With MATLAB we have a powerful set of predefined data analysis, and as we will see later, we can also write custom functions as needed.

Simple Analysis

- The first group of data analysis functions to consider finds maximums, minimums, sums, and products of vectors or columns of matrices

Table A.11: Simple data analysis functions

Function	Description	Function	Description

`max(x)`
`[y,k]= max(x)`
	Returns `y` the largest value in vector `x` or each column of `x`. Optionally `k` is the index where the max occurs	`max(x,y)`	Returns a matrix full of the larger of the two values in `x` and `y` at the corresponding matrix indices

`min(x)`
`[y,k]= min(x)`
	Same as `max(x)` except minimum	`min(x,y)`	Same as `max(x,y)` except minimum

`sum(x)`	The scalar (vector of column sums) $\sum_{n=1}^{N} x(n)$	`prod(x)`	The scalar (vector of column products) $\prod_{n=1}^{N} x(n)$

Table A.11: Simple data analysis functions (Continued)

Function	Description	Function	Description
`cumsum(x)`	The running or cumulative sum version of the above, hence a vector or matrix	`cumprod(x)`	The running or cumulative product version of the above, hence a vector or matrix

Example:

```
>> x = 0:10;
>> x_max = max(x); x_min = min(x);
>> x_sum = sum(x); x_prod = prod(x)
>> [x_max x_min x_sum x_prod]
ans =
    10      0     55      0
>> cumsum(x)
ans =
     0      1      3      6     10     15     21     28     36
    45     55
>> cumprod(x)
ans =
     0      0      0      0      0      0      0      0      0
     0      0 % Why zeros?
```

Sample Statistics

- When the data we are operating on is viewed as the result of an experiment or the sampling of a *population*, then we may be interested in *sample statistics*

Table A.12: Sample statistics

Function	Description
`mean(x)`	The mean or average of the elements in `x` $\mu = \frac{1}{N}\sum_{n=1}^{N} x(n)$ = `cumsum(x)/length(x)` The column mean for `x` a matrix
`var(x)`	The average squared variation of `x` about its mean $\sigma^2 = \frac{1}{N-1}\sum_{n=1}^{N} [x(n)-\mu]^2$
`std(x)`	The square root of the variance = σ

Flow Control Using a Selection Statement

- In all programming languages there are means (usually more than one) to control the flow of the program
- The most common means of flow control is via an **if** statement/code block

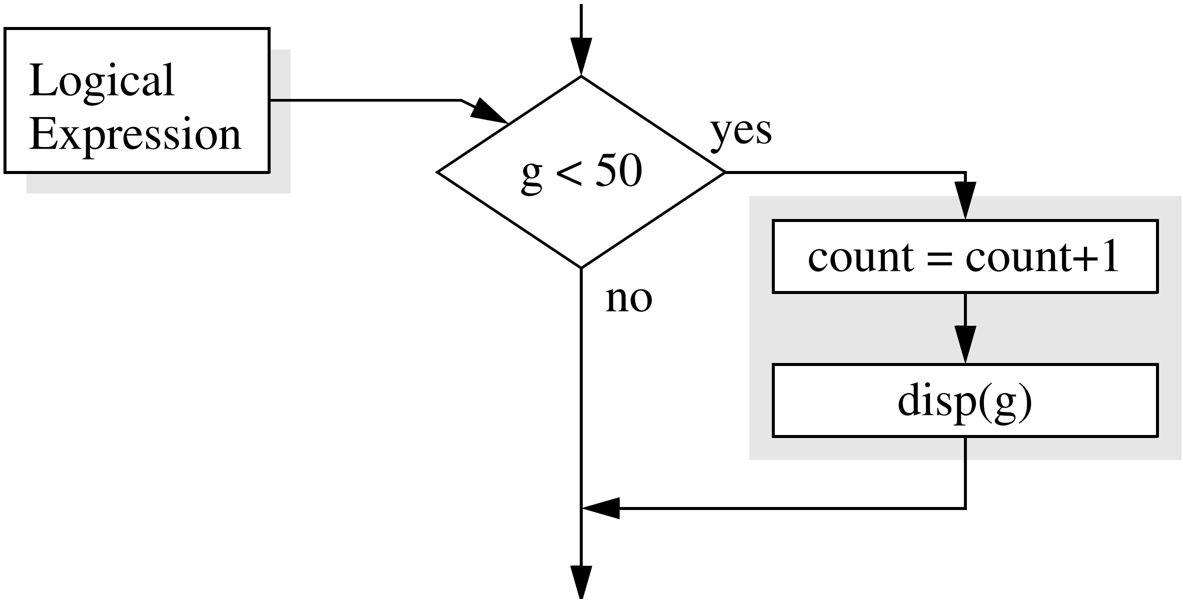
The MATLAB `if` Statement

- The full form of the `if` code block is

```
if logical_expression_#1
  statements
elseif logical_expression_#2
```

```
        statements
              .
              .
              .
    elseif logical_expression_#N
        statements

     else  % Default code to execute is in here
        statements
     end
```

- The general `if` code block shown above is very powerful, but all the features that it offers may be confusing at first, and in fact are often not used
- In flow chart form a simple flow control may be the following:



- The MATLAB code is the following:

```
%some program statements
if g < 50  % Note statements inside if block are
           % indented to improve readability.
    count = count + 1;
```

```
        disp(g);
    end
    %more program statements
```

- In an `if` code block decisions are made as a result of evaluating a *logical expression(s)*, e.g.,

```
    if logical_expression
        run_this_code
    end
```

  - If the logical expression is true, then the code placed between the `if` and `end` statements is evaluated, otherwise we skip over it to the first code statement following the `end` statement

- Before studying more `if` code block examples, we will briefly study the use of relational and logical operators to construct logical expressions

Relational and Logical Operators

Using *relational operators* we can construct a single logical expression. If needed, several logical expressions can be combined into one logical expression using *logical operators*.

- MATLAB supports six relational operators as described in Table A.13

Table A.13: Relational operators

Operator	Purpose/Functionality
<	Used to form the logical expression $a < b$ which returns 1 if true and 0 if false

Table A.13: Relational operators

Operator	Purpose/Functionality
<=	Used to form the logical expression $a \le b$ which returns 1 if true and 0 if false
>	Used to form the logical expression $a > b$ which returns 1 if true and 0 if false
>=	Used to form the logical expression $a \ge b$ which returns 1 if true and 0 if false
==	Used to form the logical expression $a = b$ which returns 1 if true and 0 if false
~=	Used to form the logical expression $a \ne b$ which returns 1 if true and 0 if false

- When the logical operators are used in expressions involving vectors or matrices, the returned value is a vector or matrix containing 0's and 1's respectively

Example: Let $a = \begin{bmatrix} 1 & 5 & 8 \end{bmatrix}$ and $b = \begin{bmatrix} 2 & 3 & 8 \end{bmatrix}$

```
>> a = [1 5 8]; b = [2 3 8];
>> a < b
ans =
     1     0     0
>> a > b
ans =
     0     1     0
>> a == b
ans =
     0     0     1
>> a ~= b
```

```
ans =
     1     1     0
```

- More complex logical expressions can be obtained by combining several sub expressions using the logical operators given in Table A.14

Table A.14: Logical operators

Operator/ Symbol	Purpose/Functionality
`not, ~`	`not(a)` or `~a` returns inverts a 1 to 0 and a 0 to a 1
`and, &`	`and(a,b)` or `a&b` returns 1 if `a` <u>and</u> `b` are both true (1), otherwise 0
`or, |`	`or(a,b)` or `a|b` returns 1 if `a` <u>or</u> `b` are true (1), otherwise 0
`xor`	`xor(a,b)` returns 1 if `a` <u>and</u> `b` are different, otherwise 0

Example: Continuation of previous example

```
>> a = [1 5 8]; b = [2 3 8];
>> (a < b)
ans =
     1     0     0
>> ~(a < b)
ans =
    0     1     1
>> [(a < b) (a > b)]
ans =
     1     0     0     0     1     0
>> (a > b) & (a < b)
```

```
    ans =
         0     0     0
    >> and(a > b , a < b)
    ans =
        0     0     0
```

Example: Determine if the following expressions are true or false. Then check your answers using MATLAB.

$$a = 5.5, b = 1.5, k = -3$$

```
>> a = 5.5; b = 1.5; k = -3;
```

1. `a < 10.0`

   By inspection the statement $a < 10.0$ is true, or logical 1

   ```
   >> a < 10.0
   ans =     1
   ```

5. `~(a == 3*b)`

   We know that $a = 5.5$ and $3b = 4.5$, thus the interior logical expression is false (logical 0), and its complement is true (logical 1)

   ```
   >> ~(a == 3*b)
   ans =     1
   ```

7. `a<10 & a>5`

   By inspection $a < 10$ is true and $a > 5$ is true, the logical and of these to expressions is true or logical 1

   ```
   >> a<10 & a>5
   ans =     1
   ```

More `if` Code Blocks

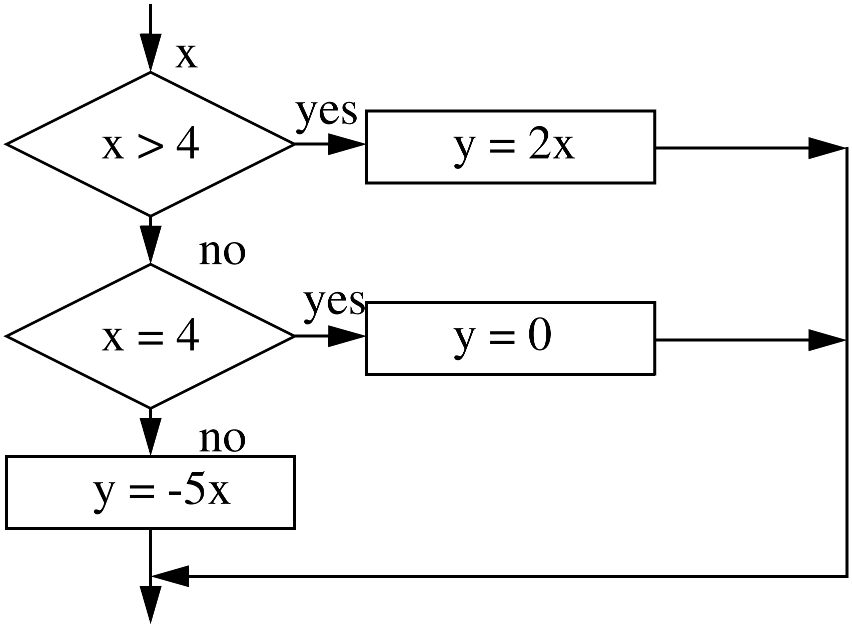
In this section we study the `else` and `elseif` clauses that are part of the complete `if` code block, and also consider nested `if` code blocks

- The stripped down `if` code block discussed earlier is useful when an operation is either performed or not performed
- When say two or more different operations are desired, depending upon a corresponding set of logical expression outcomes, we can use the full `if` code block

Example: Suppose we want to conditionally evaluate

$$y = \begin{cases} 2x, & x > 4 \\ 0, & x = 4 \\ -5x, & \text{otherwise} \end{cases} \tag{A.5}$$

- In flow chart form we have



- In MATLAB we have:

```
if x > 4
    y = 2*x;
elseif x == 4
```

```
    y = 0;
else
    y = -5*x;
end
```

The switch (case) Code Block

- New to MATLAB 5 is the switch or case code block
- Most all high level languages provide the program flow construct, so it only seems reasonable that MATLAB has adopted it as well
- The basic form of the switch statement is

```
switch switch_expr
    case case_expr,
      statement, ..., statement
    case {case_expr1, case_expr2, case_expr3,...}
      statement, ..., statement
   ...
    otherwise,
      statement, ..., statement
  end
```

  – The basic functionality of the switch statement is to *switch* to the first code following a particular case statement where the switch expression and case expressions match

  – The code following the otherwise statement is run only if no match is found in the above case statements

  – Code execution continues following the end statement

Example: Adjust program flow based on a user entered text string.

```
% MATLAB script file switch_demo.m
% Demo of the Switch (Case) Code Block
input_text = ...
    input('Enter some text in single quotes: ');
%Make sure case is not an issue:
input_text = lower(input_text);
switch input_text
case 'yes'
   disp('You have answered the question with a yes.');
case 'no'
   disp('You have answered the question with a no.');
otherwise
   disp('You have not answered the question with a yes
        or no.');
end
```

- Sample output

```
>> switch_demo
Enter Some Text in single quotes: 'help'
You have not answered the question with a yes or no.
>> switch_demo
Enter Some Text in single quotes: 'NO'
You have answered the question with a no.
```

Example: Provide a MATLAB code block that performs the steps indicated.

2. If $\ln(x) > 3$ set `time` equal to 0 and increment `count` by one

```
if log(x) > 3
    time = 0;
    count = count + 1;
end
```

4. If $\mathrm{dist} \geq 100.0$ increment `time` by 2.0. If $50 < \mathrm{dist} < 100$ increment `time` by 1. Otherwise increment `time` by 0.5.

```
if dist >= 100.0
    time = time + 2.0;
elseif (dist > 50) & (dist < 100)
    time = time + 1;
else
    time = time + 0.5;
end
```

Logical Functions

- Logical functions are useful is vectorizing algorithms for efficient execution in the MATLAB environment
- Six functions of particular interest are given in Table A.15

Table A.15: Logical functions

Function	Description
`any(x)`	Returns 1 if <u>any</u> element in `x` is nonzero, 0 otherwise. For a matrix returns a row vector whose elements are 1 if the corresponding column of `x` contains <u>any</u> nonzero element.

Table A.15: Logical functions

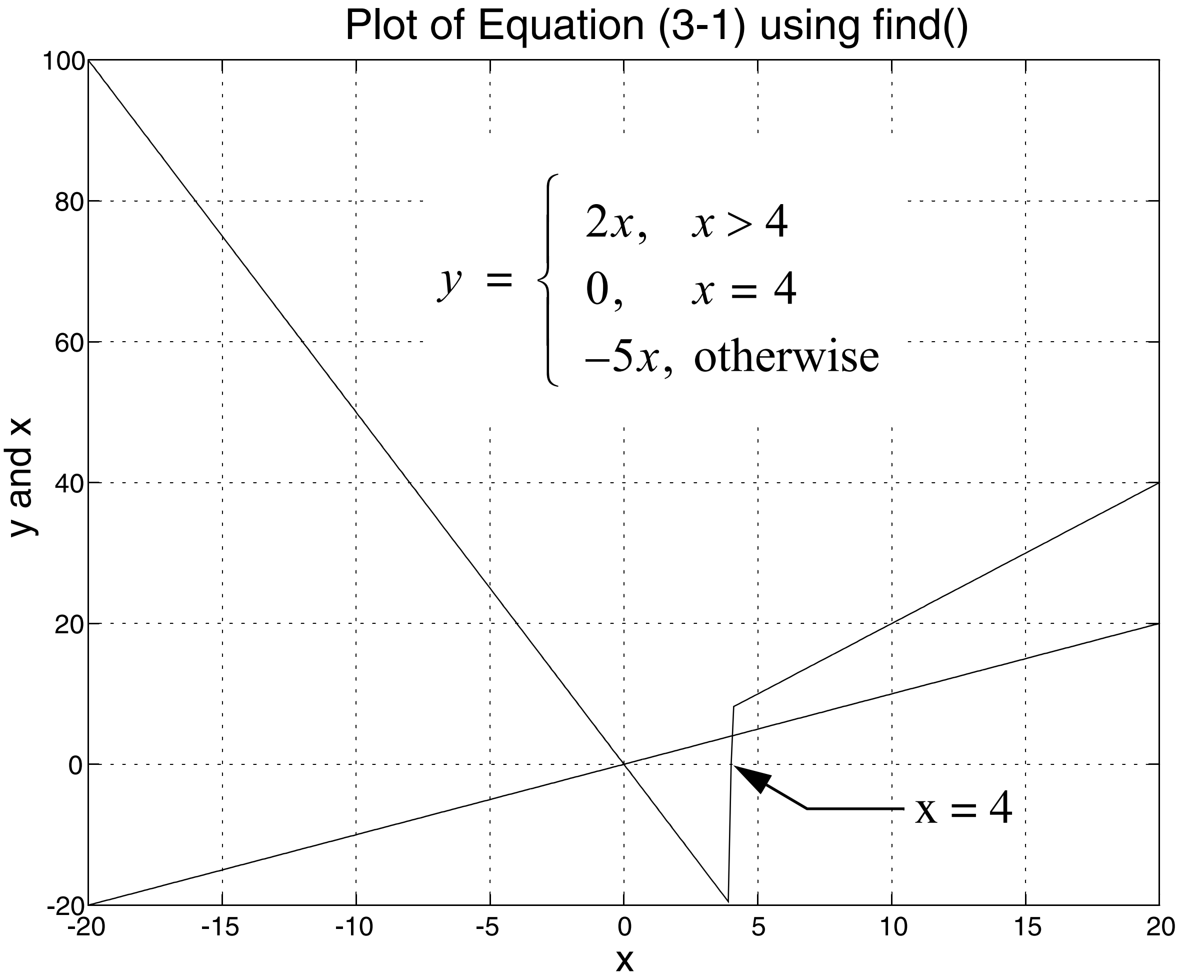
Function	Description
`all(x)`	Returns 1 if all elements in `x` is nonzero, zero otherwise. For a matrix returns a row vector whose elements are 1 if the corresponding column of `x` contains all nonzero elements.
`find(x)`	Returns a vector of the indices of `x` corresponding to nonzero elements. For `x` a matrix `x` is first reshaped into a single column vector formed from the columns of the original `x`, then processed as before.
`isnan(x)`	Returns a matrix filled with 1's where the elements of `x` are `Nan` and 0 otherwise.
`finite(x)`	Returns a matrix filled with 1's where the elements of `x` are finite and 0 if they are infinite or `Nan`.
`isempty(x)`	Returns 1 if `x` is an empty matrix, 0 otherwise.

- The `find` function is particularly useful in vectorizing the evaluation of expressions such as (A.5)
- We can modify the code block on p. 47–48 and place it into an m-file as follows:

```
% MATLAB script file eqn_A_5.m
% Vectorize equation (3-1)
%Work x > 4 case:
indices = find(x > 4);
y(indices) = 2*x(indices);
%Work x == 4 case:
indices = find(x == 4);
y(indices) = 0*x(indices);
%Work x < 4 case:
indices = find(x < 4);
y(indices) = -5*x(indices);
```

- Usage of `eqn_A_5.m`:

```
>> x = -20:.1:20;
>> eqn_A_5
>> plot(x,y)
>> eqn_A_5
>> plot(x,y)
>> plot(x,x,x,y)
>> grid
>> title('Plot of Equation (A-5) using find()',...
        'fontsize',16)
>> ylabel('y and x','fontsize',14)
>> xlabel('x','fontsize',14)
```



Writing MATLAB Functions

- A user written functions are one of the main features of the most power features of the MATLAB environment
- A MATLAB function is very similar to the script files we have created in the past
  - A function is created using the m-file editor and saved in a file with the `.m` extension
  - A function type m-file is different than a script since the file must begin with the *keyword* `function` followed by

a list of the values the function returns, the function name, and a list of the input arguments used by the function

```
function [rvar1,rvar2,...] =  my_fct(in1,in2,...)
% A short description of the function is typed to
% serve as on-line help for the function should
% the user type >> help my_fct
.
.
******Function Body Code is Here **********
.
% Somewhere in the function body we use
% in1, in2, ... to perform scalar, vector, or matrix
% calculations. The results are assigned to the
% return variables, rvar1, rvar2, etc.
%
```

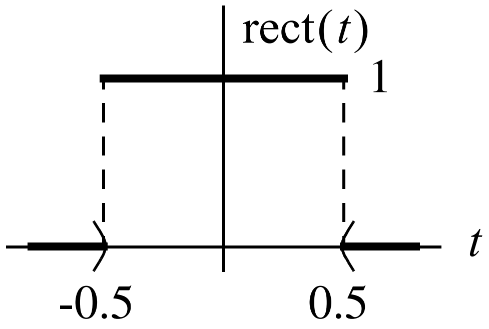
- The official MATLAB definition of a function m-file is:

```
>> help function
 FUNCTION Add new function.
    New functions may be added to MATLAB's vocabulary if
they are expressed in terms of other existing functions.
The commands and functions that comprise the new func-
tion must be put in a file whose name defines the name
of the new function, with a filename extension of '.m'.
At the top of the file must be a line that contains the
syntax definition for the new function. For example, the
existence of a file on disk called stat.m with:

            function [mean,stdev] = stat(x)
            n = length(x);
            mean = sum(x) / n;
            stdev = sqrt(sum((x - mean).^2)/n);

defines a new function called STAT that calculates the
```

```
mean and standard deviation of a vector. The variables
within the body of the function are all local variables.
    See SCRIPT for procedures that work globally on the
work-space.
```

- A particular caution in defining your own functions is to make sure that your names do not conflict with any of MATLAB's predefined functions
- The m-file name <u>must</u> be the same as the function name, e.g., `my_fct` is saved as `my_fct.m`
- All variables created and used within a function are local to that function, and will be destroyed when you are done using the function
- Likewise the only MATLAB workspace variables that the function has knowledge of are those that are passed into it via the input parameter list, i.e., `in1`, `in2`, etc.,

Example: A common function used in signals and systems problem solving is a function for generating a rectangular pulse



$$\mathrm{rect}(t) = \begin{cases} 1, & |t| \le \dfrac{1}{2} \\ 0, & \text{otherwise} \end{cases}$$

- We would like to create a new function `rect(t)` that allows to input a <u>vector</u> (a scalar version would trivial and not as useful) of time samples `t` and returns the corresponding functional values rect(t)

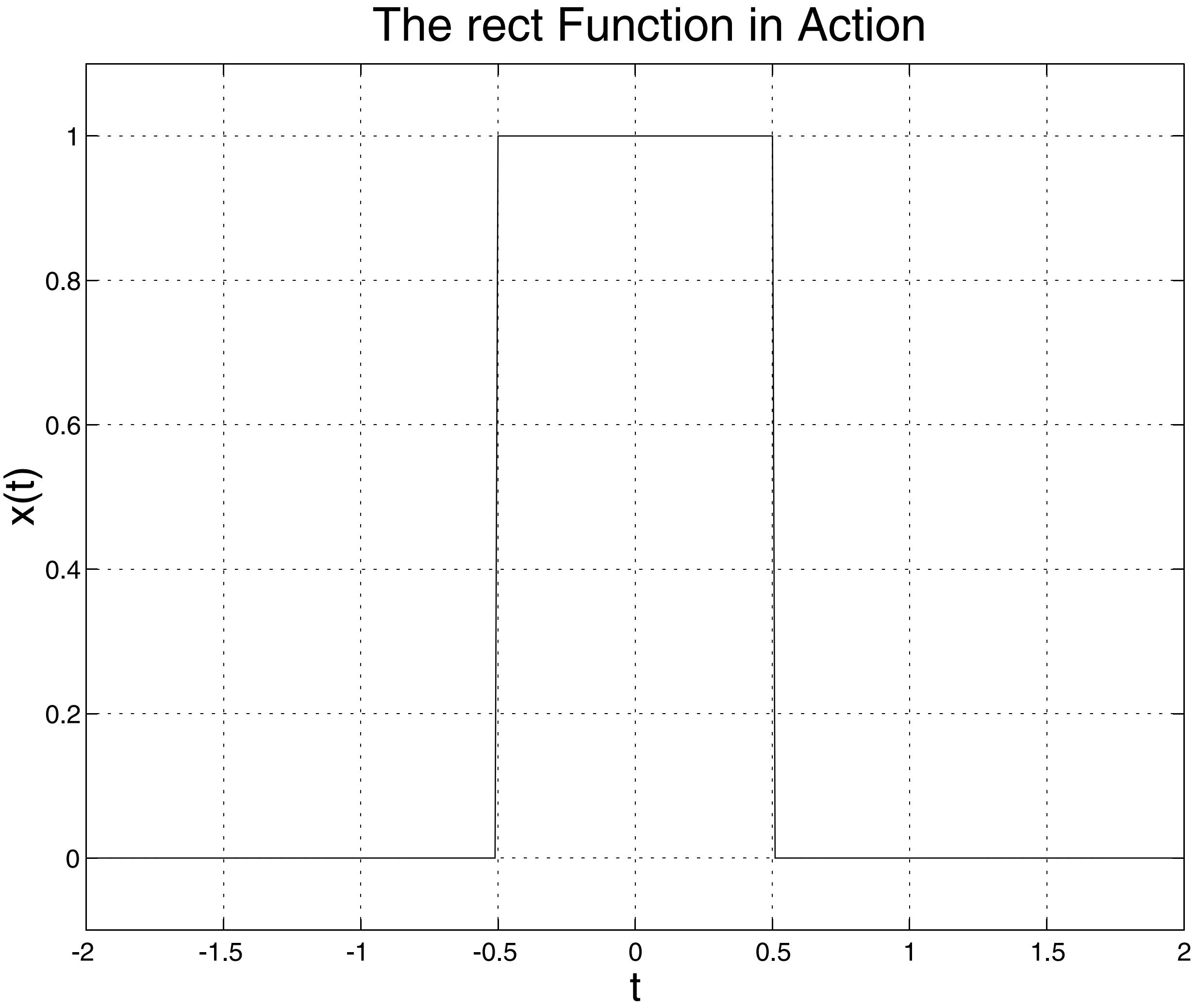
```
function x = rect(t)
%  RECT x = rect(t): A function that is defined
%        to be 1 on [-0.5,0.5] and 0 otherwise.

% Initialize a vector with zeros in it that is
% the same length as the input vetor t:
x = zeros(size(t));
% Create an index vector that holds the indices
% of t where abs(t) <= 0.5:
set1 = find(abs(t) <= 0.5);
% Use set1 to change the corresponding values
% of x from zero to one:
x(set1) = ones(size(set1));
% We are finished!
```

- Now we need to test the function and see if it performs as expected
  - Check the on-line help

```
    >> help rect
       RECT x = rect(t): A function that is defined
             to be 1 on [-0.5,0.5] and 0 otherwise.
```

  - Run a test vector into the function and plot the results

```
>> t = -2:.01:2;
>> x = rect(t);
>> plot(t,x); grid;
>> axis([-2 2 -.1 1.1])
>> title('The rect Function in Action', ...
   'fontsize',18)
>> ylabel('x(t)','fontsize',16)
>> xlabel('t','fontsize',16)
```



Matrix Manipulation Functions

- We know that a specialty of MATLAB is working with matrices
- Early on in this course we investigated the colon operator as a basic tool for matrix manipulation
- In this section additional functions for matrix manipulation are be studied

Rotation

- A matrix can be rotated in the counter clockwise direction using the function `rot90`
  - `rot90(A)` rotates `A` by 90° counterclockwise
  - `rot90(A,n)` rotates `A` by $n \cdot 90°$ counterclockwise

```
>> A = [1 2; 3 4]
A =
     1     2
     3     4
>> [rot90(A) rot90(A,2)]
ans =
     2     4     4     3
     1     3     2     1
```

Flipping

- A matrix function that finds uses in signal processing is the ability to flip a matrix either from left-to-right, `fliplr(A)`, or from up-to-down, `flipud`

```
>> A = [1 2; 3 4]
A =
     1     2
     3     4
>> [fliplr(A) flipud(A)]
ans =
     2     1     3     4
     4     3     1     2
```

Reshaping

- The $m \times n$ size of a matrix can be reshaped using `reshape(A,m,n)` so long as the product $m \cdot n$ is held constant

```
>> B = [1 2 3 4; 5 6 7 8] % 2 x 4 = 8
B =
     1     2     3     4
     5     6     7     8
>> reshape(B,4,2) % 4 x 2 = 8, taken in columns from B
ans =
     1     3
     5     7
     2     4
     6     8
>> reshape(B,1,8) % 1 x 8 = 8, taken in columns from B
ans =
     1     5     2     6     3     7     4     8
```

Looping Structures

- The ability to repeat a set of statements with certain parameters changing is a feature of all modern programming languages
- MATLAB provides the `for` loop and the `while` loop for performing such operations, with the caution that *vectorizing* algorithms is preferred since loops slow down program execution

The `for` Loop

- The basic structure of a `for` loop is the following

```
for index=start_value:stop_value
    % ...
    % Code to be executed repeatedly
```

```
         % ...
     end %This marks the end of the index for loop
```

- Often times we design a `for` loop to simply run over the index values of a vector we wish to operate on

Example: A `for` loop version of the rectangular pulse function

```
function x = rect_loop(t)
%  RECT x = rect_loop(t): A function that is defined to
%        be 1 on [-0.5,0.5] and 0 otherwise. This imple-
%       mentation uses a for loop which is inefficient.

% Initialize a vector with zeros in it that is
% the same length as the input vetor t:
x = zeros(size(t));
for k=1:length(t)
   if abs(t(k)) <= 0.5
      x(k) = 1;
   end
end
```

- Compare execution times for the two functions using `tic` and `toc`, making sure to run both functions at least twice so that the routines may be converted to fast binary form in the MATLAB workspace (the machine is a P120 running Win95)

```
>> t = -10:.001:10; %Second pass results are below
>> length(t)
ans =       20001
>> tic; x_vec = rect(t); toc;
elapsed_time =     0.06000000000000 % Time in seconds
>> tic; x_loop = rect_loop(t); toc;
elapsed_time =  1.32000000000000 % Time in seconds
```

– <u>Note</u>: Once the two functions are 'compiled' to a fast binary form, the vector version of `rect` runs 1.32/0.06 = 22 times faster than the loop version!

- The `for` loop can be used in different ways depending upon how the loop indexing is defined
- In general loop control is accomplished using

  ```
  for index=expression
  ```

  where `expression` may be a scalar, vector, or a matrix
  - scalar case: Here the loop executes just once with `index` equal to the scalar value
  - vector case: This is the most common, with the vector often taking on integer values or of the form

    ```
    start_val:step_size:final_val
    ```

  - matrix case: This is very non traditional as far as high-level programming languages is concerned, but if used `index` becomes a column vector iterating over the columns of the expression matrix

The `while` Loop

- Another useful looping structure is the `while` loop, which loops *while* some condition is true
- The `while` code block is of the form

  ```
  while expression
     loop_statements
  end
  ```

Example: Finding $N = 2^{\nu} \geq x > 0$, that is the nearest power of two is greater than or equal to some positive number.

```
function N = pow2up(x)
% POW2UP N = pow2up(x) finds the power of 2
%             that is equal to or greater than x.
N = 1;
while x > N
   N = 2*N;
end
```

- Testing the function:

```
>> [pow2up(1) pow2up(15) pow2up(56) pow2up(1026)]
ans =
           1          16          64        2048
```

- In most high-level languages loops must be used in order to solve problems
- In writing MATLAB programs we strive to eliminate loops if at all possible
- Sometimes loops cannot be avoided