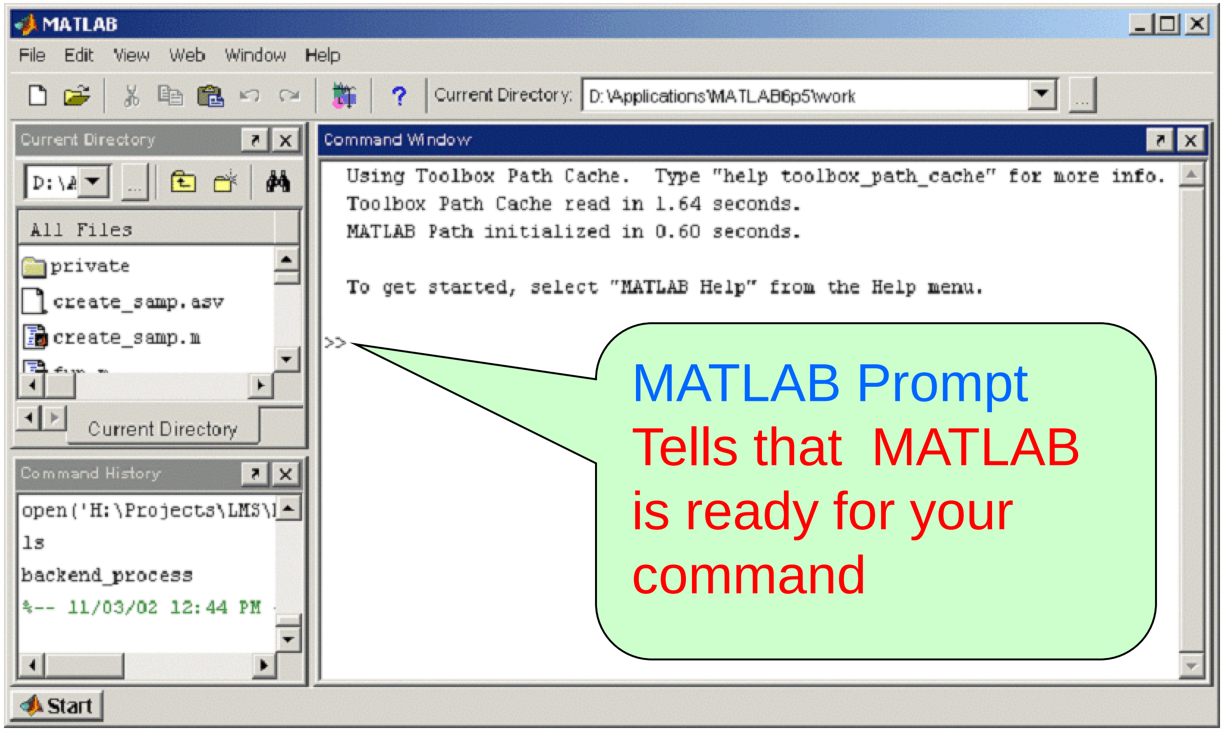
# Introduction to MATLAB

# MATLAB

- MATLAB==MATrix LABaratory
- It is widely used to solve different types of scientific problems.
- The basic data structure is a complex double precision matrix.

# Run MATLAB

Type ‘matlab’ in a shell



Current Directory: D:\Applications\MATLAB6p5\work

```
private
create_samp.asv
create_samp.m
fun.m
```

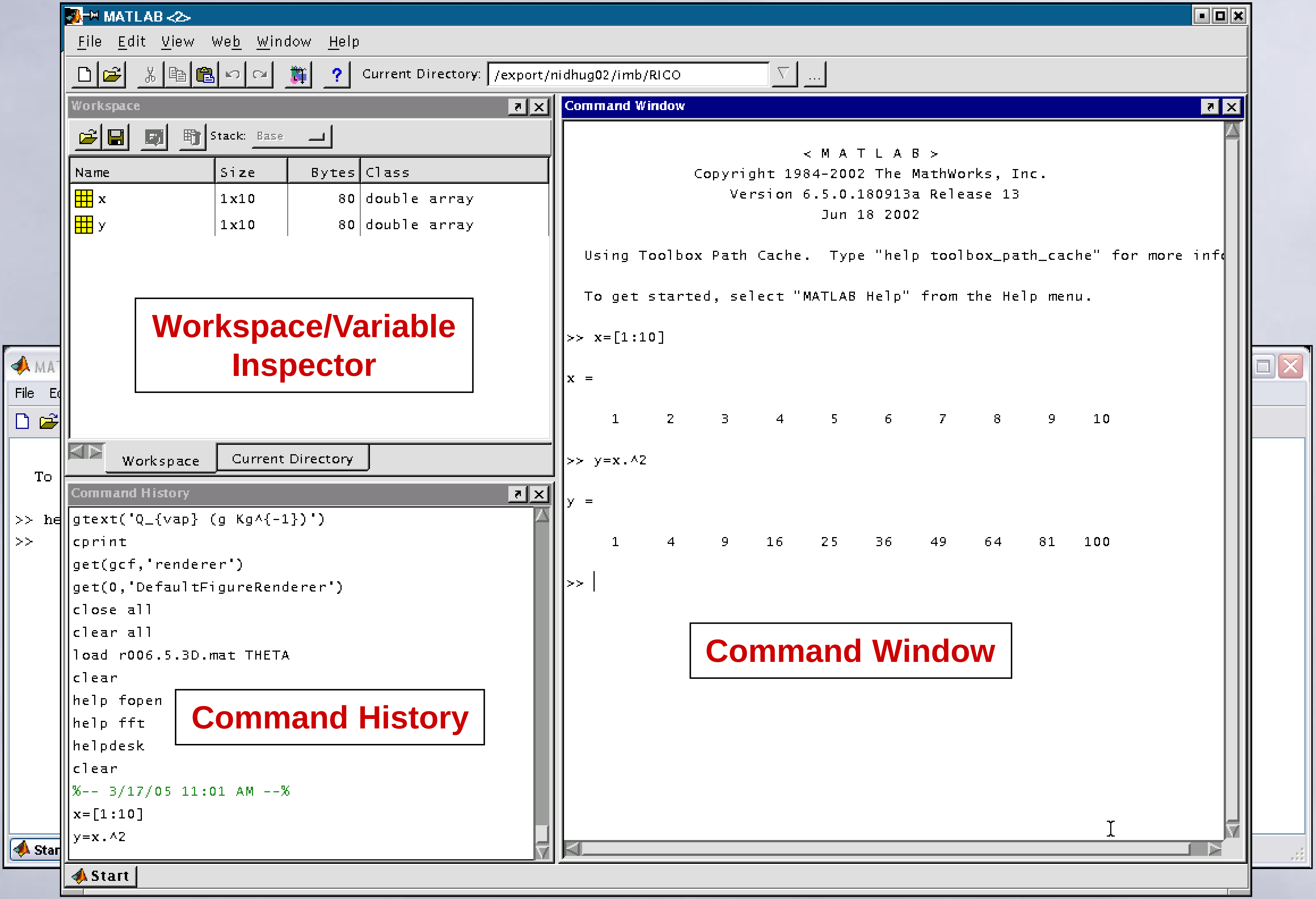
```
Using Toolbox Path Cache.  Type "help toolbox_path_cache" for more info.
Toolbox Path Cache read in 1.64 seconds.
MATLAB Path initialized in 0.60 seconds.

To get started, select "MATLAB Help" from the Help menu.

>>
```

```
open('H:\Projects\LMS\
ls
backend_process
%-- 11/03/02 12:44 PM
```

MATLAB Prompt

Tells that MATLAB is ready for your command

# MATLAB User Environment



Workspace/Variable Inspector

| Name | Size | Bytes | Class |
|---|---|---|---|
| x | 1x10 | 80 | double array |
| y | 1x10 | 80 | double array |

Command History

```
gtext('Q_{vap} (g Kg^{-1})')
cprint
get(gcf,'renderer')
get(0,'DefaultFigureRenderer')
close all
clear all
load r006.5.3D.mat THETA
clear
help fopen
help fft
helpdesk
clear
%-- 3/17/05 11:01 AM --%
x=[1:10]
y=x.^2
```

Command Window

```
                      < M A T L A B >
          Copyright 1984-2002 The MathWorks, Inc.
              Version 6.5.0.180913a Release 13
                        Jun 18 2002


  Using Toolbox Path Cache.  Type "help toolbox_path_cache" for more inf

  To get started, select "MATLAB Help" from the Help menu.

>> x=[1:10]

x =

     1     2     3     4     5     6     7     8     9    10

>> y=x.^2

y =

     1     4     9    16    25    36    49    64    81   100

>> 
```

# Getting help

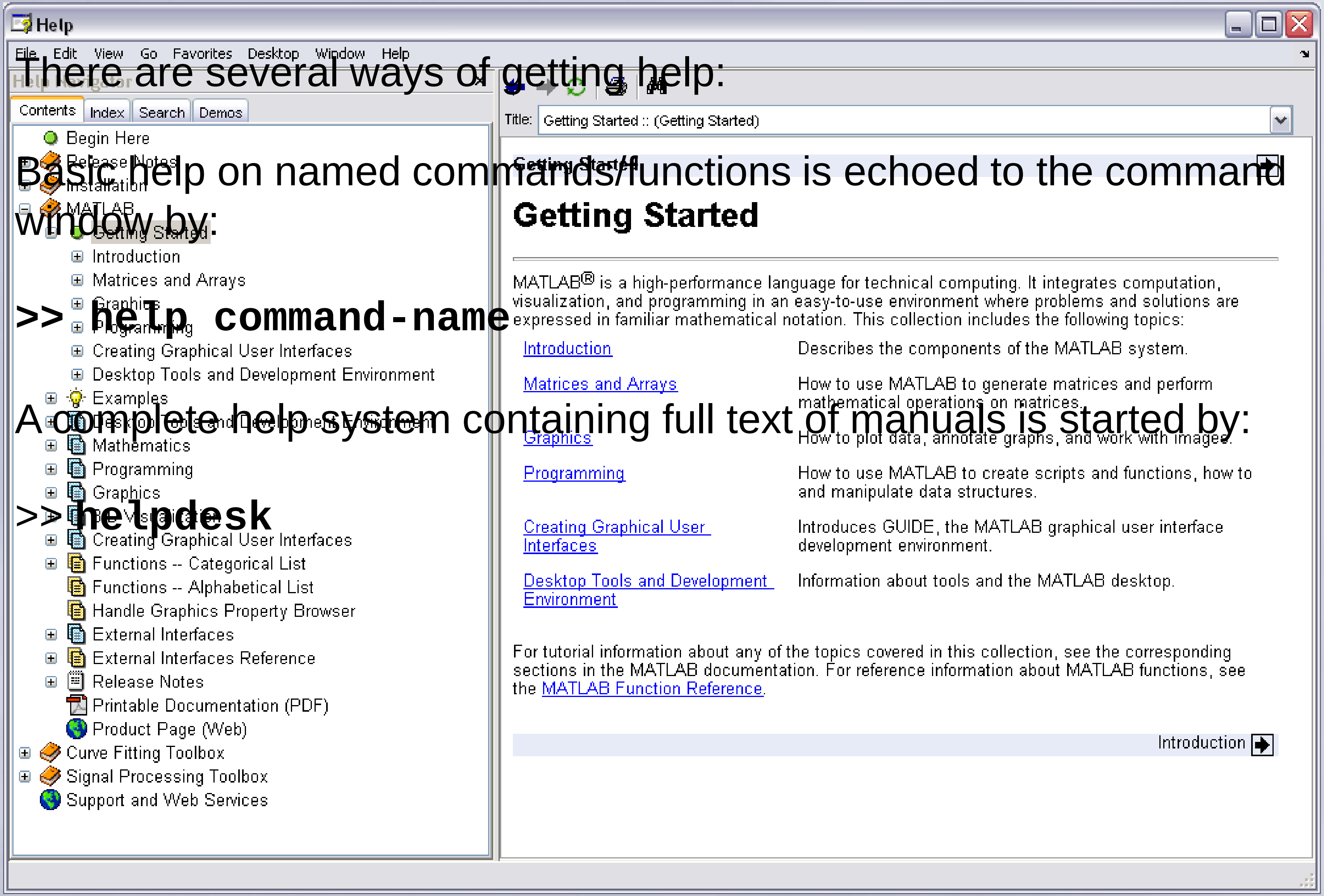
There are several ways of getting help:

Basic help on named commands/functions is echoed to the command window by:

```
>> help command-name
```

A complete help system containing full text of manuals is started by:

```
>> helpdesk
```


Help
File Edit View Go Favorites Desktop Window Help
Help Navigator
Contents
Index
Search
Demos
Begin Here
Release Notes
Installation
MATLAB
Getting Started
Introduction
Matrices and Arrays
Graphics
Programming
Creating Graphical User Interfaces
Desktop Tools and Development Environment
Examples
Desktop Tools and Development Environment
Mathematics
Programming
Graphics
3-D Visualization
Creating Graphical User Interfaces
Functions -- Categorical List
Functions -- Alphabetical List
Handle Graphics Property Browser
External Interfaces
External Interfaces Reference
Release Notes
Printable Documentation (PDF)
Product Page (Web)
Curve Fitting Toolbox
Signal Processing Toolbox
Support and Web Services

Title: Getting Started :: (Getting Started)

Getting Started

## Getting Started

MATLAB® is a high-performance language for technical computing. It integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation. This collection includes the following topics:

| | |
|---|---|
| Introduction | Describes the components of the MATLAB system. |
| Matrices and Arrays | How to use MATLAB to generate matrices and perform mathematical operations on matrices. |
| Graphics | How to plot data, annotate graphs, and work with images. |
| Programming | How to use MATLAB to create scripts and functions, how to and manipulate data structures. |
| Creating Graphical User Interfaces | Introduces GUIDE, the MATLAB graphical user interface development environment. |
| Desktop Tools and Development Environment | Information about tools and the MATLAB desktop. |

For tutorial information about any of the topics covered in this collection, see the corresponding sections in the MATLAB documentation. For reference information about MATLAB functions, see the MATLAB Function Reference.

Introduction

# The WORKSPACE

- MATLAB maintains an active **workspace**, any variables (data) loaded or defined here are always available.
- Some commands to examine workspace, move around, etc:

**who** : lists variables in workspace

```
>> who

Your variables are:

x  y
```

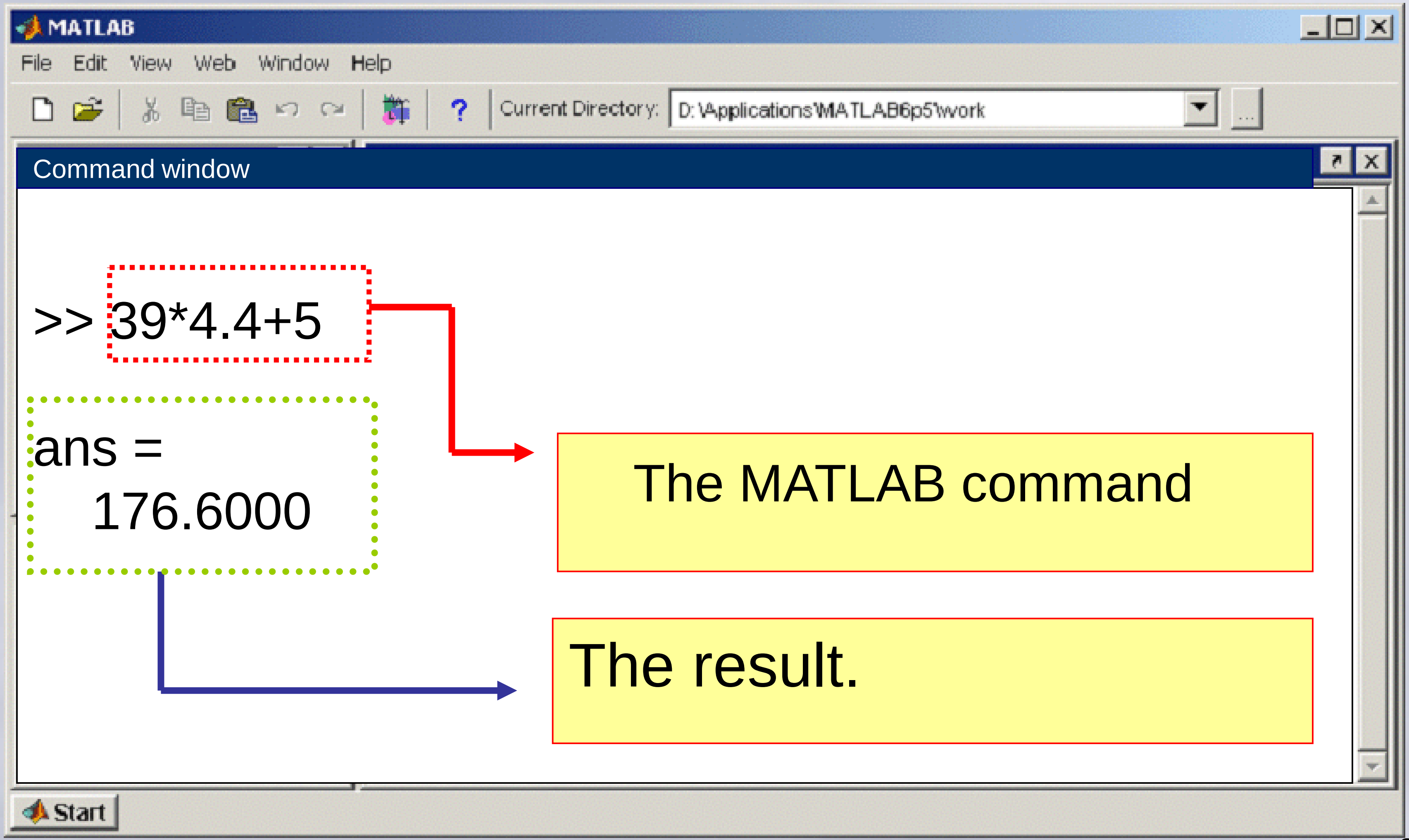
**whos** : lists names and basic properties of variables in the workspace

```
>> whos
  Name      Size                   Bytes  Class

  x         3x1                       24  double array
  y         3x2                       48  double array

Grand total is 9 elements using 72 bytes
```

**pwd, cd, dir, ls** : similar to operating system (but no option switches)

```
>> pwd
ans =

D:\

>> cd cw96\jun02
>> dir
.     30m_wtv.mat     edson2km.mat    jun02_30m_runs.mat
..    960602_sst.mat  edson_2km_bulk.mat
```

# MATLAB AS A CALCULATOR



Current Directory: D:\Applications\MATLAB6p5\work

```
>> 39*4.4+5

ans =
   176.6000
```

The MATLAB command

The result.

# MATLAB

- Variable names:
  - Starts with a letter
  - Up to 31 characters ( some use 19 or 21)
  - May contain letters, digits and underscore_
  - Case sensitive (“A” is not the same as “a”)
- Use a `;` at the end of the line to stop commands from echoing to the screen
- Use ↑ ↓keys to scroll through previously entered commands and rerun.
- … continues a command to the next line.
- `help` will tell you how to use a function and what it does. (e.g. `>> help plot` )

# VARIABLES

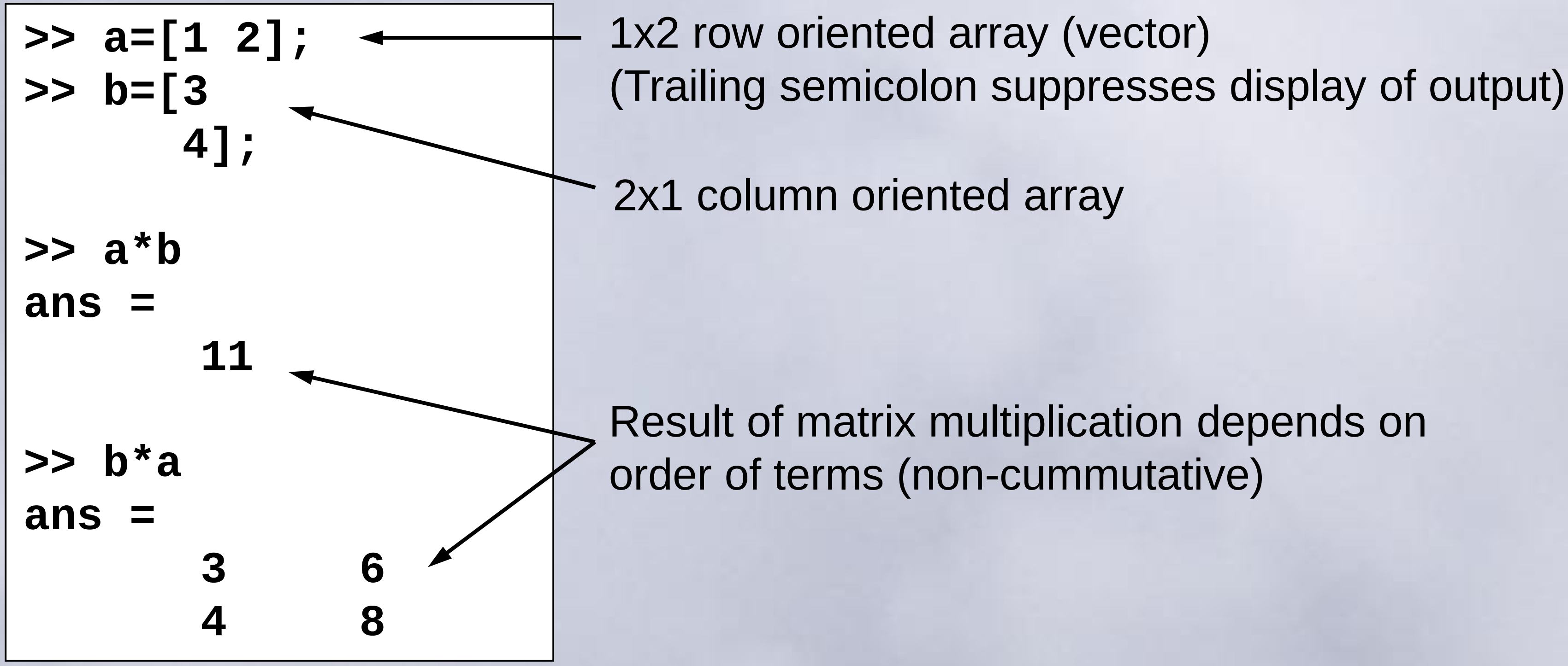
- **Everything** (almost) is treated as a **double-precision floating point array** by default
  - *Typed* variables (integer, float, char,…) are supported, but usually used only for specific applications. Not all operations are supported for all typed variables.

```
>> x=[1 2 3]
x =
     1     2     3



>> x=[1,2,3]
x =
     1     2     3

>> x=[1
      2
      3
      4];
>> x=[1;2;3;4]

x =
     1
     2
     3
     4
```

When defining variables, a **space** or **comma** separates elements on a row.

A **newline** or **semicolon** forces a new row; these 2 statements are equivalent.
NB. you can break definitions across multiple lines.

- 1 & 2D arrays are treated as formal matrices
  - Matrix algebra works by default:



```
>> a=[1 2];
>> b=[3
      4];

>> a*b
ans =
       11

>> b*a
ans =
       3     6
       4     8
```

1x2 row oriented array (vector)
(Trailing semicolon suppresses display of output)

2x1 column oriented array

Result of matrix multiplication depends on
order of terms (non-cummutative)

- Element-by-element operation is forced by preceding operator with ‘.’

```
>> a=[1 2];
>> b=[3
      4];

>> a.*b
??? Error using ==> times
Matrix dimensions must agree.
```

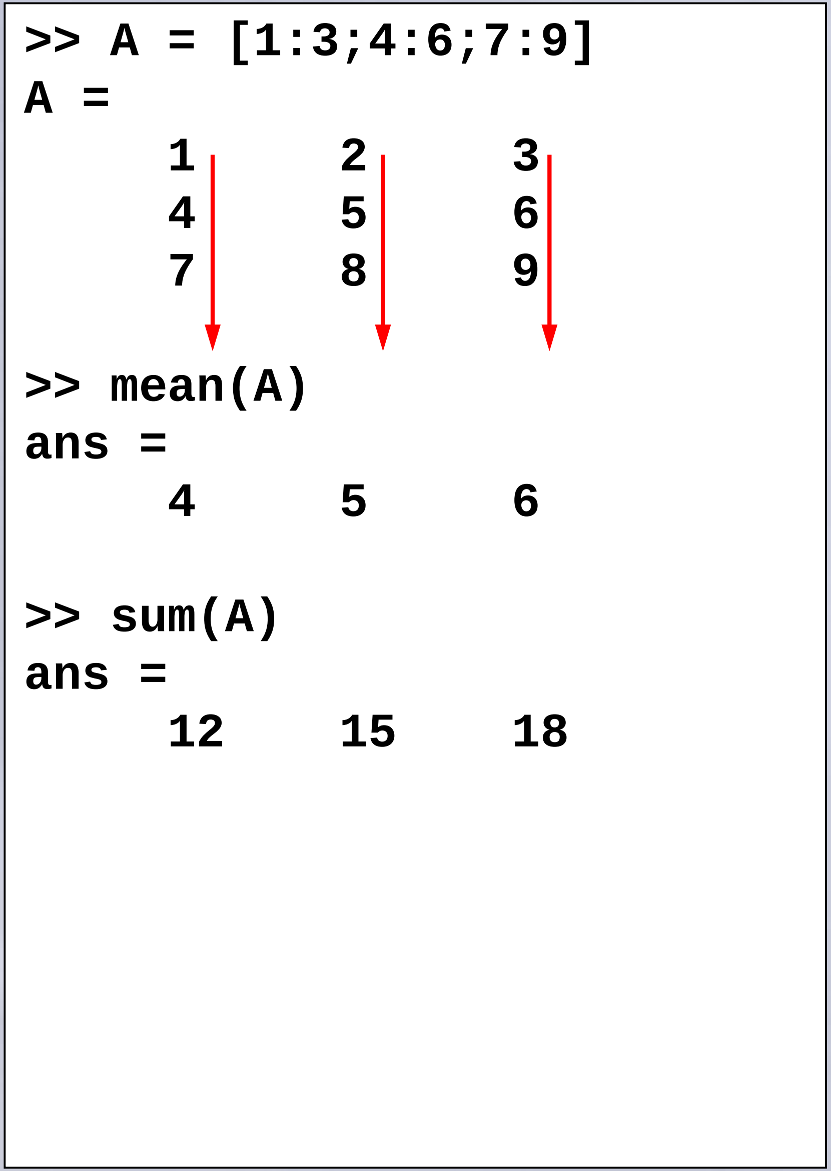
Size and shape must match

```
>> a=[1 2]
A  =
       1    2

>> b=[3 4];

>> a.*b
ans =
       3    8

>> c=a+b
c =
       4    6
```

No trailing semicolon, immediate display of result

Element-by-element multiplication

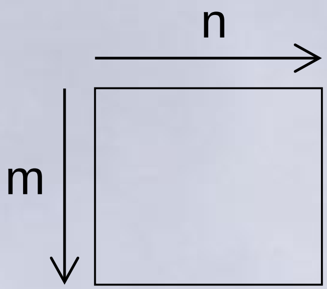
Matrix addition & subtraction operate element-by-element anyway. Dimensions of matrix must still match!



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

>> mean(A)
ans =
     4     5     6

>> sum(A)
ans =
     12    15    18
```

Most common functions operate on columns by default

# INDEXING ARRAYS



- **MATLAB indexes arrays:**
  - **1 to N**
  - **[row,column]**

```
[1,1   1,2   .   1,n
 2,1   2,2   .   2,n
 3,1   3,2   .   3,n
  .     .    .
 m,1   m,2   .   m,n]
```

- IDL indexes arrays:
  - 0 to N-1
  - [column,row]

```
[0,0   1,0   .  n-1,0
 0,1   1,1   .  n-1,1
 0,2   1,2   .  n-1,2
   .    .    .    .
0,m-1 1,m-1 . n-1,m-1]
```

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

>> A(2,3)
ans =
       6

>> A(1:3,2)
ans =
       2
       5
       8

>> A(2,:)
ans =
       4    5    6
```

The colon indicates a range, a:b (a to b)

A colon on its own indicates ALL values

# THE COLON OPERATOR

- Colon operator occurs in several forms
  - To indicate a range (as above)
  - To indicate a range with non-unit increment

```
>> N = 5:10:35
N =
      5      15     25     35
>> P = [1:3; 30:-10:10]
P =
      1      2      3
      30     20     10
```

- To extract ALL the elements of an array (extracts everything to a single column vector)

```
>> A = [1:3; 10:10:30;

        100:100:300]

A =

       1      2      3

       10     20     30

       100    200    300
```

```
>> A(:)
ans =
       1
       10
       100
       2
       20
       200
       3
       30
       300
```

# LOGICAL INDEXING

- Instead of indexing arrays directly, a logical mask can be used – an array of same size, but consisting of 1s and 0s – usually derived as result of a logical expression.

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

# Basic Operators

**`+, -, *, /`** : basic numeric operators

**`\`** : left division (matrix division)

**`^`** : raise to power

**`′`** : transpose (of matrix) – flip along diagonal

- **`fliplr(), flipud()`** : flip matrix about vertical and horizontal axes.

# *Entering Data - Scalars, Vectors*

Scalar

Note:

```
>> a = 5;
>>
```

Vector

`[ ]`

row vector

use `n′` for column vector

use set of 2 : to make a regularly spaced set of data.

```
>> a = 5

a =
      5

>> b = [1 2 3 2 1]

b =
      1 2 3 2 1

>> c = [1:0.25:2]′

c =
      1.00
      1.25
      1.50
      1.75
      2.00
```

# *Entering Data - 2D Matrices*

Matrices

; separates rows.

```
>> c = [1 2 3; 4 5 6; 7 8 9]

c =
      1 2 3
      4 5 6
      7 8 9
```

N’ switches rows and columns

```
>> c = c’

c =
      1 4 7
      2 5 8
      3 6 9
```

# *Accessing Data - 2D Arrays*

Extracting part of a matrix

`( )` specify elements of an array

`,` separates dimension (rows,columns)

Combining Arrays

horizontal concatenation

no punctuation

vertical concatenation

`;` separates arrays

```
>> d = c(1:2,2:3)

d =
      4 7
      5 8
>> e = [d d]

e =
      4 7 4 7
      5 8 5 8
>> f = [d; d]

f =
      4 7
      5 8
      4 7
      5 8
```

# *Accessing Data - 2D Arrays*

Functions for extracting data:

`min, max, mean, median, sort`

`find`

default returns indices of nonzero elements

can also use to find which indices have a specific value

```
>> x = [3 2 0 1 5 4];
>> minx = min(x)

minx =
      0
>> y = sort(x)

y =
      0 1 2 3 4 5

>> find(x)

ans =
      1 2 4 5 6
>> i4 = find(x==4)

i4 =
      6
```

# *Entering Data - 3D Matrices*

```
>> c = [1 2 3; 4 5 6; 7 8 9;...
2 2 2; 3 3 3; 4 4 4]’

c =

     1     4     7     2     3     4
     2     5     8     2     3     4
     3     6     9     2     3     4

>> size ( c)

     3     6
```

```
>> d = reshape(c, 3,3,2)

d(:,:,1) =

     1     4     7
     2     5     8
     3     6     9

d(:,:,2) =

     2     3     4
     2     3     4
     2     3     4
```

```
>> size(d)

ans =
     3   3   2
```

# *Some Predefined Variables*

| | |
|---|---|
| `pi` | 3.14159265 |
| `i,j` | sqrt(-1) imaginary unit |
| `eps` | 2.2204e-016 (small number, can also be set to a number) |
| `inf` | infinity (from 1/0) |
| `nan` | not-a-number (from 0/0) |

| | |
|---|---|
| `zeros(n,m)` | an n x m matrix of zeros |
| `ones(n,m)` | an n x m matrix of ones |
| `rand(n,m)` | an n x m matrix of uniformly distributed numbers |
| `randn(n,m)` | an n x m matrix of normally distributed numbers |

# *Operators and Matrix Operations*

| | |
|---|---|
| `+ -` | addition, subtraction |
| `* / ^` | *matrix* multiplication, division, power |
| | * [ 2 x 3] * [3 x 2] = [2 x 2] |
| | / A/B = (B'\A')'B'\A |
| | ^ z = x^y, x or y must be a scalar. |
| `\` | X = A\B is the solution to the equation A*X = B |
| `.* ./ .^` | *array* multiplication, division, power |
| | (arrays must be the same size) |
| `n’` | transpose, complex conjugate |
| `[ ]` | define matrix |
| `;` | separate rows, non-echo end of line. |
| `:` | “all elements” of row, column or between 2 numbers. |
| `( )` | specify elements of a defined vector/matrix |

# *Array versus Matrix Operations*

Array operations include an implicit loop through the entire array.

The operation is repeated for each element in the array.

The matrix operation does the linear algebra operation for the matrices.

```
>> a = [1 2 3 4 5];
>> b = a;
>>
>> c = a.*b
c =
      1   4   9   16   25


>>
>> d = a*b’
d =
      55
```

# *Some Other Useful Commands*

`clear` clears all or specified variable from memory

`keyboard` within a dot-M file gives command control to the keyboard. Type return to continue.

`dbquit` used for debugging, to exit from a dot-M file after keyboard has been used to give control to the command line

`save` save variables to a file (default is matlab format, file.mat)

# *Inputting Data From File*

| | |
|---|---|
| `load` | file of ascii data in columns<br>`>> load filename`<br>data loaded into variable `filename`<br>`>> A = load(‘filename’);`<br>data loaded into variable `A` |
| `fscanf` | formatted ascii data |
| `fread` | binary data file |
| `fopen` | open files (for `fscanf, fread`) |
| `fclose` | closes files |

# *Plotting - General Commands*

| | |
|---|---|
| `figure` | creates a new figure window |
| `subplot` | defines multiple plots on a single page |
| `title` | defines plot title text |
| `xlabel` | x axes label text |
| `ylabel, zlabel` | y or z axes label text |
| `text` | puts text at specified coordinates |
| `axis` | defines the axis limits |
| `colorbar` | creates a colorbar |
| `shading` | defines type of shading (faceted, flat, interp) |
| `set` | use to define properties of the axis or figure |
| `gca` | returns handle to current plot axis |
| `gcf` | returns handle to current figure |

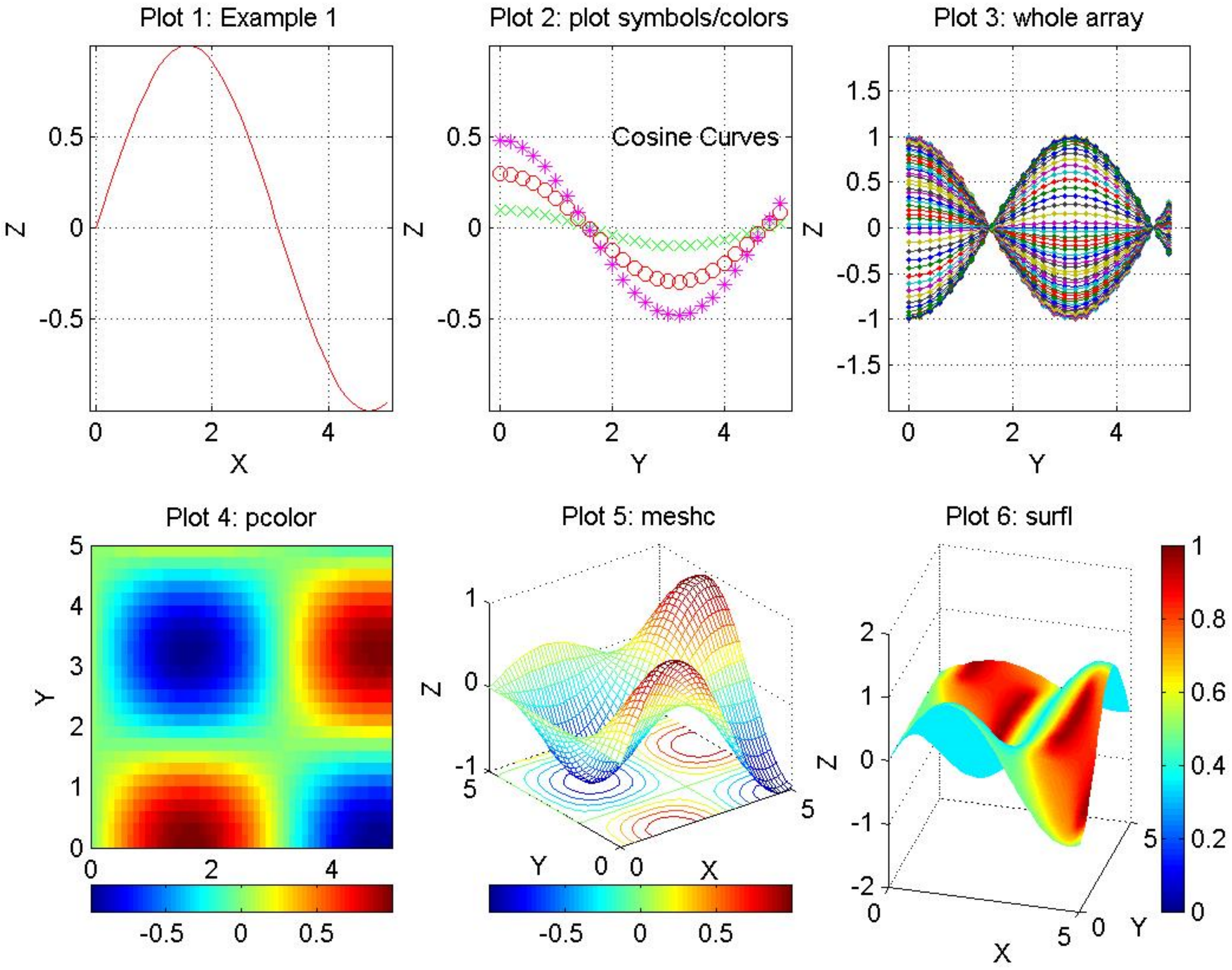
# *2D Graphics Functions*

| | |
|---|---|
| `plot` | linear plot of lines, points or symbols |
| `fill` | filled 2D polygons |
| `loglog` | log-log scale plot |
| `semilogx` | log (x-axis) - linear (y-axis) plot |
| `semilogy` | log (y-axis) - linear (x-axis) plot |
| `bar` | Linear plot with error bars |
| `errorbar` | Function plot |
| `fplot` | histogram |
| `hist` | polar coordinate plot |
| `polar` | angular histogram plot |
| `rose` | stairstep plot |
| `stairs` | |

# *3D Graphics Functions*

| | |
|---|---|
| `plot3` | Plot lines and points in 3D space |
| `fill3` | Filled 3D polygons in 3D space |
| `contour,contour3` | contour data in 2D or 3D space |
| `pcolor` | checkerboard color plot in 2D |
| `quiver, quiver3` | vector plot in 2D or 3D |
| `stream, stream3` | Stream lines in 2D or 3D |
| `mesh, meshc, meshz` | Mesh surface in 3D, with contours or zero plane. |
| `surf, surfc, surfl` | Shaded surface in 3D |
| `surfnorm` | Display surface normal vectors. |
| `cylinder, sphere` | Generate a cylinder or sphere. |

# *Example: dot-M files, functions, plotting (6)*

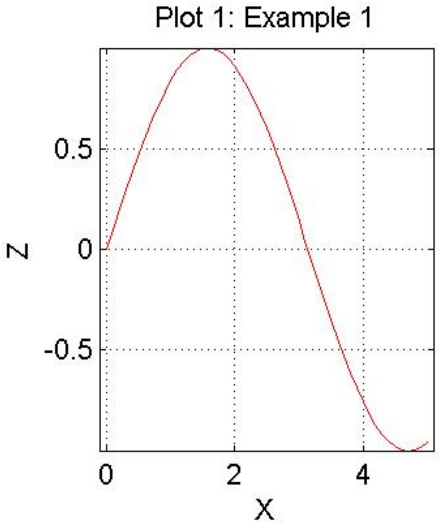


# *Example: dot-M files, functions, plotting (1)*

```
>> example1('Example 1',0.1,0,5,0.2,0,5);
% example1.m
% Comment lines start with the percent sign
% This is an example function which takes 3 parameters modelname is a string with the
% name of the model to be used in the title line. dx and dy are the intervals for the
% x and y variables, which range in value from from xmin to xmax or ymin to ymax.

function example1(modelname,dx,xmin,xmax,dy,ymin,ymax);

% Define x and y
x = [xmin:dx:xmax];
y = [ymin:dy:ymax];

% Make a mesh out of x and y
[X,Y] = meshgrid(x,y);   % takes vectors and repeats x for every y and y for every x.

% Define a function for every x, y pair
Z = sin(X).*cos(Y);
whos X Y Z
```

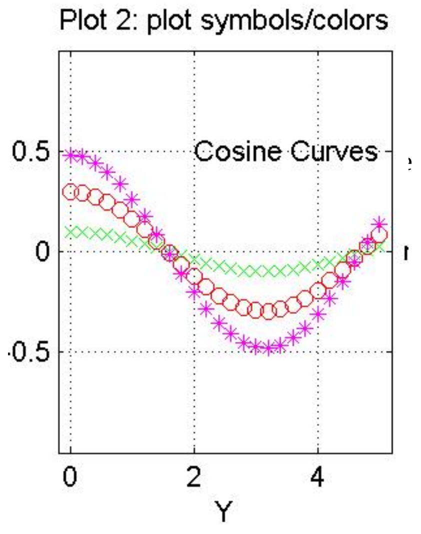
# *Example: dot-M files, functions, plotting (2)*

```
minz = min(min(Z));
maxz = max(max(Z));

save ex1dat X Y Z              % saves variables in binary matlab format

% Plotting
figure;                        % opens the figure window.
subplot(2,3,1)                 % 6 plots in figure split into 2 rows and columns
                               % 1- first plot is in top-left
plot(X(1,:),Z(1,:),'r')        % plot all the points in the
                               % first column as a red line
                               % (also b, g, y, c, m, k, w).
title(['Plot 1: ' modelname]);
xlabel('X');
ylabel('Z');
grid;                          % Plots grid lines
axis([xmin-dx xmax+dx minz maxz]);
```

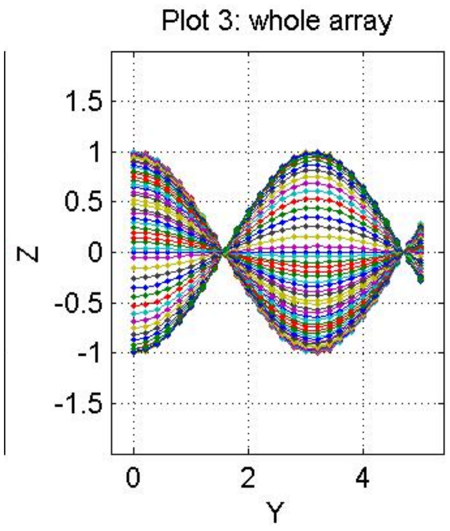


# *Example: dot-M files, functions, plotting (3)*

```
subplot(2,3,2)
plot(Y(:,2),Z(:,2),'gx');          % plots points as green x's
                                   % (type 'help plot' for full list).
vlab = input(‘Enter label: ‘,’s’);    % get data from command line
hold on;      % allows you to overlay more symbols, lines etc...
plot(Y(:,4),Z(:,4),'ro');
plot(Y(:,6),Z(:,6),'m*');
title('Plot 2: plot symbols');
text(2,0.5,vlab);                  % add text labels
xlabel('Y'); ylabel('Z'); grid;
axis([ymin-dy ymax+dy minz maxz]);
```

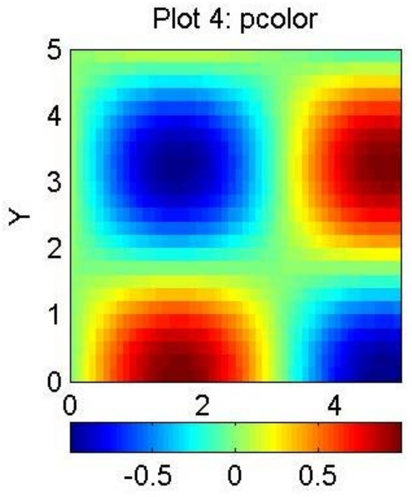


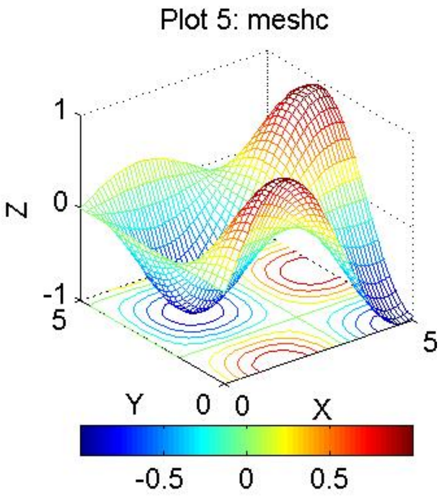
```
subplot(2,3,3)
plot(Y,Z,'.-');          % plot all rows of X versus all rows of Z
                          % -- dashed line, also -solid, : dotted, -.dash-dot
                          % can also combine symbols and line 'o-;
title('Plot 3: whole array');
xlabel('Y'); ylabel('Z'); grid;
axis([ymin-2*dy ymax+2*dy 2*minz 2*maxz]);
```

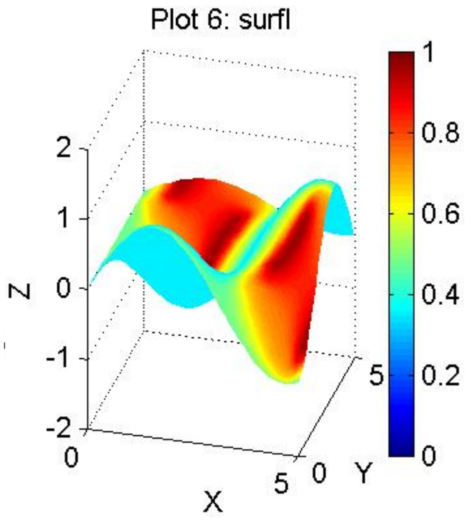


# *Example: dot-M files, functions, plotting (5)*

```
subplot(2,3,4)
pcolor(X,Y,Z);
shading flat; colorbar('horiz');
title('Plot 4: pcolor');
xlabel('X'); ylabel('Y');

subplot(2,3,5)
meshc(X,Y,Z);
shading flat; colorbar('horiz');
title('Plot 5: meshc');
xlabel('X'); ylabel('Y'); zlabel('Z')

subplot(2,3,6)
surfl(X,Y,Z);
shading interp; colorbar;
set(gca,'Zlim',[-2 2]);
view(15,20);
title('Plot 6: surfl');
xlabel('X'); ylabel('Y'); zlabel('Z')
```

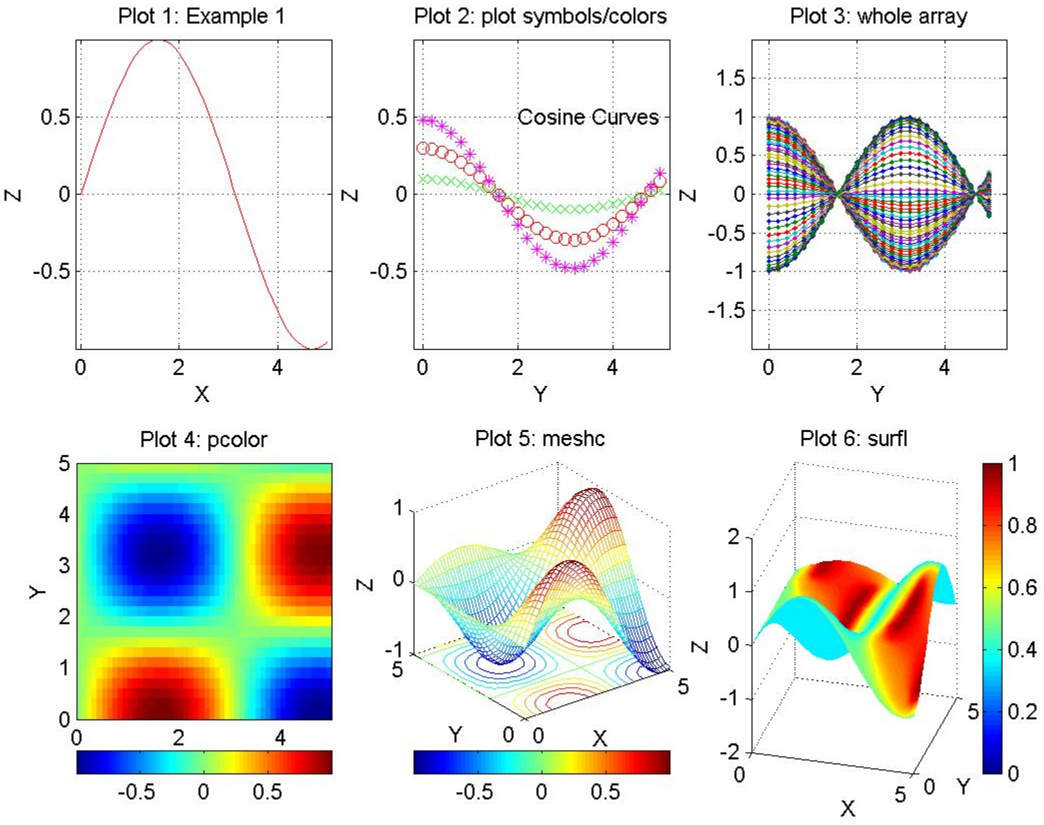






# *Example: dot-M files, functions, plotting (7)*

```
% Save data to ascii file as column data
[nx,ny] = size(X);
X = reshape(X,nx*ny,1);
Y = reshape(Y,nx*ny,1);
Z = reshape(Z,nx*ny,1);

dat = [X Y Z];
keyboard;

save ex1.asc dat -ascii
```



# strings

- MATLAB treats strings as arrays of characters
  - A 2D ‘string’ matrix must have the same number of characters on each row!

```
>> name = [‘Ian’,’Brooks’]
name =
       IanBrooks

>> name=[‘Ian';‘Brooks']
??? Error using ==> vertcat
All rows in the bracketed expression must have the same
number of columns

>> name=[‘Ian   ';‘Brooks']
Name =
       Ian
       Brooks
```

- Can concatenate strings as:

```
>> firstname = ‘Ian’;
>> secondname = ‘Brooks’;
>> fullname = [firstname,‘ ’,secondname]
fullname =
      Ian Brooks
```

- Strings defined within single quotation marks

  ⇒ quotation mark within a string needs double quoting

```
>> sillyname = 'Ian ''matlab-guru'' Brooks'
sillyname =
      Ian 'matlab-guru' Brooks
```

# Evaluating strings

- The ‘**eval**’ function takes a string argument and evaluates it as a MATLAB command line – this can be a useful way of building commands to execute without knowing in advance all of the terms to include.

```
>> filename = input(‘enter filename to save to:’,’s’);
enter filename to save to:MyData

>> eval([‘save ‘,filename])
```

First line requests input as a string from the user, this is assigned to the variable ‘filename’ – here the string ‘MyData’ has been entered

Second line evaluates/executes the string as ‘save MyData’

# Cell arrays

- Cell arrays are arrays of arbitrary data objects, as a whole they are dimensioned similar to regular arrays/matrices, but each element can hold any valid matlab data object: a single number, a matrix or array, a string, another cell array…
- They are indexed in the same manner as ordinary arrays, but with curly brackets

```
>> X{1}=[1 2 3 4];
   >> X{2}=‘some random text’
   X =
       [1x4 double]    'some random text'
```

Index an array element within a cell, by concatenating the indices:

```
>> X{1}
ans =
       1    2    3    4
>> X{1}(2)
ans =
       2
```

Cell arrays are particularly useful for storing arrays of strings, rather than arrays of characters.

```
>> drinks = {‘beer’,’whisky’,’gin’,’wine’,’water’}
drinks =
       ‘beer’    ‘whisky’    ‘gin’    ‘wine’    ‘water’
>> drinks{4}
ans =
       ‘wine’
```

- Low-level file input and output is very similar to that in 'C', but with inherent vectorization.
- Files are opened with

```
>> fid = fopen(filename)
>> fid = fopen(filename,mode)
```

- **`fid`** is a file identifier, used as a reference to the file for all subsequent read and write operations
- Files are closed with:

```
>> fclose(fid)
```

# fopen modes

| | |
|---|---|
| 'r' | Open file for reading (**default**). |
| 'w' | Open file, or create new file, for writing; discard existing contents, if any. |
| 'a' | Open file, or create new file, for writing; append data to the end of the file. |
| 'r+' | Open file for reading and writing. |
| 'w+' | Open file, or create new file, for reading and writing; discard existing contents, if any. |
| 'a+' | Open file, or create new file, for reading and writing; append data to the end of the file. |
| 'A' | Append without automatic flushing; used with tape drives. |
| 'W' | Write without automatic flushing; used with tape drives. |

```
>> fopen(filename,mode,format)
```

Opens a binary file and treats all data read or written as being of the specified format. This may be necessary to read binary files written under a different operating system.

## FORMATS

| | |
|---|---|
| 'cray' or 'c' | Cray floating point with big-endian byte ordering |
| 'ieee-be' or 'b' | IEEE floating point with big-endian byte ordering |
| 'ieee-le' or 'l' | IEEE floating point with little-endian byte ordering |
| 'ieee-be.l64' or 's' | IEEE floating point with big-endian byte ordering and 64-bit long data type |
| 'ieee-le.l64' or 'a' | IEEE floating point with little-endian byte ordering and 64-bit long data type |
| 'native' or 'n' | Numeric format of the machine on which MATLAB is running (the default) |
| 'vaxd' or 'd' | VAX D floating point and VAX ordering |
| 'vaxg' or 'g' | VAX G floating point and VAX ordering |

# Reading Binary Data

```
>> data = fread(fid)
>> data = fread(fid,count,precision,skip,format)
```

NB. All input parameters except fid are optional.

Count may take the forms:

n : read n values into a column array

inf : read to end of file, data is a column array

[m,n] : read enough data to fill a matrix of size [m,n], matrix is filled in column order, and padded with zeros if insufficient data to fill it. **m** must be a positive integer, **n** may be **inf** – read to end of file, data has m rows, and however many columns are required.

# valid precisions

| MATLAB | C or Fortran | Interpretation |
|---|---|---|
| 'schar' | 'signed char' | Signed character; 8 bits |
| 'uchar' | 'unsigned char' | Unsigned character; 8 bits |
| 'int8' | 'integer*1' | Integer; 8 bits |
| 'int16' | 'integer*2' | Integer; 16 bits |
| 'int32' | 'integer*4' | Integer; 32 bits |
| 'int64' | 'integer*8' | Integer; 64 bits |
| 'uint8' | 'integer*1' | Unsigned integer; 8 bits |
| 'uint16' | 'integer*2' | Unsigned integer; 16 bits |
| 'uint32' | 'integer*4' | Unsigned integer; 32 bits |
| 'uint64' | 'integer*8' | Unsigned integer; 64 bits |
| 'float32' | 'real*4' | Floating-point; 32 bits |
| 'float64' | 'real*8' | Floating-point; 64 bits |
| 'double' | 'real*8' | Floating-point; 64 bits |

# Writing Binary Data

```
>> count = fwrite(fid,data,precision)
>> count = fwrite(fid,data,precision,skip)
```

- **`Data` is written to the file in column order.**

# Reading & Writing Formatted Ascii Data

To write formatted data:

```
>> count = fprintf(fid,format,data,...)
```

To read formatted ascii data:

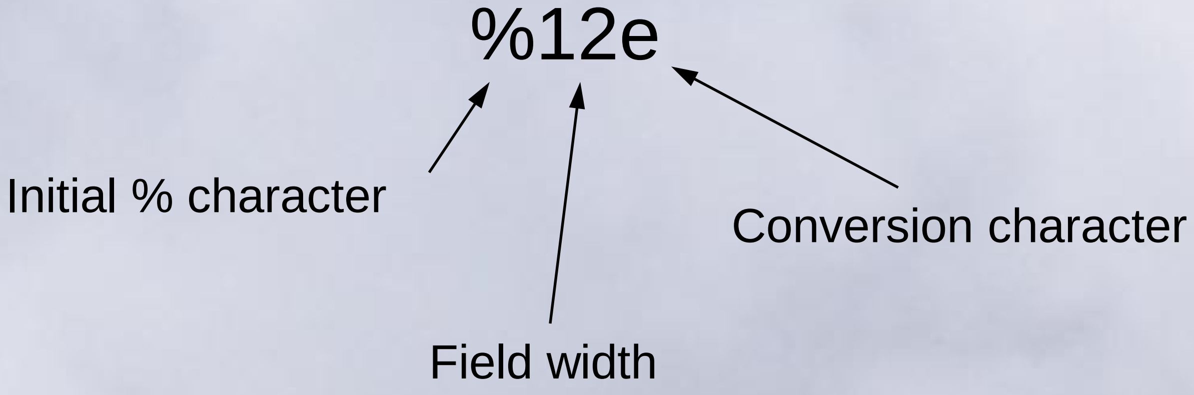
```
>> data = fscanf(fid,format)
>> [A,count] = fscanf(fid,format,size)
```

`format` is a string specifying the ascii data format, same as used in ‘C’

**fscanf** differs from its ‘C’ equivalent in that it is vectorized – multiple reads of format string carried out until end of file is reached, or matrix `size` is reached.

# Format strings

- The format string consists of ordinary characters, and/or conversion specifications indicating data type:



Add one or more of these characters between the % and the conversion character:

| | |
|---|---|
| An asterisk (*) | Skip over the matched value. If %*d, then the value that matches d is ignored and is not stored. |
| A digit string | Maximum field width. For example, %10d. |
| A letter | The size of the receiving object, for example, h for short, as in %hd for a short integer, or l for long, as in %ld for a long integer, or %lg for a double floating-point number. |

| | |
|---|---|
| %c | Sequence of characters; number specified by field width |
| %d | Base 10 integers |
| %e, %f, %g | Floating-point numbers |
| %i | Defaults to base 10 integers. Data starting with 0 is read as base 8. Data starting with 0x or 0X is read as base 16. |
| %o | Signed octal integer |
| %s | A series of non-white-space characters |
| %u | Signed decimal integer |
| %x | Signed hexadecimal integer |
| [...] | Sequence of characters (scanlist) |

# Reading whole lines

```
>> tline = fgetl(fid)
```

reads the next whole line of text from fid, returning it **without** the line termination character.

```
>> tline = fgets(fid)
```

reads the next line of text, **including** line termination character.

```
>> tline = fgets(fid,nchar)
```

reads **`nchar`** characters from the next line of fid, next read will start on next line.

# Intrinsic function - Example Polynomials

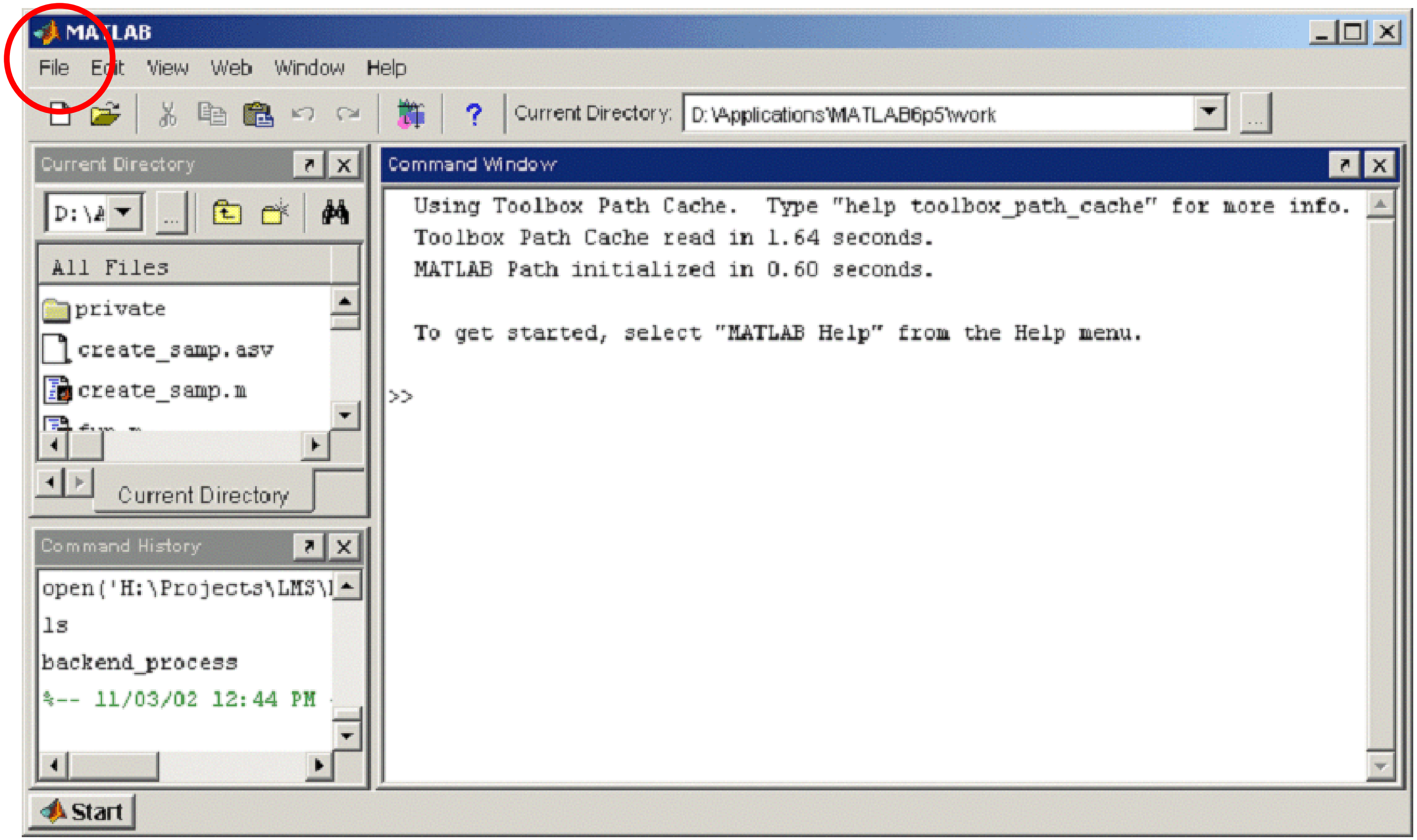
| | |
|---|---|
| roots(p) | Find the roots of a polynomial whose coefficients are given in p |
| roots([1  4  2.1]) | Find the roots of $x^2+4x+2.1=0$ |
| polyval(p,v) | Evaluate the polynomial whose coefficients are given in p at x=v |

# Scripts

- It is possible to achieve a lot simply by executing one command at a time on the command line (even possible to run loops, if…then conditional statements, etc).
- It is easier & more efficient to save extended sets of commands as a ***script*** – a simple ascii file with a ‘***.m***’ file extension, containing all the commands you wish to run. The script is run by entering its filename (without .m extension)
- In order for MATLAB to see a script it must be:
  - In the current working directory, or
  - In a directory on the current path

# Creating M-files

Select FILE → OPEN → NEW → M-files

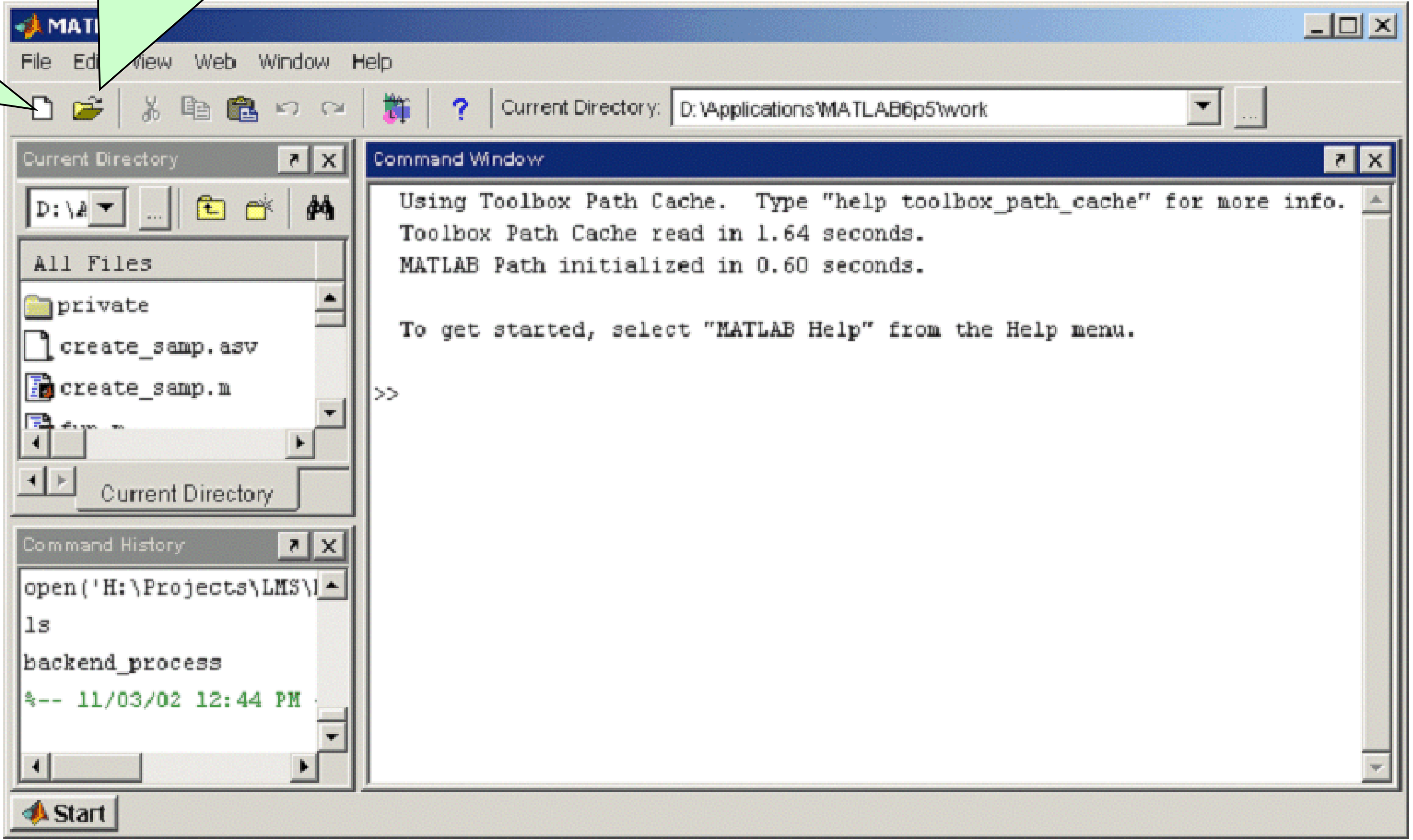


```
Using Toolbox Path Cache.  Type "help toolbox_path_cache" for more info.
Toolbox Path Cache read in 1.64 seconds.
MATLAB Path initialized in 0.60 seconds.

To get started, select "MATLAB Help" from the Help menu.

>>
```

# MATLAB shortcuts

Create a New file

Open an existing files



```
Using Toolbox Path Cache.  Type "help toolbox_path_cache" for more info.
Toolbox Path Cache read in 1.64 seconds.
MATLAB Path initialized in 0.60 seconds.

To get started, select "MATLAB Help" from the Help menu.

>>
```

```
open('H:\Projects\LMS\
ls
backend_process
%-- 11/03/02 12:44 PM
```

- Scripts are executed in the current workspace – they have access to all existing variables. New variables or changes to existing variables made by the script remain in the workspace after script has finished.

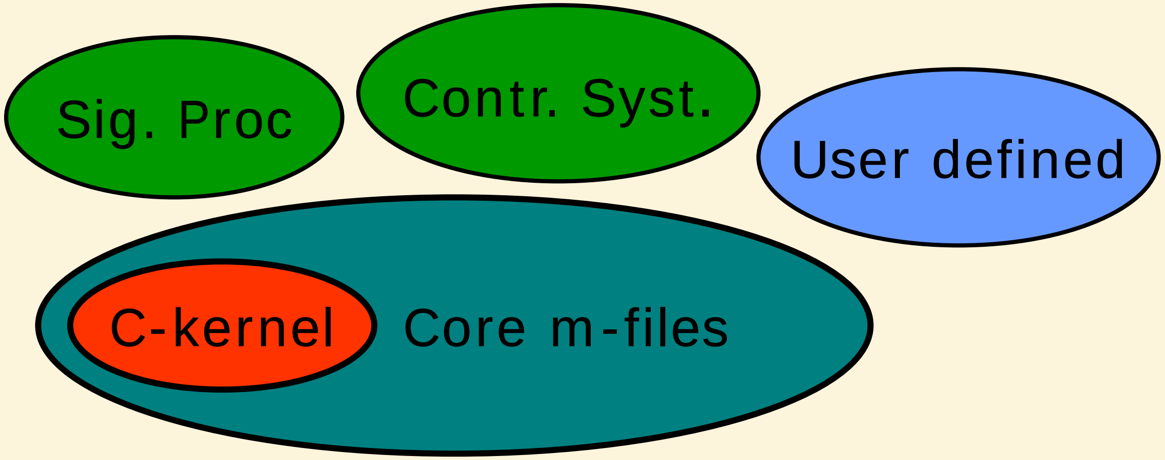
# Matlab environment

Matlab construction

Core functionality as compiled C-code, m-files

Additional functionality in toolboxes (m-files)

Today: Matlab programming (construct own m-files)



# The programming environment

The working directory is controlled by

```
>> dir
>> cd catalogue
>> pwd
```

The path variable defines where matlab searches for m-files

```
>> path
>> addpath
>> pathtool
>> which function
```

# The programming environment

Matlab can’t tell if identifier is variable or function

```
>> z=theta;
```

Matlab searches for identifier in the following order

1. variable in current workspace
2. built-in variable
3. built-in m-file
4. m-file in current directory
5. m-file on search path

Note: m-files can be located in current directory, or in path

# Script files

Script-files contain a sequence of Matlab commands

factscript.m

```
%FACTSCRIPT – Compute n-factorial, n!=1*2*...*n
y = prod(1:n);
```

- Executed by typing its name

  `>> factscript`

- Operates on variables in global workspace
  - Variable `n` must exist in workspace
  - Variable `y` is created (or over-written)
- Use comment lines (starting with %) to document file!

# Displaying code and getting help

To list code, use type command

```
>> type factscript
```

The help command displays first consecutive comment lines

```
>> help factscript
```

# Functions

Functions describe subprograms

- Take inputs, generate outputs
- Have local variables (invisible in global workspace)

```
[output_arguments]= function_name(input_arguments)
% Comment lines
<function body>
```

**factfun.m**

```
function [z]=factfun(n)
% FACTFUN – Compute factorial
% Z=FACTFUN(N)

z = prod(1:n);
```

```
>> y=factfun(10);
```

# Scripts or function: when use what?

Functions

- Take inputs, generate outputs, have internal variables
- Solve general problem for arbitrary parameters

Scripts

- Operate on global workspace
- Document work, design experiment or test
- Solve a very specific problem once

Exam: all problems will require you to write functions

facttest.m

```
% FACTTEST – Test factfun

N=50;
y=factfun(N);
```

# Flow control - selection

The if-elseif-else construction

```
if <logical expression>
   <commands>
elseif <logical expression>
   <commands>
else
   <commands>
end
```

```
if height>170
   disp(’tall’)
elseif height<150
   disp(’small’)
else
   disp(’average’)
end
```

# Logical expressions

Relational operators (compare arrays of same sizes)

| | | | |
|---|---|---|---|
| == | (equal to) | ~= | (not equal) |
| < | (less than) | <= | (less than or equal to) |
| > | (greater than) | >= | (greater than or equal to) |

Logical operators (combinations of relational operators)

| | |
|---|---|
| & | (and) |
| \| | (or) |
| ~ | (not) |

Logical functions

- xor
- isempty
- any
- all

```
if (x>=0) & (x<=10)
  disp(‘x is in range [0,10]’)
else
  disp(‘x is out of range’)
end
```

# Flow control - repetition

Repeats a code segment a fixed number of times

```
for index=<vector>
   <statements>
end
```

The <statements> are executed repeatedly.
At each iteration, the variable `index` is assigned a new value from `<vector>`.

```
for k=1:12
   kfac=prod(1:k);
   disp([num2str(k),’ ‘,num2str(kfac)])
end
```

# Example – selection and repetition

fact.m

```
function y=fact(n)
% FACT – Display factorials of integers 1..n
if nargin < 1
   error(’No input argument assigned’)
elseif n < 0
   error(’Input must be non-negative’)
elseif abs(n-round(n)) > eps
   error(’Input must be an integer’)
end

for k=1:n
   kfac=prod(1:k);
   disp([num2str(k),’ ’,num2str(kfac)])
   y(k)=kfac;
end;
```

# Repetition: Animation demo

The function movie replays a sequence of captured frames

Construct a movie of a 360° tour around the Matlab logo

**logomovie.m**

```
% logomovie – make movie of 360 degree logo tour
logo;
no_frames=40;
dtheta=360/no_frames;
for frame = 1:no_frames,
  camorbit(dtheta,0)
  M(frame) = getframe(gcf);
end
% now display captured movie
movie(gcf,M);
```

# Flow control – conditional repetition

while-loops

```
while <logical expression>
   <statements>
end
```

`<statements>` are executed repeatedly as long as the `<logical expression>` evaluates to true

```
k=1;
while prod(1:k)~=Inf,
    k=k+1;
end
disp([‘Largest factorial in Matlab:’,num2str(k-1)]);
```

# Flow control – conditional repetition

Solutions to nonlinear equations

$$f(x) = 0$$

can be found using Newton's method

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

**Task**: write a function that finds a solution to

$$f(x) = e^{-x} - \sin(x)$$

Given $x_0$, iterate `maxit` times or until $|x_n - x_{n-1}| \leq \text{tol}$

# Flow control – conditional repetition

newton.m

```
function [x,n] = newton(x0,tol,maxit)
% NEWTON – Newton’s method for solving equations
% [x,n] = NEWTON(x0,tol,maxit)
x = x0; n = 0; done=0;
while ~done,
  n = n + 1;
  x_new = x - (exp(-x)-sin(x))/(-exp(-x)-cos(x));
  done=(n>=maxit) | ( abs(x_new-x)<tol );
  x=x_new;
end
```

```
>> [x,n]=newton(0,1e-3,10)
```

# Function functions

Do we need to re-write `newton.m` for every new function?

No! General purpose functions take other m-files as input.

```
>> help feval
>> [f,f_prime]=feval(’myfun’,0);
```

**myfun.m**

```
function [f,f_prime] = myfun(x)
% MYFUN– Evaluate f(x) = exp(x)-sin(x)
% and its first derivative
% [f,f_prime] = myfun(x)

f=exp(-x)-sin(x);
f_prime=-exp(-x)-cos(x);
```

# Function functions

Can update `newton.m`

newtonf.m

```
function [x,n] = newtonf(fname,x0,tol,maxit)
% NEWTON – Newton’s method for solving equations
% [x,n] = NEWTON(fname,x0,tol,maxit)
x = x0; n = 0; done=0;
while ~done,
  n = n + 1;
  [f,f_prime]=feval(fname,x);
  x_new = x – f/f_prime;
  done=(n>maxit) | ( abs(x_new-x)<tol );
  x=x_new;
end
```

```
>> [x,n]=newtonf(’myfun’,0,1e-3,10)
```

# Function functions in Matlab

Heavily used: integration, differentiation, optimization, ...

```
>> help ode45
```

Find the solution to the ordinary differential equation

$$\dot{x}_1 = x_2$$
$$\dot{x}_2 = -x_1 + 0.1(1 - x_1^2)x_2$$

**myodefun.m**

```
function x_dot = myodefun(t,x)
% MYODEFUN – Define RHS of ODE
  x_dot(1,1)=x(2);
  x_dot(2,1)=-x(1)+0.1*(1-x(1)^2)*x(2);
```

```
>> ode45(‘myodefun’,[0 10],[1;-10]);
```

# Programming tips and tricks

Programming style has huge influence on program speed!

**slow.m**

```
tic;
X=-250:0.1:250;
for ii=1:length(x)
  if x(ii)>=0,
    s(ii)=sqrt(x(ii));
  else
    s(ii)=0;
  end;
end;
toc
```

**fast.m**

```
tic
x=-250:0.1:250;
s=sqrt(x);
s(x<0)=0;
toc;
```

Loops are slow:  Replace loops by vector operations!

Memory allocation takes a lot of time: Pre-allocate memory!

Use profile to find code bottlenecks!

# continue, break, return

**continue** – forces current iteration of loop to stop and execution to resume at the start of next iteration.

**break** – forces loop to exit, and execution to resume at first line after loop.

**return** – forces current function to terminate, and control to be passed back to the calling function or keyboard.

# More examples

- MATLAB program to find the roots of

$$f(x) = 2\cos(x) - 1$$

```
% program 1 performs four iterations of

% Newton’s Method

X=.7

for i=1:4

X=X – (2*cos(X)-1)/(-2*sin(X))

end
```

Result

```
X =
   1.1111
X =
   1.0483
X =
   1.0472
X =
   1.0472
```

```
function [x1,x2,x3]=powers1to3(n)
% calculate first three powers of [1:n]
% (a trivial example function)
x1=[1:n];
x2=x1.^2;
x3=x1.^3;
```

```
>> [x1,x2,x3]=powers1to3(4)
x1 =
     1     2     3    4

x2 =
     1     4     9    16

x3 =
     1     8    27    64
```

```
>> [x1,x2]=powers1to3(4)
x1 =
     1     2     3    4

x2 =
     1     4     9    16
```

If fewer output parameters used than are declared, only those used are returned.

# Examples for functions

- Write a function file to compute the factorial of a number.
- Input: N
- Output :NF
- Function name: factorial

# A solution

output

Function name

input

First statement must start with ‘function’

```
function  [FC]=factorial(N)
FC=1;
    for  i=1:N
        FC=FC*i;
    end
```

Save the program using  ‘factorial’ as a name

# Creating function file

Open an m-file and start typing the file

```
function  [FC]=factorial(N)
FC=1;
   for  i=1:N
      FC=FC*i;
   end
```

- Save the program using  ‘factorial’ as a name
- If  NOTEPAD is used to create the file use the name ‘factorial.m’
- Save it in directory recognized by MATLAB
- If the directory is not recognized by MATLAB add it to the MATLAB path

# A Better one

```
function  [FC]=factorial(N)
% [FC]=factorial(N)
% program to calculate the factorial of a number
% input  N : an integer
% if N is not an integer the program obtains the
% factorial of the integer part of N
% output FC : the factorial of N

%
FC=1;                              % initial value of FC
   for  i=1:N
     FC=FC*i;                     % n! =(n-1)!*n
   end
```

These comments will be displayed when

'help factorial'

is typed

Comments are used to explain MATLAB statements