Entorno de desarrollo

Start and End MATLAB Session

- Starting MATLAB
  - Double click on the MATLAB icon
  - After startup, MATLAB displays a *command window* (>>) for entering commands and display text only results.
  - MATLAB responds to commands by printing text in the command window, or by opening a *figure window* for graphical output
- Moving between windows
  - Toggle between windows by clicking on them with mouse
- Ending MATLAB
  - Type “quit” at the command prompt (>>)

    ```
    >> quit
    ```
  - Click on the window toggle (x)

MATLAB Desktop

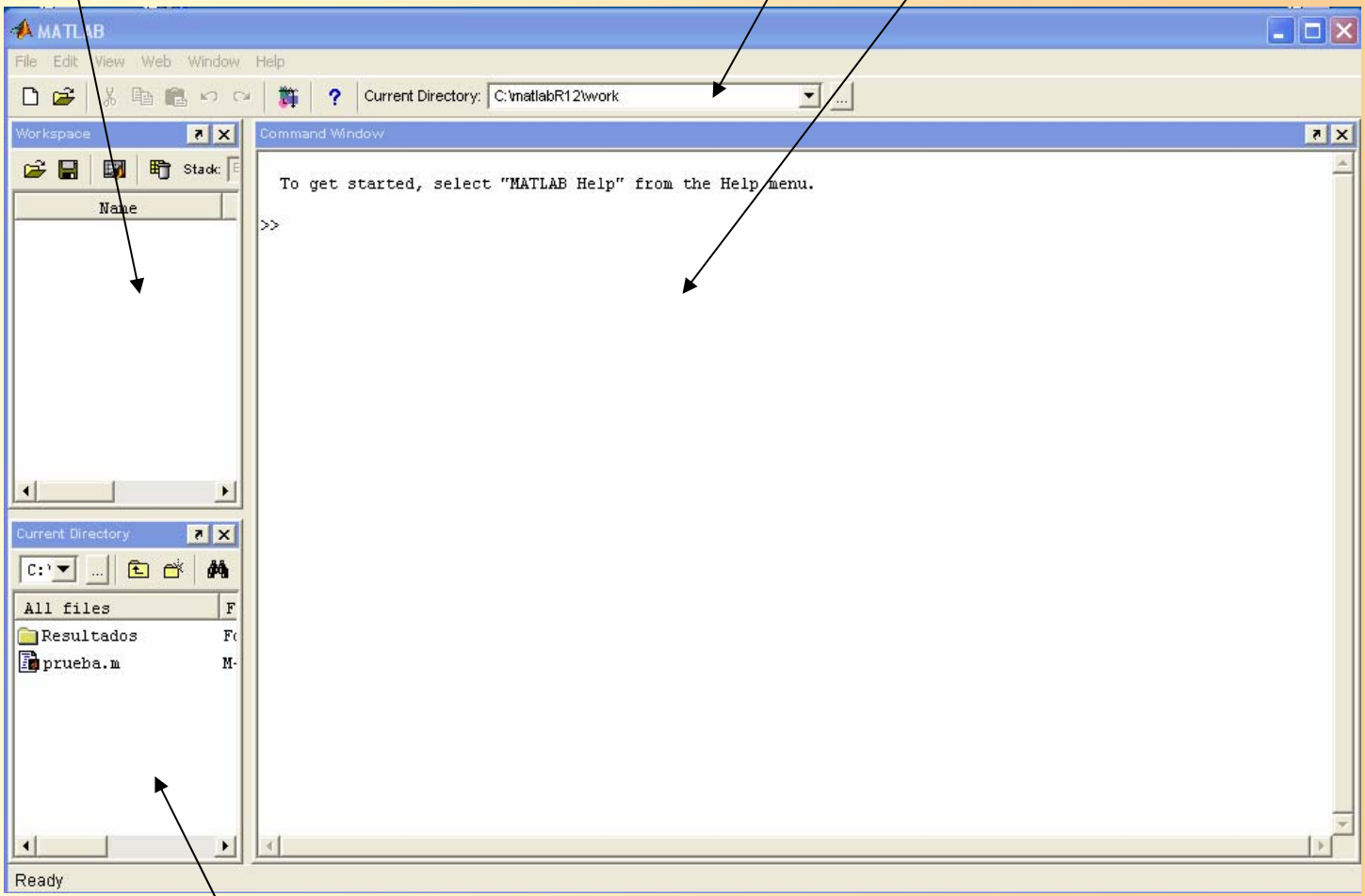
Launch Pad/workspace

Ver documentación / variables en workspace

Directorio de trabajo actual

Command Window

Introducir comandos



Current directory/Command History

Cambiar directorios /ver comandos tecleados recientemente

Basic Operations

- Enter formula at the command prompt

```
>> 3 + 4 - 1
ans =
     6
>> ans/3
ans =
     2
```

- Define and use variables

```
>> a = 6;
>> b = 7
b =
     7
>> c = a/b
c =
   0.8571
```

Note: Results of intermediate steps can be suppressed with semicolon

Built-in Variables and Functions

- **pi** (= π) and **ans** are a built-in variable

```
>> pi
ans =
   3.1416
>> sin(ans/4)
ans =
   0.7071
```

Note: There is no “degrees” mode. All angles are measured in radians.

- Many standard mathematical functions, such as **sin**, **cos**, **log**, and **log10**, are built in

```
> log(10)
ans =
   2.3026
>> log10(10)
ans =
    1
```

Note: log represents a natural logarithmic.

MATLAB Workspace

- All variables defined as the result of entering statements in the command window, exist in the MATLAB workspace

```
>> who
Your variables are:
a    ans  b    c
```

- Being aware of the workspace allows you to
  - Create, assign and delete variables
  - Load data from external files
  - Manipulate the MATLAB path
- The **whos** command lists the name, size, memory allocation and the class of each variables defined in the workspace

```
>> whos
  Name      Size            Bytes  Class

  a         1x1                 8  double array
  ans       1x1                 8  double array
  b         1x1                 8  double array
  c         1x1                 8  double array

Grand total is 4 elements using 32 bytes
```

Built-in variable **classes** are **double**, **char**, **sparse**, **struct**, and **cell**
The class of a variable determines the type of data that can be stored

On-line Help

- Use on-line help to request info on a specific function
- Use lookfor to find functions by keywords

- Syntax

  help *functionName*

  lookfor *functionName*

- Examples

```
>> help log10

 LOG10  Common (base 10) logarithm.
    LOG10(X) is the base 10 logarithm of the elements of X.
    Complex results are produced if X is not positive.

    See also LOG, LOG2, EXP, LOGM.

>> lookfor logarithmic

LOGSPACE Logarithmically spaced vector.
LOGSIG Logarithmic sigmoid transfer function.
```

Introduciendo datos

Notations

- Subscript notation
  - If A is a matrix, A(*i,j*) selects the element in the *i*-th row and *j*-th column
  - The subscript notation can be used on the right hand side (or left hand side) of expression to refer to (or assign to) a matrix element
- Colon notation
  - Colon notation is very powerful and very important in the effective use of MATLAB. The colon is used as an operator and as a wildcard
    - Create vector
    - Refer to (or extract) ranges of matrix elements
  - Syntax:

    Startvalue:*endvalue*

    Startvalue:*increment*:*endvalue*

Type of Variables

- MATLAB variables are created with an assignment statement

  >> x = *expression*

  Where *expression* is a legal combination of numerical values, mathematical operators, variables and function calls that evaluates to a matrix, vector or scalar
- Matrix
  - A two or *n* dimensional array of values
  - The elements can be numeric values (real or complex) or characters (must be defined first when executed)
- Vector
  - A one dimensional array of values
  - A matrix with *one row* or *one column*
- Scalar
  - A single value
  - A matrix with *one row* and *one column*

Matrices and Vectors

- Manual Entry
  - The elements in a vector (or matrix) are enclosed in square brackets.
    - When creating a row vector, separate elements with a space.
    - When creating a column vector, separate elements with a semicolon

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

Multiple statements per line

- Use commas or semicolons to enter more than one statement at once. Commas allow multiple statements per line without suppressing output

```
>> a = 1; b = [8 9], c = b'
b =
    8    9
c =
    8
    9
```

Examples of subscript and colon notations

```
>> a = [4 5 6;7 8 9;2 3 4];
>> b = a(3,2)
b =
    3
```

Note: Referring to an element on the third row and second column.

```
>> c = a(3,4)
???  Index exceeds matrix dimensions.
```

Note: Referring to elements outside of current matrix dimensions results in an error.

```
>> d = a(1:3,3)
d =
    6
    9
    4
```

Note: Referring to elements on the first 3 rows and third column.

Strings

- Strings are matrices with character elements
- String constant are enclosed in single quotes
- Colon notation and subscript operation apply

```
>> first = 'Yudi';
>> last = 'Samyudia';
>> name = [first,' ',last]
name =
          Yudi Samyudia
>> length(name)
ans =
    13
>> name(4:6)
ans =
i S
```

Manipulando matrices

Working with Matrices and Vectors

- **Addition and subtraction**

```
>> a = [2 3 4];
>> b = [2 1 2];
>> c = a-b
c =
    0     2     2
>> d = a+b
d =
    4     4     6
```

- **Polynomials**
  - MATLAB polynomials are stored as vectors of coefficients. The polynomial coefficients are stored in decreasing powers of *x*
  - *Example:* $y = x^3 - 2x + 12$. We want to know *y(1.5)*

```
>> y = [1 0 -2 12];
>> polyval(y,1.5)
ans =
   12.3750
```

- **Array Operators**
  - Array operators support element by element operations that are not defined by the rules of linear algebra
  - Array operators are designated by a period pre-pended to the standard operator

Symbol	Operation
.*	element by element multiplication
./	element by element “right” division
.	element by element “left” division
.^	element by element exponentiation

  - Array operators are a very important tool for writing vectorized code

Examples of using array operators

```
>> a = [1 2 3];
>> b = [6 7 8];
>> c = a.*b

c =

    6    14    24

>> c = a./b

c =

   0.1667    0.2857    0.3750

>> d = a.\b

d =

   6.0000    3.5000    2.6667
```

Equation Solving

Function	Purpose
\	Use \ (left division) to solve linear equations. See the Arithmetic Operators reference page.
fsolve	Nonlinear equation solving
fzero	Scalar nonlinear equation solving

Initial Value ODE Problem Solvers

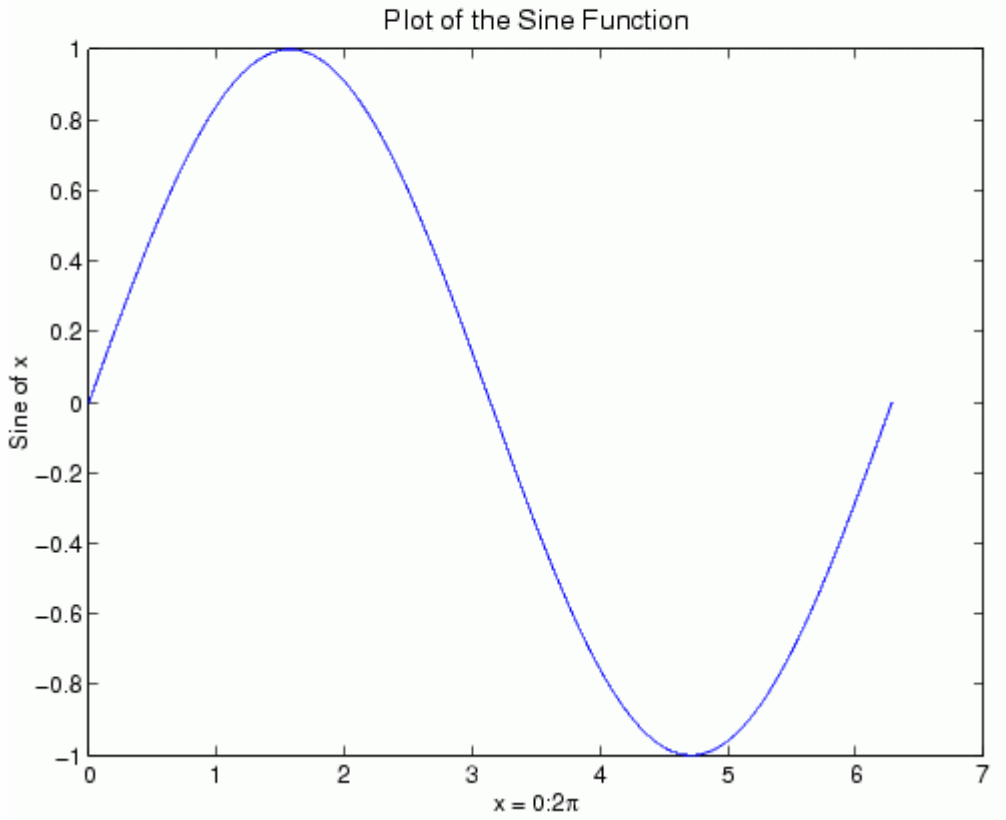
Solver	Solves These Kinds of Problems	Method
ode45	Nonstiff differential equations	Runge-Kutta
ode23	Nonstiff differential equations	Runge-Kutta
ode113	Nonstiff differential equations	Adams
ode15s	Stiff differential equations and DAEs	NDFs (BDFs)
ode23s	Stiff differential equations	Rosenbrock
ode23t	Moderately stiff differential equations and DAEs	Trapezoidal rule
ode23tb	Stiff differential equations	TR-BDF2

Haciendo Gráficos

Plotting

- Plotting (x,y) data
  >> plot(x,y)
  >> plot(xdata,ydata,symbol)
  >> plot(x1,y1,symbol1,x2,y2,symbol2,…)

```
xlabel('x = 0:2\pi')
ylabel('Sine of x')
title('Plot of the Sine Function','FontSize',12)
```

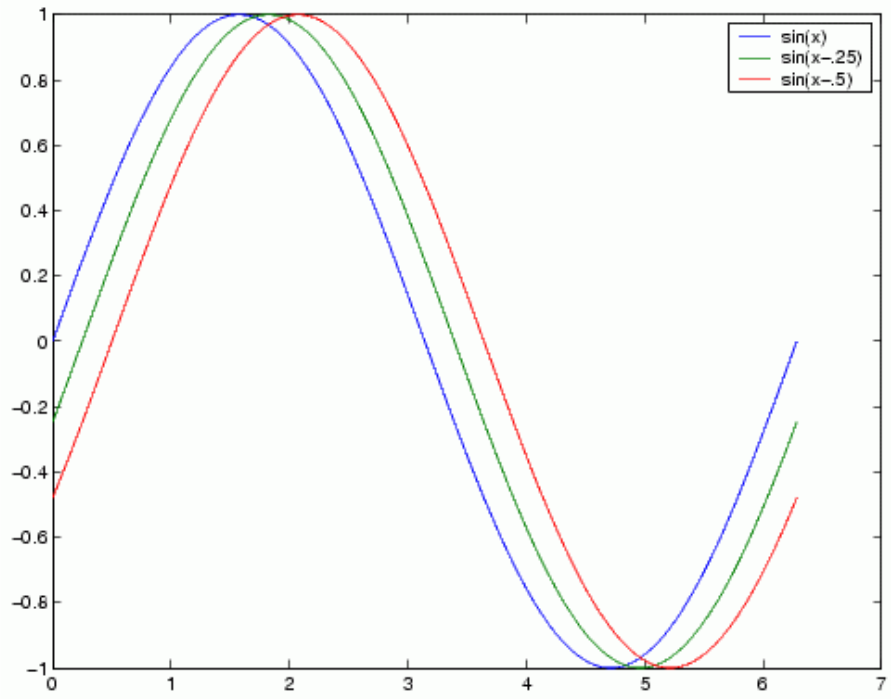


- Axis scaling and annotation
  - >> loglog(x,y) log10(y) versus log10(x)
  - >> plot(x,y) linear y versus linear x
  - >> semilogx(x,y) linear y versus log10(x)
  - >> semilogy(x,y) log10(y) versus linear x

```
y2 = sin(x-.25);
y3 = sin(x-.5);
plot(x,y,x,y2,x,y3)
```

The legend command provides an easy way to identify the individual plots.

```
legend('sin(x)','sin(x-.25)','sin(x-.5)')
```



- Multiple plot

  >> subplot(2,2,1) two rows, two column, this figure

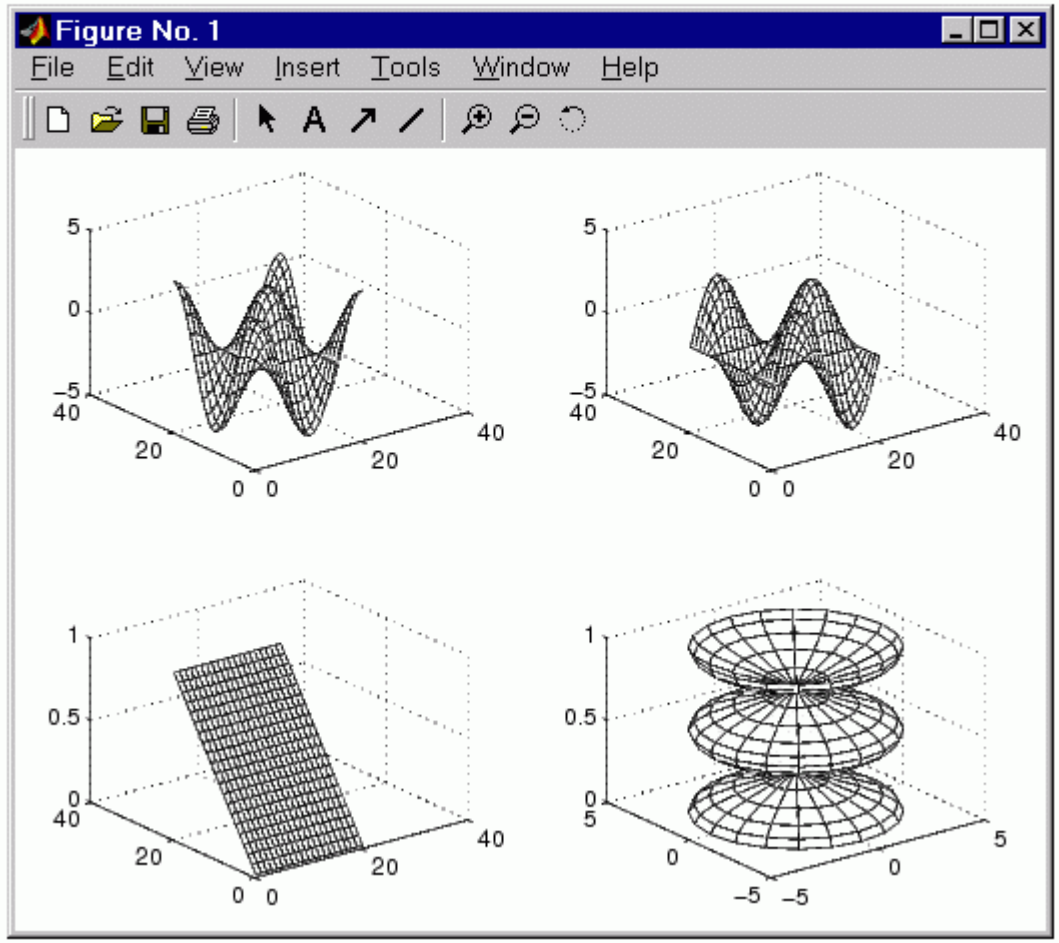
- 2D (contour) and 3D (surface) plotting

  >> contour

  >> plot3

  >> mesh

```
t = 0:pi/10:2*pi;
[X,Y,Z] = cylinder(4*cos(t));
subplot(2,2,1); mesh(X)
subplot(2,2,2); mesh(Y)
subplot(2,2,3); mesh(Z)
subplot(2,2,4); mesh(X,Y,Z)
```



Programando en Matlab

Preliminaries

- M-files are files that contain MATLAB programs
  - Plain text files
  - File must have “.m” extension
  - Use MATLAB editor (**File**, **Open/New**, **M-File**)
- Executing M-files
  - M-files must be in the current active MATLAB path
    - Use **pwd** to check the current active MATLAB path
    - Manually modify the path: **path**, **addpath**, **rmpath**, or **addpwd**
    - ….or use interactive **Path Browser**
  - A program can exist, and be free of errors, but it will not run if MATLAB cannot find it

MATLAB Script M-Files

- Collection of executed MATLAB commands
  - Not really a program
  - Useful for tasks that never change
  - Script variables are part of workspace
  - Useful as a tool for documenting assignments
    - Use **a script M-file** to run function for *specific parameters* required by the assignment
    - Use **a function M-file** to solve the problem for *arbitrary parameters*

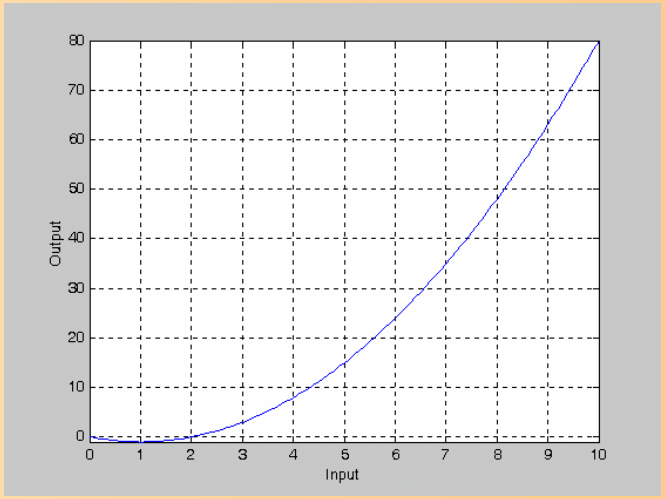
Tips:

- As **a script M-file** is a collection of executed MATLAB commands, no advantages over the use of script, except for “documentation”.
- The main program is often implemented using **a script M-file**
- Always use **a function M-file** when dealing with the possible changes in parameters/inputs

Development of a Script M-file

- Choose **New**… from **File** menu
- Enter the sequence of command lines
  - *Example*: Plotting a quadratic function (exp1.m)

```
x = [0:.1:10];
y = x.^2 - 2*x;
plot(x,y);
xlabel('Input');
ylabel('Output');
grid on;
axis([min(x) max(x) min(y) max(y)]);
```



- Choose **Save** …. from the **File** menu
  - Save as exp1.m
- Run it

  >> exp1

Side Effects of Script M-Files

- All variables created in a script M-file are added to the workspace.
  - The variables already existing in the workspace may be *overwritten*
  - The execution of the script can be affected by the state variables in the workspace
- Side Effects from scripts
  - Create and change variables in the workspace
  - Give no warning that workspace variables have changed

"Because scripts have side effects, it is better to encapsulate any mildly complicated numerical in **a function M-file**"

Function M-Files

- Function M-files are subprograms
  - Functions use *input* and *output parameters* to communicate with other functions and the command window
  - Functions use *local variables* that exist only while the function is executing. Local variables are distinct from the variables of the same names in the workspace or in other functions
- Input parameters allow the same calculation procedure (algorithm) to be applied for different data.
  - Function M-files are *reusable*
- Functions can call other functions
- Specific tasks can be encapsulated into functions.
  - Enable the development of *structured solutions (programming)* to complex problems

Syntax of function m-files

- The first line of a function m-file has the form

  function [outArg] = funName(inArg)

- **outArg** are the assigned output parameters for this function
  - A comma separated list of variable names
  - [ ] is optional for only one output argument
  - Functions with no **outArg** are legal

- **inArg** are the input parameters to be used in the function
  - A comma separated list of variable names
  - Functions with no **inArg** are legal

Examples of a Function M-File

```
>> x = 1; y = 3;
>> mult(x,y)
```

Script-file as main program to assign data for the input parameters

mult.m

```
function mult(x,y),
%
%
x*y
```

```
>> x = 1; y = 3; z= 4;
>> [t,n] = kali(x,y,z)
t =
     7
n =
    13
```

Script-file as main program to assign data for the input parameters

kali.m

```
function [s,p] = kali(x,y,z)
%
s = x*y+z;
p = x+y*z;
```

Further Notes on Input and Output Parameters

- Values are communicated through input and output arguments
- Variables defined inside a function are local to that function
  - Local variables are invisible to other functions and to the command window environment
- The number of return variables should be match the number of output variables provided by the function
  - If not the same, the m-file are still working but not returning all variables in the command window
  - **nargout** can relax this requirement

Flow Control

- To enable the implementation of computer algorithm, a computer language needs control structures for
  - Comparison
  - Conditional execution: *branching*
  - Repetition: *looping or iteration*

- **Comparison**
  - Is achieved with *relational operators*. Relational operators are used to test whether two values are equal, greater than or less than another.
  - The result of a comparison may also be modified by *logical operators*

Relational Operators

- Relational operators used in MATLAB are:
  - < less than
  - <= less than or equal to
  - > greater than
  - >= greater than or equal to
  - ~= not equal to
- The result of comparison: True or False. In MATLAB,
  - Any nonzero value (including non empty string) is equivalent to True
  - Only zero is equivalent to False

Note: The <=, >= and ~= operators have "=" as the second character. =<, => and =~ are not valid operators.

Examples of Relational Operators

```
>> a = 2; b = 4;
>> c = a < b
c =
    1
>> d = a>b
d =
    0
```

c = 1 means TRUE
d = 0 means FALSE

```
>> x = 3:7; y = 5:-1:1;
>> z = x>y
z =
    0    0    1    1    1
```

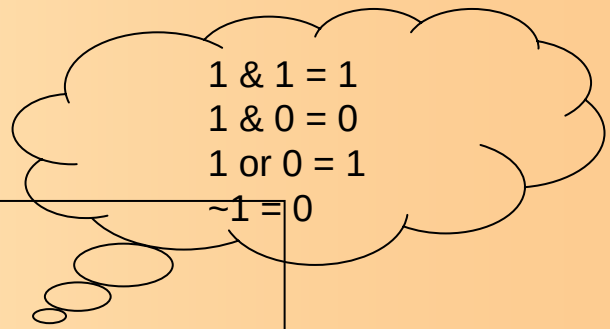
Logical Operators

- Logical operators are used to combine logical expressions (with “and” or “or”), or to change a logical value with “not”

Operator	Meaning
&	and
|	or
~	not

- **Example**:

```
>> a = 2; b = 4;
>> c = a < b
c =
    1
>> d = a>b
d =
    0
>> e = a&d
e =
    0
```



Conditional Execution or Branching (1)

- A comparison or another logical test is often followed by a block of commands to be executed (or skipped).
- Conditional execution in MATLAB:
  (1) Use **if…else….end**

```
if expression
    block of statements
end
```

```
if expression
    block of statements
else
    block of statements
end
```

```
if expression
    block of statements
elseif expression
    block of statements
else
    block of statements
end
```

Examples

```
if x>0
    disp('x is positive')
end
```

```
if x<0
    disp('x is negative')
else
    disp('x is positive')
end
```

```
if x>2
    disp('x is greater than two')
elseif x<0
    disp('x is negative')
else
    disp('x is between zero and two')
end
```

Conditional Execution or Branching (2)

- Conditional execution in MATLAB:
  (2) Use **switch …. case …case….end**

```
switch expression
case value1
    block of statements
case value2
    block of statements
case value3
    block of statements
otherwise
    block of statements
end
```

Example

```
x = '....';
switch x
case 'red'
    disp('Color is red')
case 'green'
    disp('Color is green')
case 'black'
    disp('Color is black')
otherwise
    disp('Color is not red, green or black')
end
```

“A switch construct is useful when a test value can take on discrete value that are either integers or strings”

Repetition or Looping

- A sequence of calculations is repeated until either
  - All elements in a vector or matrix have been processed, OR
  - The calculations have produced a result that meets a predetrmined termination criterion
- Repetition in MATLAB
  - **for** loops

```
for index = expression
    block of statements
end
```

  - **while** loops

```
while expression
    block of statements
end
```

Examples of **for** loops

```
for i = 1:2:n,
    y(i) = x(i)^2;
end
for i = n:-1:1,
    y(i) = x(i)^2;
end
x = 1:5;
sumx = 0;
for i = 1:length(x),
    sumx = sumx + x(i);
end
```

Increment is *increasing* or *decreasing*

"**for** loops are most often used when each element in a vector or matrix is to be processes"

Examples of **while** loops

```
x = ….
y = …..
while abs(x-y) < error,
    z = x – 2*x +1;
    y = sqrt(z);
end
```

- "**while** loops are most often used when an iteration is repeated until a termination criterion is met".
- The **break** and **return** statements provide an alternative way to exit from a loop construct. **break** and **return** may be applied to **for** loops or **while** loops
- **break** is used to escape from an enclosing **while** or **for** loop. Execution continues at the end of the enclosing loop construct
- **return** is used to force an exit from a function. This can have the effect of escaping from a *function*. Any statements following the loop that are in the function body are skipped.

Comparison of **break** and **return**

```
function k = demobreak(n)
.....
while k<=n
    if x(k)>0.8
        break;
    end
    k= k+1;
end
……
```

Jump to end of enclosing "while …. end" block

```
function k = demoreturn(n)
.....
while k<=n
    if x(k) > 0.8
        return;
    end
    k = k+1
end
```

Return to calling function

Programming Tips (1)

1. Variable Input and Output Arguments
   - Each function has internal variables **nargin** and **nargout**
     - Use the value of **nargin** at the beginning of a function to find out how many input arguments were supplied
     - Use the value of **nargout** at the end of a function to find out how many input arguments are expected
   - Usefulness:
     - Allows a single function to perform multiple related tasks
     - Allows functions to assume default values for some inputs, thereby simplifying the use of the function for some tasks
   - Examples: see plot.m
2. Indirect function evaluation (**feval** function)
   - The **feval** function allows a function to be evaluated indirectly
   - Usefulness:
     - Allows routines to be written to process an arbitrary f(x)
     - Separates the reusable algorithm from the problem specific code

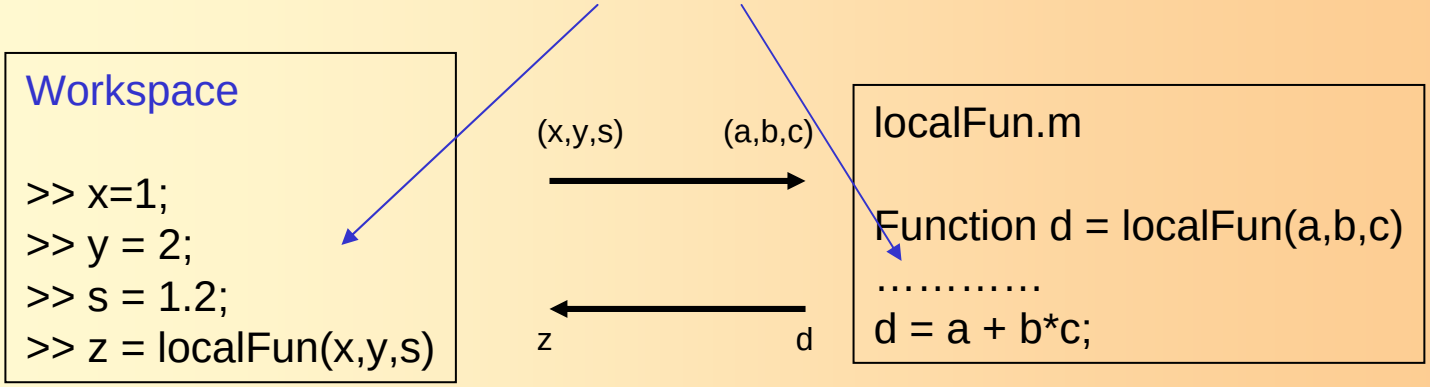
Programming Tips (2)

3. Inline function objects
   - Usefulness:
     - Eliminate need to write separate m-files for functions that evaluate a simple formula
     - Useful in all situations where feval is used.
   - Example:

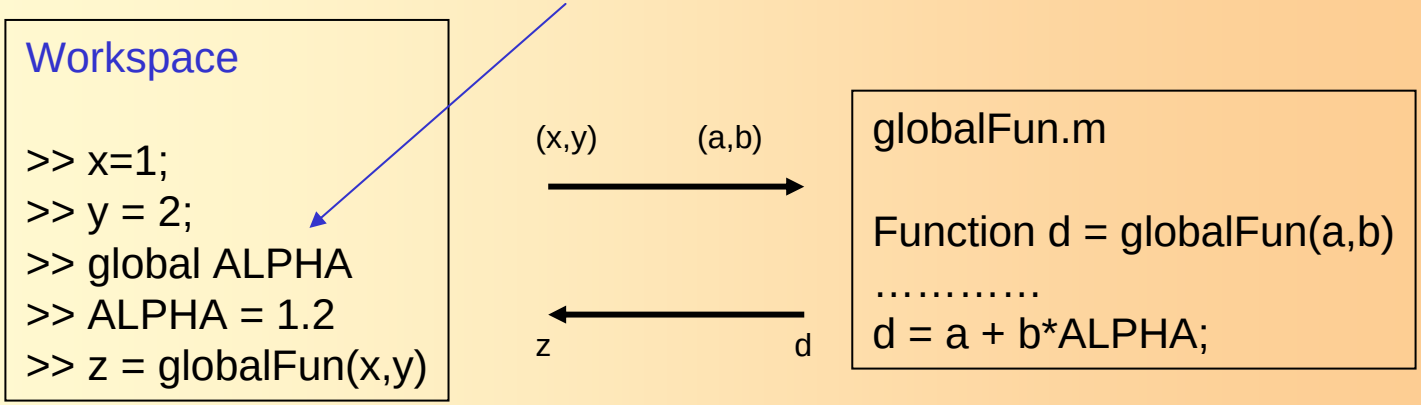
```
function y = myFun(x)
y = x.^2 - log(x);


myFun = inline('x.^2 - log(x)');
```

4. Global variables
   - Usefulness:
     - Allows bypassing of input parameters if no other mechanism (such as pass-through parameters) is available
     - Provides a mechanism for maintaining program state (GUI application)

Local variables



Global variable



Debugging and Organizing MATLAB Programs

- Debugging……..
  - Is inevitable
  - Can be anticipated with good program design
  - Can be done interactively in MATLAB

- Organized programs are…..
  - Easier to maintain
  - Easier to debug
  - Not much harder to write

Preemptive Debugging

- Use defensive programming
  - Do not assume the input is correct. Check it.
  - Provide a “catch” or default condition for a **if…elseif…else….**
  - Include optional print statements that can be switched on when trouble occurs
  - Provide diagnostic error messages
- Break large programming projects into modules
  - Develop reusable tests for key modules
  - Good test problems have known answers
  - Run the tests after changes are made to the module
- Include diagnostic calculations in a module
  - Enclose diagnostic inside **if…end** blocks so that they can be turned off
  - Provide extra print statements that can be turned on and off

Programming Style

- A consistent programming style gives your program a visual familiarity that helps the reader quickly comprehend the intention of the code
- A programming style consists of:
  - Visual appearance of the code
  - Conventions used for variable names
  - Documentation with comment statements
- Use visual layout to suggest organization
  - Indent **if….end** and **for….end** blocks
  - Blank lines separate major blocks of code
- Use meaningful variable names
- Follow Programming and Mathematical Conventions

Example

```
function x = Gauss(A,b),
%
% Inputs:
%  A is the n by n coefficient matrix
%  b is the n by k right hand side matrix
%
% Outputs:
%  x is the n by k solution matrix
%

[n,k1]=size(A); [n1,k] = size(b); x = zeros(n,k);

for i=1:n-1,
   m=-A(i+1:n,i)/A(i,i);
   A(i+1:n,:) = A(i+1:n,:) + m*A(i,:);
   b(i+1:n,:) = b(i+1:n,:) + m*b(i,:);
end;

x(n,;) =b(n,:) ./A(n,n);

for i=n-1:-1:1,
     x(i,:) =(b(i,:)-A(i,i+1:n)*x(i+1:n,:))./A(i,i);
end;
```

Meaningful name

Put Comments

Indent for repetition

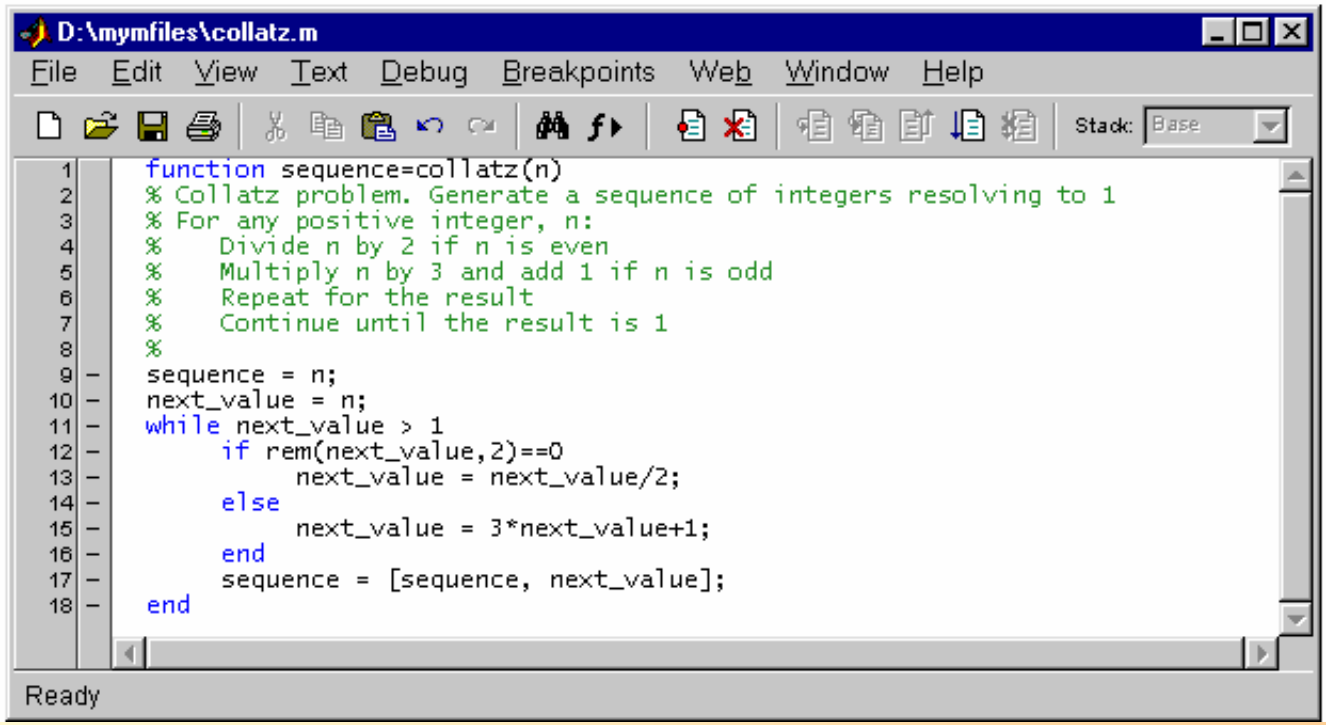
Comment Statements

- Write comments as you write the code, not after
- Include a prologue that supports “help”
  - First line of a function is the definition
  - Second line must be a comment statement
  - All text from the second line up to the first non-comment is printed in response to: **help** *fileName*.
- Assume that the code is going to be used more than once
- Comments should be short notes that augment the meaning of the program statements. Do not parrot the code
- Comments alone do not create good code
  - You cannot fix a bug by changing the comments

Modular Code

- A module should be dedicated to one task
  - Flexibility is provided by input/output parameters
- General purpose modules need….
  - Description of input/output parameters
  - Meaningful error messages so that user understands the problem
- Reuse modules
  - Debug once, use again
  - Minimize duplication of code
  - Any improvements are available to all programs using that module
  - Error messages must be meaningful so that user of general purpose routine understands the problem
- Organization takes experience
  - Goal is not to maximize the number of M-files
  - Organization will evolve on complex projects

Following is an illustration of the Editor/Debugger opened to an existing M-file.



Toolboxes

- Simulink
  - [+] Using Simulink
  - [+] Writing S-Functions
  - [+] SB2SL
- [+] Stateflow
- Real-Time Workshop
  - [+] Real-Time Workshop User's Guide
  - [+] Target Language Compiler
- [+] CDMA Reference Blockset
- [+] Communications Toolbox
- [+] Communications Blockset
- Control System Toolbox
  - [+] Getting Started
  - [+] Creating and Manipulating Models
  - [+] Customization
  - [+] Design Case Studies
  - [+] Reliable Computations
  - [+] GUI Reference
  - [+] Function Reference
- [+] Data Acquisition Toolbox
- [+] Database Toolbox
- [+] Datafeed Toolbox
- [+] Dials & Gauges Blockset
- [+] DSP Blockset
- [+] Excel Link

- [+] Filter Design Toolbox
- [+] Financial Toolbox
- [+] Financial Derivatives Toolbox
- [+] Financial Time Series
- [+] Fixed-Point Blockset
- [+] Fuzzy Logic Toolbox
- [+] GARCH Toolbox
- [+] Image Processing Toolbox
- [+] Instrument Control Toolbox
- [+] Mapping Toolbox
- MATLAB C/C++ Math Library
  - [+] Using the C Math Library
  - [+] C Math Library Reference
  - [+] Using the C++ Math Library
  - [+] C++ Math Library Reference
- [+] MATLAB C/C++ Graphics Library
- [+] MATLAB Compiler
- [+] MATLAB Runtime Server
- [+] MATLAB Web Server
- [+] Motorola DSP Developer's Kit
- Model Predictive Control Toolbox
- Mu-Analysis and Synthesis Toolbox
- Nonlinear Control Design Blockset
- [+] Neural Network Toolbox
- [+] Optimization Toolbox
- Partial Differential Equations (PDE) Toolbox
- [+] Power System Blockset
- [+] Real-Time Windows Target
- [+] Requirements Management Interface
- [+] Report Generator
- Robust Control Toolbox
- [+] Signal Processing Toolbox
- [+] Spline Toolbox
- [+] Statistics Toolbox
- [+] Symbolic Math Toolbox
- [+] System Identification
- [+] Wavelet Toolbox
- [+] xPC Target
- Installation
  - [+] MATLAB Installation Guide for PC
  - [+] MATLAB Installation Guide for Unix
- [+] Support and Web Services