

Built-In Functions and User-Defined Function

Multiple Useful Functions

- Basics of Built-in Functions
- Help Feature
- Elementary Functions
- Data Analysis
- Random Numbers
- Complex Numbers

What is a Built-In Function?

- A computational expression that uses one or more input values to produce an output value.
- MATLAB functions have 3 components: input, output, and name
- For example, for `b = tan(x)`
 - x is the input,
 - b is the output,
 - tan is the name of a built-in function

MATLAB Functions

- Functions take the form:
variable = function(number or variable)
- MATLAB has many functions stored in its file system.
- To use one of the built-in functions you need to know its name and what the input values are.
- For example, the square root function: **sqrt ()**.
- Find the square root of 9 using MATLAB

```
>> a = sqrt(9)  
a = 3
```

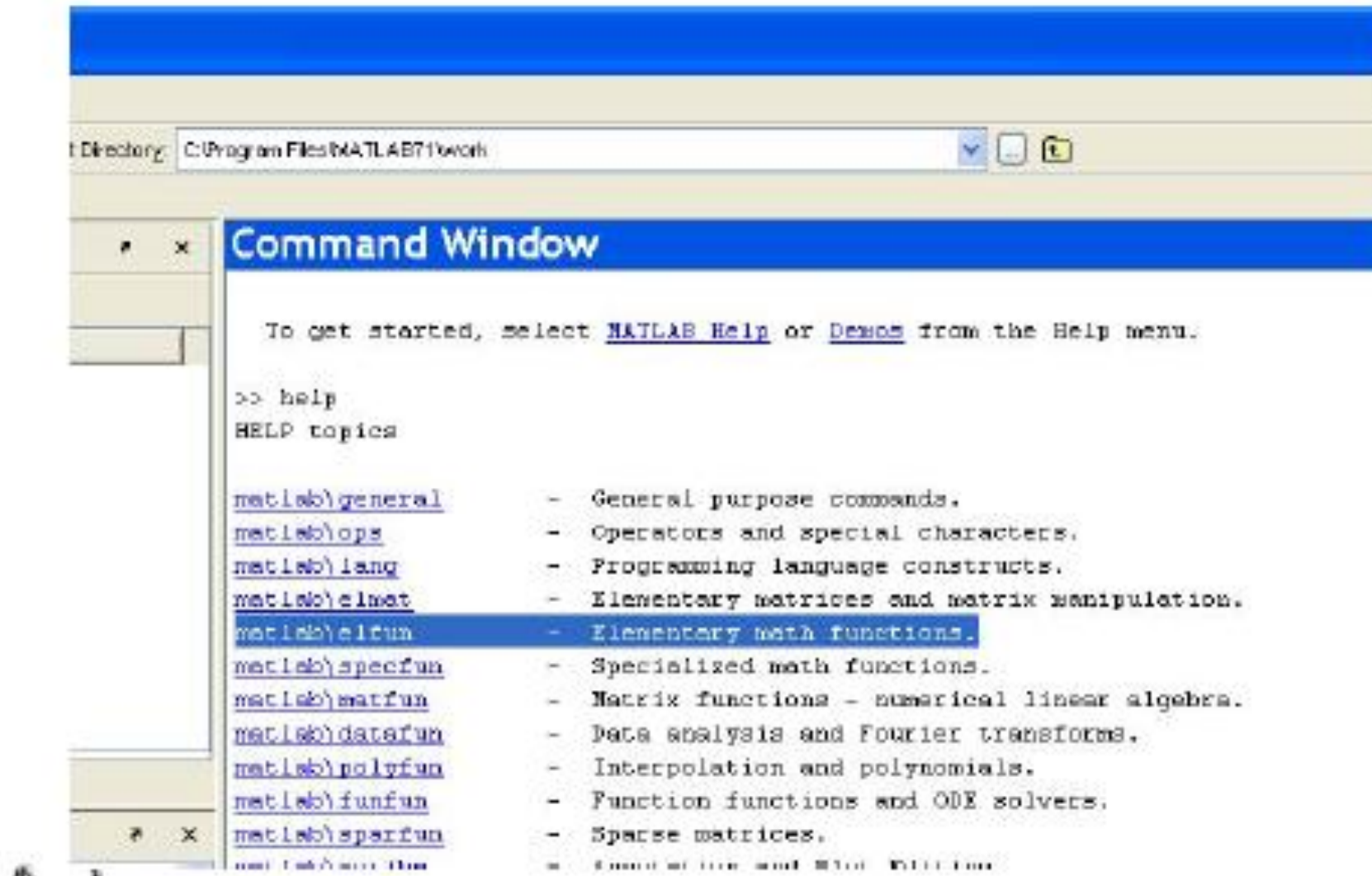
HELP Feature

- You may not always recall the name and syntax of a function you want to use since MATLAB has so many built-in functions.
- MATLAB has an indexed library of its built-in functions that you can access through the **help** feature.
- If you type **help** in the command window MATLAB provides a list of help topics.

Help

- In MATLAB command window type
`>> help`
- If we are interested in the elementary mathematics functions, we find it on the list of help topics (5th down) and click it.
- A list of commands appears with the description of what each one does.
- Try a few!

MATLAB Help



More Help

- For more specific help: **help topic**

- Check it out:

```
>> help tan
```

- MATLAB describes what the function **tan** does and provides helpful links to similar or related functions to assist you.

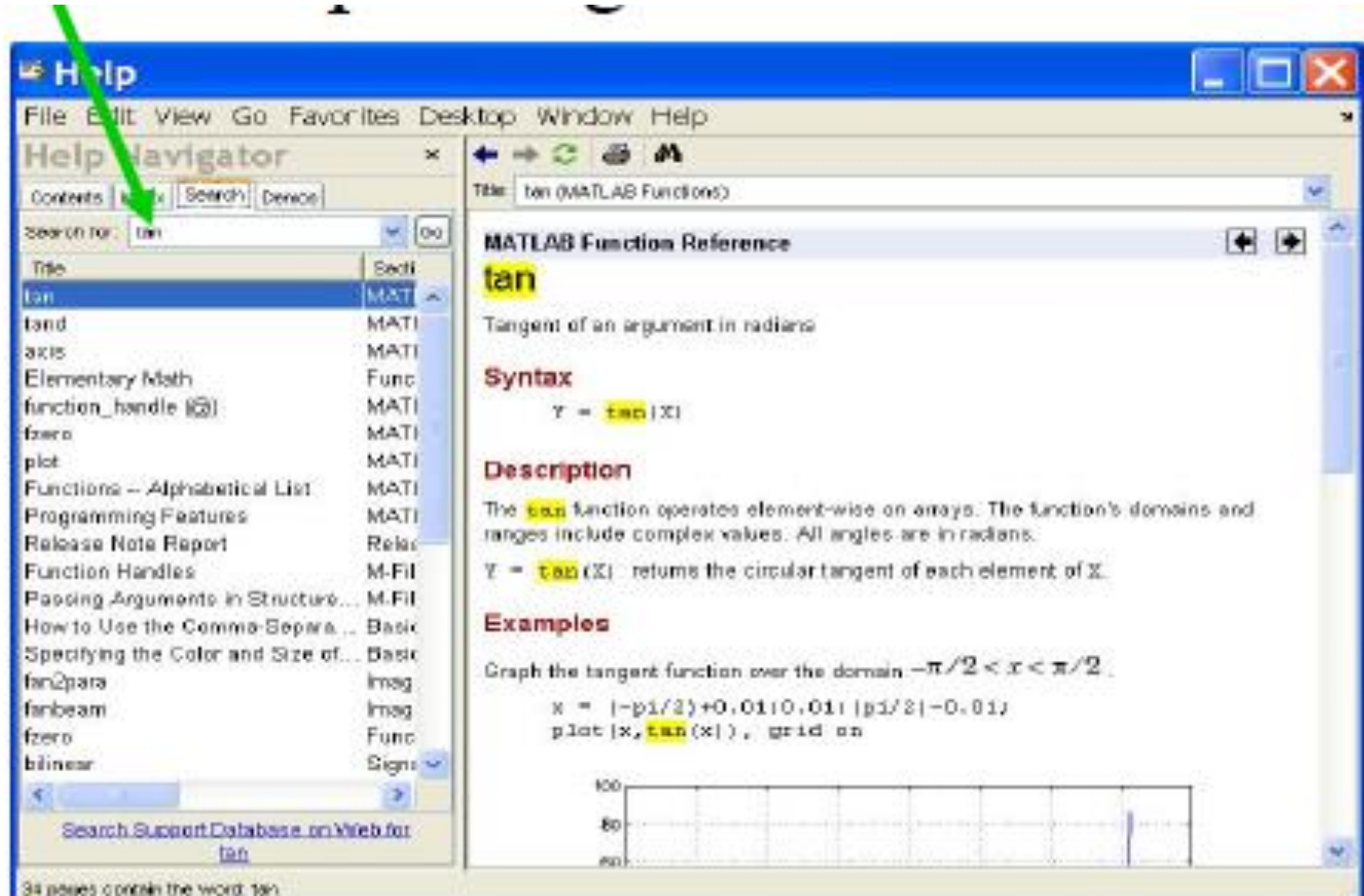
Hands-On

- Use MATLAB help to find the exponential, natural logarithm, and logarithm base 2 functions
- Calculate e^7 `>> exp(7)`
- Calculate $\ln(4)$ `>> log(4)`
- Calculate $\log_2(12)$ `>> log2(12)`

Help Navigator

- Click on **Help** on the tool bar at the top of MATLAB, and select **MATLAB Help**.
- A HELP window will pop up.
- Under **Help Navigator** on the left of screen select the **Search** tab.
- You can Search for specific help topics by typing your topic in the space after the **:**.
- You can also press **F1** on your keyboard to access the windowed help.

Help Navigator window



Elementary Mathematical Functions

- MATLAB can perform all of the operations that your calculator can and more.
- Search for the topic **Elementary math** in the Help Navigator just described.
- Try the following in MATLAB to continue your exploration of MATLAB capabilities

```
>> sqrt(625)
```

```
ans = 25
```

```
>> log(7)
```

```
ans = 1.9459
```

Of course, try a few others.

Rounding Functions

- Sometimes it is necessary to round numbers. MATLAB provides several functions to do this.

```
>> round(x) % round to nearest  
            % whole number
```

```
>> fix(x)    % round towards zero
```

```
>> floor(x)  % round down
```

```
>> ceil(x)   % round up
```

Trigonometric Functions

- MATLAB can compute numerous trigonometric functions
- The default units for angles is radians.
 - To convert degrees to radians, the conversion is based on the relationship: $180 \text{ degrees} = \pi \times \text{radians}$
 - Note: `pi` is a built-in constant in MATLAB.
- Inverse functions are **`asin()`**, **`atan()`**, **`acos()`**, etc.
- There are trigonometric functions defined for degrees instead of radian such as **`sind()`**, **`cosd()`**, etc.
- Type **`help elfun`** in the command window for more functions and information.

Data Analysis Functions

- It is often necessary to analyze data statistically.
- MATLAB has many statistical functions:

`max()`

`sum()`

`size()`

`min()`

`prod()`

`length()`

`mean()`

`sort()`

`std()`

`median()`

`sortrows()`

`var()`

`find()`

Data Analysis Practice

```
>> x = [5 3 7 10 4]
```

What is the largest number in array x and where is it located?

```
>> [value, position] = max(x)
```

```
Value = 10      % highest value is 10
```

```
Position = 4    % it is the 4th value
```

What is the median of the above array?

```
>> median (x)
```

```
ans = 5
```

What is the sum of the above array?

```
>> sum(x)
```

```
ans = 29
```


Hands-On

```
>> v = [2 24 53 7 84 9]
```

```
>> y = [2 4 56; 3 6 88]
```

Sort `v` in descending order

Find the size of `y`

Find the standard deviation of `v`

Find the cumulative product of `v`

Sort the rows of `y` based on the 3rd column.

Generation of Random Numbers

- `rand` produces random number between 0 and 1
- `rand(n)` produces $n \times n$ matrix of random numbers between 0 to 1.
- `rand(n,m)` produces $n \times m$ matrix of random numbers between 0 and 1.

To produce a random number between 0 and 40:

```
>> w = 40*rand
```

To produce a random number between -2 and 4:

```
>> w = -2 + (4 - (-2)) * rand
```

Note: `rand` will NEVER give exactly 0 or exactly 1.

Complex Numbers

- Complex numbers take the form of $a+bi$:
 - a is the real part
 - b is the imaginary part
 - and $i = \sqrt{-1}$
- Complex numbers can be created as follows:

```
>> a = 2; b = 3;  
>> c = a+b*i           %method 1  
>> c = complex(a,b)    %method 2
```

$c = 2.0000 + 3.0000i$
- Note: both i and j are built-in MATLAB constants that equal $\sqrt{-1}$

Complex Numbers Continued

- Function to extract the real and imaginary components of a complex number:

`real(c)`

`imag(c)`

- Function to find the absolute value or modulus of a complex number:

`abs(c) % = sqrt(real(c)^2 + imag(c)^2)`

- Function to find the angle or argument (in radians) of a complex number:

`angle(c) % = atan(imag(c)/real(c))`

Other Useful Functions

- `clock` %Outputs 1x6 array containing year, month, day, hour, min, sec.
- `date` %Outputs date as string
- `tic` %Start timer
- `toc` %Output time elapsed
- `pause(XX)` %pause for XX seconds

Exercises

- Find the modulus (magnitude, `abs`) and angle (argument) of the complex number $3+4i$.
- Generate a 4x4 array of random numbers between 0 and 10. Sort each column of the array.
- Use MATLAB's help function to find built-in functions to determine:
 - The base 10 logarithm of 5
 - The secant of π

User-Defined Functions in MATLAB

- MATLAB permits us to create our own functions, which are indispensable when it comes to breaking a problem down into manageable logical pieces.
- A function M-file is similar to a script file in that it also has a .m extension.
- These are scripts that take in certain inputs and return a value or set of values
- A function M-file differs from a script file in that a function M-file communicates with the MATLAB workspace only through specially designated input and output arguments

Basic rules

functionname.m

```
function [output arguments] = functionname(input arguments)
% Comment describing the function
Statements here; these must include assigning values to
all of the output arguments listed in the header
```

areacirc.m

```
function [area, circum] = areacirc(rad)
% This function calculates the area and
% the circumference of a circle
area = pi * rad .* rad;
circum = 2 * pi * rad;
```

```
>> [a c] = areacirc(4)
a =
    50.2655
c =
    25.1327
```


Basic rules

```
function [ outarg1, outarg2, ... ] = name( inarg1, inarg2, ... )
% comments to be displayed with help
...
outarg1 = ... ;

outarg2 = ... ;
...
```

- **function keyword**
The function file *must* start with the keyword `function` (in the function definition line).
- **Input and output arguments**
The input and output arguments (*inarg1*, *outarg1*, etc.) are “dummy” variables, and serve only to define the function’s means of communication with the workspace. Other variable names may therefore be used in their place when the function is called (referenced).
- **Multiple output arguments**
If there is more than one output argument, the output arguments *must* be separated by commas and enclosed in square brackets in the function definition line, as shown.
If there is only one output argument square brackets are not necessary.
- **Naming convention for functions**
Function names must follow the MATLAB rules for variable names.
If the filename and the function definition line name are different, the internal name is ignored.
- **Help text**
When you type `help function_name`, MATLAB displays the comment lines that appear between the function definition line and the first non-comment (executable or blank) line.
The first comment line is called the *H1 line*. The `lookfor` function searches on and displays only the H1 line. The H1 lines of all M-files in a directory are displayed under the Description column of the Desktop Current Directory browser.

functionname.m

```
function [output arguments] = functionname(input arguments)
% Comment describing the function
Statements here; these must include assigning values to
all of the output arguments listed in the header
```

- **Local variables: scope**
Any variables defined inside a function are inaccessible outside the function. Such variables are called *local variables*—they exist only inside the function, which has its own workspace separate from the base workspace of variables defined in the Command Window.
- **Global variables**
Variables which are defined in the base workspace are not normally accessible inside functions, i.e., their scope is restricted to the workspace itself, unless they have been declared `global`, e.g.,

```
global PLINK PLONK
```

Basic rules

```
function [ outarg1, outarg2, ... ] = name( inarg1, inarg2, ... )
% comments to be displayed with help
...
outarg1 = ... ;

outarg2 = ... ;
...
```

■ Persistent variables

A variable in a function may be declared **persistent**. Local variables normally cease to exist when a function returns. Persistent variables, however, remain in existence between function calls. A persistent variable is initialized to the empty array.

Persistent variables remain in the memory until the M-file is cleared or changed, e.g.,

```
clear test
```

The function `mlock` inside an M-file prevents the M-file from being cleared. A locked M-file is unlocked with `munlock`. The function `mislocked` indicates whether an M-file can be cleared or not.

The Help entry on `persistent` declares confidently: "It is an error to declare a variable persistent if a variable with the same name exists in the current workspace." However, this is definitely not the case at the time of writing (I tried it!).

■ Functions that do not return values

```
functionname.m
function functionname(input arguments)
% Comment describing the function
Statements here
```

Omit the output arguments and the equal sign in the function definition line.

```
function stars(n)
asteriks = char(abs('**')*ones(1,n));
disp( asteriks)
```

codes = unicode2native('*', 'US-ASCII')

■ Vector arguments

When an output argument is a vector, it is initialized each time the function is called, any previous elements being cleared. Its size at any moment is therefore determined by the most recent call of the function. For example, suppose the function `test.m` is defined as:

```
function a=test
a(3)=92;
```

Then if `b` is defined in the base workspace as:

```
b =
    1     2     3     4     5     6
```

the statement:

```
b=test
```

results in,

```
b =
    0     0    92
```

Basic rules

```
function [ outarg1, outarg2, ... ] = name( inarg1, inarg2, ... )
% comments to be displayed with help
...
outarg1 = ... ;

outarg2 = ... ;
...
```

■ How function arguments are passed

If a function changes the value of any of its input arguments, the change is *not reflected* in the actual input argument on return to the workspace (unless the function is called with the same input and output argument—see below). For the technically minded, input arguments appear to be passed *by value*.

■ Simulated pass by reference

A function may be called with the same actual input and output argument. For example, the following function `prune.m` removes all the zero elements from its input argument:

```
function y=prune(x)
y=x(x ~= 0);
```

You can use it to remove all the zero elements of the vector `x` as follows:

```
x=prune(x)
```

■ Checking the number of function arguments

A function may be called with all, some, or none of its input arguments. If called with no arguments, the parentheses must be omitted. You may not use more input arguments than appear in its definition.

The same applies to output arguments—you may specify all, some, or none of them when you use the function. If you call a function with no output arguments, the value of the first one in the definition is returned.

There are times when a function may need to know how many input/output arguments are used on a particular call. In that case, the functions `nargin` and `nargout` can be used to determine the number of actual input and output arguments. For example:

```
function c=addme(a,b)
switch nargin
    case 2
        c=a+b;
    case 1
        c=a+a;
    otherwise
        c=0;
end
end
```

Variable scope

The *scope* of any variable is the workspace in which it is valid. The workspace created in the Command Window is called the *base workspace*.

As we have seen before, if a variable is defined in any function it is a *local variable* to that function, which means that it is known and used only within that function.

However, scripts (as opposed to functions) *do* interact with the variables that are defined in the Command Window.

mysum.m

```
function runsum = mysum(vec)
% This function sums a vector
runsum = 0;
for i=1:length(vec)
    runsum = runsum + vec(i);
end
```

```
% This script sums a vector
vec = 1:5;
runsum = 0;
```

```
for i=1:length(vec)
    runsum = runsum + vec(i);
end
disp(runsum)
```

Persistent Variables

Normally, when a function stops executing, the local variables from that function are cleared. That means that every time a function is called, memory is allocated and used while the function is executing, but released when it ends. With variables that are declared as *persistent variables*, however, the value is not cleared so the next time the function is called, the variable still exists and retains its former value.

persistex.m

```
% This script demonstrates persistent variables
% The first function has a variable count
fprintf('This is what happens with a normal variable:\n')
func1
func1
% The second fn has a persistent variable count
fprintf('\nThis is what happens with a persistent variable:\n')
func2
func2
```

func1.m

```
function func1
% This function increments a variable count
count = 0;
count = count + 1;
fprintf('The value of count is %d\n',count)
```

func2.m

```
function func2
% This function increments a persistent variable count
persistent count
if isempty(count)
    count = 0;
end
count = count + 1;
fprintf('The value of count is %d\n',count)
```

User-Defined Functions

- Suppose we want to plot

$$\sin(3*x) + \sin(3.1*x)$$

- Create user-defined function

```
function r=f(x)  
    r=sin(3*x)+sin(3.1*x)
```

- Save as f.m

User-Defined Functions (cont)

- Now just call it:

```
x=0:0.1:50;
```

```
y=f(x);
```

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

Practice

- Create an m-file that calculates the function
 $g(x) = \cos(x) + \cos(1.1 * x)$
- Use it to plot $g(x)$ from $x=0$ to 100
- Note: previous function was

```
function r=f(x)  
r=sin(3*x)+sin(3.1*x)
```

- ... and plot commands were

```
x=0:0.1:50;  
y=f(x);  
plot(x,y)
```


Practice

- Create an m-file that calculates the function $g(x, \delta) = \cos(x) + \cos((1+\delta)x)$ for a given value of δ
- Use it to plot $g(x, \delta)$ from $x=0$ to 100 for $\delta = 0.1$ and 0.2

Subfunctions

A function M-file may contain the code for more than one function. The first function in a file is the *primary* function, and is the one invoked with the M-file name. Additional functions in the file are called *subfunctions*, and are visible only to the primary function and to other subfunctions.

Each subfunction begins with its own function definition line. Subfunctions follow each other in any order *after* the primary function.

The subfunction will not be available to other functions

function example

clear all

r=sumofcubes(20);

fprintf('The sum of the first 20 cubes is %i\n',r)

%

function r=sumofcubes(N)

ans=0;

for i=1:N

ans=ans+i^3;

end

r=ans;

Subfunctions(cont.)

rectarea.m

```
% This program calculates & prints the area of a rectangle

% Call a fn to prompt the user & read the length and width
[length, width] = readlenwid;
% Call a fn to calculate and print the area
printrectarea(length, width)
```

readlenwid.m

```
function [l,w] = readlenwid
% This function prompts the user for the length and width
l = input('Please enter the length: ');
w = input('Please enter the width: ');
```

printrectarea.m

```
function printrectarea(len, wid)
% This function prints the rectangle area
% It calls a subfunction to calculate the area
area = calcrectarea(len,wid);
fprintf('For a rectangle with a length of %.2f\n',len)
fprintf('and a width of %.2f, the area is %.2f\n', ...
    wid, area);

function area = calcrectarea(len, wid)
% This function calculates the rectangle area
area = len * wid;
```

This is an example of running this program:

```
>> rectarea
Please enter the length: 6
Please enter the width: 3
For a rectangle with a length of 6.00
and a width of 3.00, the area is 18.00
```

An Example – with Numeric

- Suppose we're looking for a \$100k, 30-year mortgage. What interest rate do I need to keep payments below \$700 per month?

$$100000 - 700 \left[\frac{(1+i)^{360} - 1}{i(1+i)^{360}} \right] = 0$$

- Solve for i

Create the function

```
function s=f(i)  
p=100000;  
n=360;  
a=700;  
s=p-a*((1+i).^n-1)./(i.*(1+i).^n);
```

Plot the Function

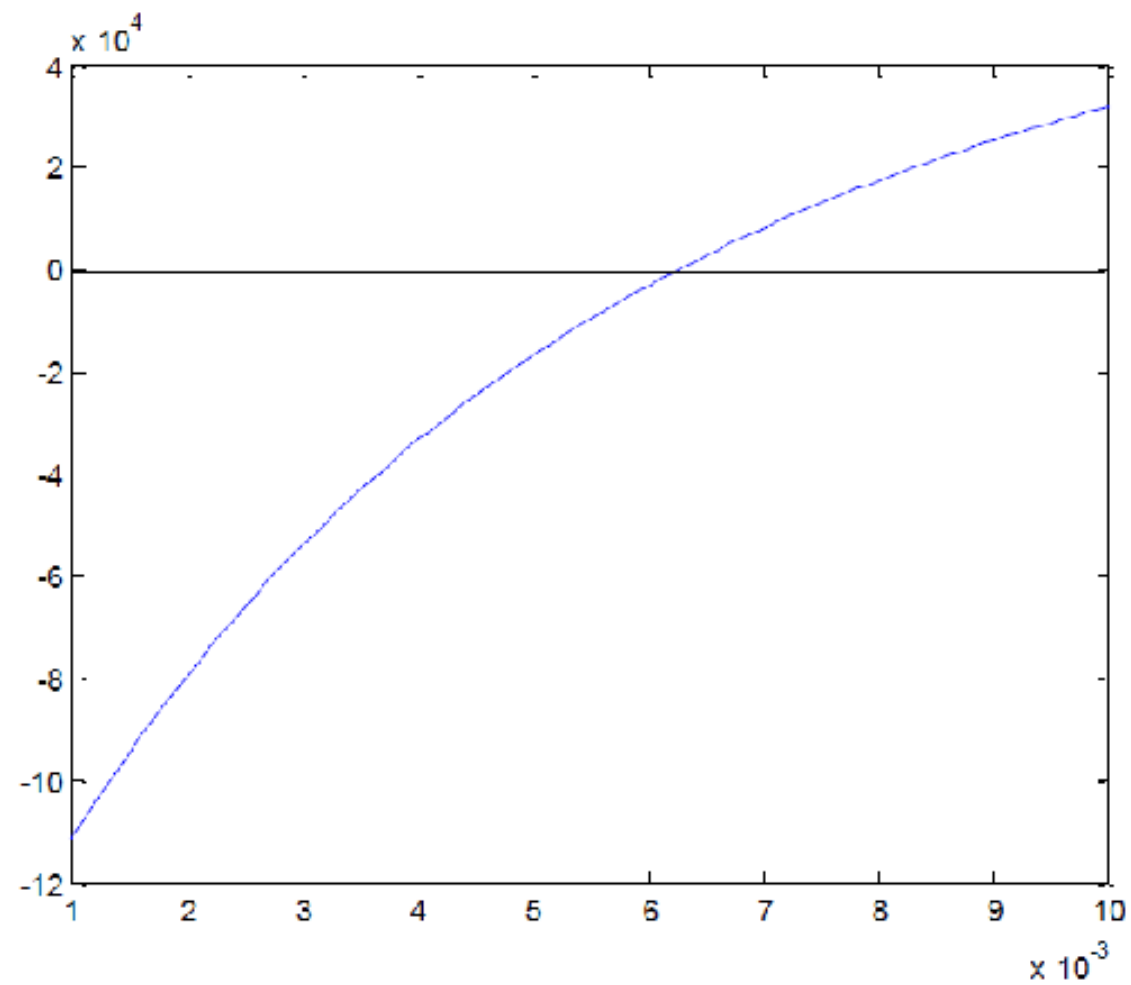
- First save file as f.m
- Now enter the following

```
i=0.001:0.0001:0.01;
```

```
y=f(i);
```

```
plot(i,y)
```

The Plot



Result

- Zero=crossing is around $i=0.006$
- Annual interest rate is $12*i$, or about 7%
- Try more accurate solution

`12*fzero('f',0.006)`

- This gives about 7.5%

Function handles

Try the following on the command line:

```
fhandle = @sqrt;  
feval(fhandle, 9)  
feval(fhandle, 25)
```

`feval(fhandle, x)` is the same as `sqrt(x)`

The handle provides a way of referring to the function, in a list of input arguments to another function.

```
x=0:0.01:100;
```

```
delta=1.05
```

```
gg=@(x,delta) cos(x)+cos(delta*x)
```

```
y=gg(x,delta);
```

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

Practice

- Consider the function

$$f(x) = \exp(-a * x) * \sin(x)$$

- Plot using an inline function
- Use $0 < x < 10$ and $a = 0.25$
- Note: syntax can be taken from:

gg=@(x, delta) cos(x)+cos(delta*x)

Command/Function duality

In the earliest versions of MATLAB there was a clear distinction between *commands* like:

```
clear
save junk x y z
whos
```

and *functions* like

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

If commands had any arguments they had to be separated by blanks with no brackets. Commands altered the environment, but didn't return results. New commands could not be created with M-files.

From Version 4 onwards commands and functions are "duals," in that commands are considered to be functions taking string arguments. So:

```
axis off
```

is the same as,

```
axis('off' )
```

Other examples are

```
disp Error!
hold('on')
```

This duality makes it possible to generate command arguments with string manipulations, and also to create new commands with M-files.

Function name resolution

Remember that a variable in the workspace can “hide” a built-in function of the same name, and a built-in function can hide an M-file.

Specifically, when MATLAB encounters a name it resolves it in the following steps:

1. Checks if the name is a variable.
2. Checks if the name is a *subfunction* of the calling function.
3. Checks if the name is a *private function*.
4. Checks if the name is in the directories specified by MATLAB's search path.

MATLAB therefore always tries to use a name as a variable first, before trying to use it as a script or function.

MATLAB Errors and Debugging

- The process of finding and correcting errors is called debugging. It is a primary task in code design.
- Debugging can be very difficult and frustrating
- Three common types of errors
 - Syntax – incorrect spelling or use of arguments
 - Run-time – illegal operations
 - Logic – mistakes in math, branching, or looping
 - Can be very tough to find
- Two good debugging approaches for MATLAB are:
 - Breaking script into sections and running each section separately
 - Using MATLAB's debugging tools

Syntax Errors

- Errors in the MATLAB statement itself, such as spelling or punctuation errors. If found MATLAB won't even try to run your script.
- Examples:
 - `2=x` or `for=10` - Where's the error(s)?
 - Error: The expression to the left of the equals sign is not a valid target for an assignment.
 - `a=(4+3)/2)` - Where's the error(s)?
 - Error: Unbalanced or unexpected parenthesis or bracket.

Run-time Errors

- Illegal operations that cause a script/program to “crash” when trying to run
- Examples:
 - `x = [1 2 3;4]` - Where's the error(s)?
 - **Error: Dimensions of matrices being concatenated are not consistent.** – How can you fix it?
 - Fix: `x = [1 2 3;4 0 0]` or similar
 - `sni(pi)` - Where's the error(s)?
 - **Error: Undefined function 'sni' for input arguments of type 'double'.** – How can you fix it?
 - Fix: `sin(pi)`

Example Run-time Error

- Running the following generates a run-time error. Can you spot it?

```
x = 4:10;  
for k = 0:length(x)  
    v = v + x(k);  
end
```

– Error 1: Undefined function or variable 'v'. Fix?

- Fix by initializing v before **for** loop: **v = 0**

– Error 2: Attempted to access x(0); index must be a positive integer or logical. Fix?

- Fix by changing 0 to 1: **k = 1:length(x)**

Logical Errors

- Occur when the program runs without displaying an error, but produces an unexpected result.

- Example:

```
side = input('What is the side of your square? ')
area = side + side;
fprintf('The area is %.2f.\n', area)
```

- Can you find the error?
- What do you expect the result to be if the user enters 5?
- What will be the result if the user enters 5?
- How do you fix the error?
 - `area = side*side;`
- Note: If you enter 2 instead, you will see that it could be difficult to find logical errors.

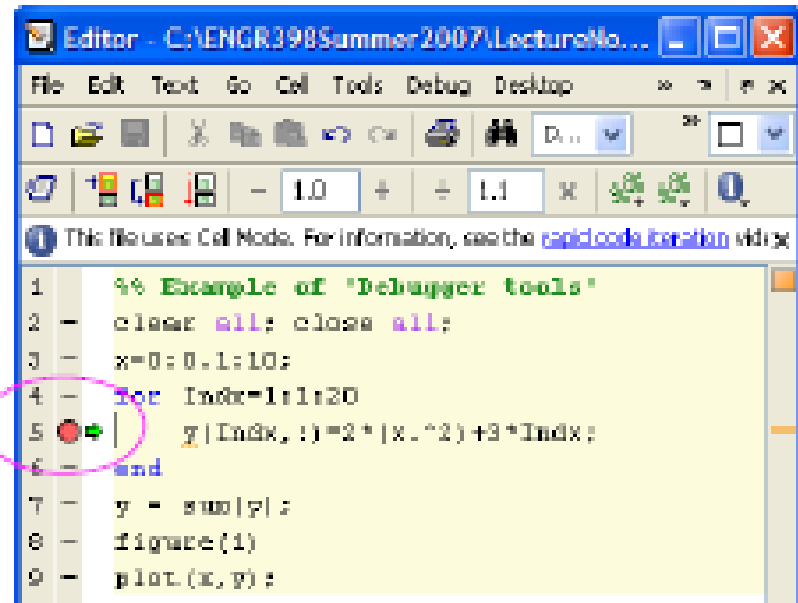
Debugging using the Command Window

- Data dump – list the variable without the “;”
 - Crude (not very space efficient), but effective for debugging
 - Watch out for large arrays
- Display the data
 - `disp (variable)`
 - `disp (string)`
 - Use `num2str ()` to convert a number to a string and include in the same line as the string
 - `disp ([‘The number is: ‘, num2str(2)])`;
 - Clean way to temporarily display data during debugging.

MATLAB Debug Tools

- There is a debugger built into the MATLAB editor
- To enter debug mode, set one or more “break points” by clicking the “–” marks beside a line numbers in the MATLAB editor.
- When the script is run, it will pause at the first “break point”.
- When the script is paused, you can use the command window to investigate current variables.
- You can then use “step” to continue executing the script one line at a time or select other options as detailed on the next slides.

Example of Debugging Tools

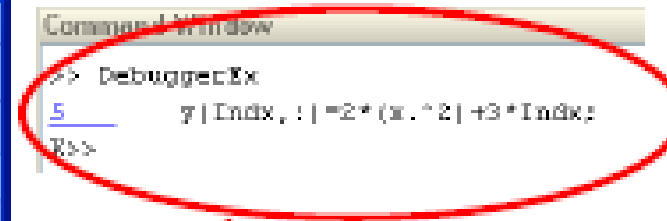


The image shows the MATLAB Editor window with a script titled "Editor - C:\ENGR398Summer2007\LectureNo...". The script contains the following code:

```
1  %% Example of 'Debugger tools'
2  - clear all; close all;
3  - x=0:0.1:10;
4  - for Index=1:1:20
5  -     y(Index,:)=2*(x.^2)+3*Index;
6  - end
7  - y = sum(y);
8  - figure(1)
9  - plot(x,y);
```

A red circle highlights the break point icon (a red circle with a green arrow) on line 5. A pink arrow points from the text "A break point is set. You can set more than one if you want." to this icon.

A break point is set. You can set more than one if you want.



The image shows the MATLAB Command Window with the following text:

```
>> DebuggerEx
5      y(Index,:)=2*(x.^2)+3*Index;
R>>
```







A red circle highlights the Command Window, and a red arrow points from the text "Note:" to it.

Note:

1. Program stopped at the break point.
2. You can check any variables, plot data, etc. at the point program stopped.

Debugging Mode Menu



Toolbar Button	Debug Menu Item	Description	Function Alternative
	Run to Cursor	Continue execution of file until the line where the cursor is positioned. Also available on the context menu.	None
	Step	Execute the current line of the file.	<code>dbstep</code>
	Step In	Execute the current line of the file and, if the line is a call to another function, step into that function.	<code>dbstep in</code>
	Continue	Resume execution of file until completion or until another breakpoint is encountered.	<code>dbcont</code>
	Step Out	After stepping in, run the rest of the called function or local function, leave the called function, and pause.	<code>dbstep out</code>
	Quit Debugging	Exit debug mode.	<code>dbquit</code>

Summary (1)

- Good structured programming requires real problem-solving programs to be broken down into function M-files.
- The name of a function in the function definition line should be the same as the name of the M-file under which it is saved. The M-file must have the extension `.m`.
- A function may have input and output arguments, which are usually its only way of communicating with the workspace. Input/output arguments are dummy variables (placeholders).
- Comment lines up to the first non-comment line in a function are displayed when `help` is requested for the function.
- Variables defined inside a function are local variables and are inaccessible outside the function.
- Variables in the workspace are inaccessible inside a function unless they have been declared `global`.

Summary (2)

- Variables in the workspace are inaccessible inside a function unless they have been declared `global`.
- A function does not have to have any output arguments.
- Input arguments have the appearance of being passed by value to a function. This means that changes made to an input argument inside a function are not reflected in the actual input argument when the function returns.
- A function may be called with fewer than its full number of input/output arguments.
- The functions `nargin` and `nargout` indicate how many input and output arguments are used on a particular function call.
- Variables declared `persistent` inside a function retain their values between calls to the function.
- Subfunctions in an M-file are accessible only to the primary function and to other subfunctions in the same M-file.
- Private functions are functions residing in a sub-directory named `private` and are accessible only to functions in the parent directory.

Summary (3)

- Functions may be parsed (compiled) with the `pcode` function.
The resulting code has the extension `.p` and is called a P-code file.
- The Profiler enables you to find out where your programs spend most of their time.
- A handle for a function is created with `@`.
A function may be represented by its handle. In particular the handle may be passed as an argument to another function.
- `feval` evaluates a function whose handle is passed to it as an argument.
- MATLAB first tries to use a name as a variable, then as a built-in function, and finally as one of the various types of function.
- Command/function duality means that new commands can be created with function M-files, and that command arguments may be generated with string manipulations.
- The Editor/Debugger enables you to work through a script or function line-by-line in debug mode, examining and changing variables on the way.
- A function may call itself. This feature is called recursion.

Practice Problem

Write a function that will receive as an input argument a temperature in degrees Fahrenheit, and will return the temperature in both degrees Celsius and Kelvin. The conversion factors are: $C = (F - 32) * 5/9$ and $K = C + 273.15$.

Practice Problem

Write a function that will receive as an input argument a length in feet and will return the length in both yards and centimeters. One yard is equal to 3 feet. One inch is equal to 2.54 centimeters, and there are 12 inches in a foot.

Practice Problem

Write a function that will receive the radius of a circle and will print both the radius and diameter of the circle in a sentence format. This function will not return any value; it simply prints.

Practice Problem

Write a function that receives a vector as an input argument and prints the elements from the vector in a sentence format.

```
>> printvecelems([5.9 33 11])  
Element 1 is 5.9  
Element 2 is 33.0  
Element 3 is 11.0
```

Practice Problem

Hurricanes are categorized based on the winds. The following table shows the category number for hurricanes with varying wind ranges and what the storm surge is (in feet above normal).

1	74–95	4–5
2	96–110	6–8
3	111–130	9–12
4	131–155	13–18
5	>155	>18

Write a function that will receive as an input argument the wind speed, and will return the category number and the minimum value of the typical storm surge.

Practice Problem

Write a function that will receive an integer n and a character as input arguments, and will print the character n times.

Practice Problem

The lump sum S to be paid when interest on a loan is compounded annually is given by $S = P(1 + i)^n$, where P is the principal invested, i is the interest rate, and n is the number of years. Write a program that will plot the amount S as it increases through the years from 1 to n . The main script will call a function to prompt the user for the number of years (and error-check to make sure that the user enters a positive integer). The script will then call a function that will plot S for years 1 through n . It will use 0.05 for the interest rate and \$10,000 for P .

Practice Problem

The resistance R in ohms of a conductor is given by $R = \frac{E}{I}$, where E is the potential in volts and I is the current in amperes. Write a script that will

- Call a function to prompt the user for the potential and the current.
- Call a function that will print the resistance; this will call a subfunction to calculate and return the resistance.

Practice Problem

Write a function *per2* that receives one number as an input argument. The function has a persistent variable that sums the values passed to it. Here are the first two times the function is called:

```
>> per2(4)
```

```
ans =
```

```
4
```

```
>> per2(6)
```

```
ans =
```

```
10
```

Practice Problem

What would be the output from the following program? Think about it, write down your answer, and then type it in to verify.

testscope.m

```
answer = 5;  
fprintf('Answer is %d\n',answer)  
pracfnc  
pracfnc  
fprintf('Answer is %d\n',answer)  
printstuff  
fprintf('Answer is %d\n',answer)
```

pracfnc.m

```
function pracfnc  
persistent count  
if isempty(count)  
    count = 0;  
end  
count = count + 1;  
fprintf('This function has been called %d times.\n',count)
```

printstuff.m

```
function printstuff  
answer = 33;  
fprintf('Answer is %d\n',answer)  
pracfnc  
fprintf('Answer is %d\n',answer)
```

Practice Problem

The hyperbolic sine for an argument x is defined as:

$$\text{hyperbolicsine}(x) = (e^x - e^{-x}) / 2$$

Write a function *hypsin* to implement this. The function should receive one input argument x and return the value of the hyperbolic sine of x . Here are some examples of using the function:

```
>> hypsin(2.1)
```

```
ans =
```

```
4.0219
```

```
>> help hypsin
```

```
Calculates the hyperbolic sine of x
```

```
>> fprintf('The hyperbolic sine of %.1f is %.1f\n',...
```

```
1.9,hypsin(1.9))
```

```
The hyperbolic sine of 1.9 is 3.3
```

Practice Problem

The velocity of an aircraft typically is given in either miles/hour or meters/second. Write a function that will receive one input argument that is the velocity of an airplane in miles per hour and will return the velocity in meters per second. The relevant conversion factors are: one hour = 3600 seconds, one mile = 5280 feet, and one foot = .3048 meters.

Practice Problem

Write a function called *pickone* that will receive one input argument *x*, which is a vector, and will return one random element from the vector. For example,

```
>> pickone(4:7)
```

```
ans =
```

```
5
```

```
>> disp(pickone(-2:0))
```

```
-1
```

```
>> help pickone
```

```
pickone(x) returns a random element from vector x
```

Practice Problem

A function can return a vector as a result. Write a function *vecout* that will receive one integer argument and will return a vector that increments from the value of the input argument to its value plus 5, using the colon operator. For example,

```
>> vecout(4)
ans =
     4     5     6     7     8     9
```

Practice Problem

If the lengths of two sides of a triangle and the angle between them are known, the length of the third side can be calculated. Given the lengths of two sides (b and c) of a triangle, and the angle between them α in degrees, the third side a is calculated as:

$$a^2 = b^2 + c^2 - 2b c \cos(\alpha)$$

Write a script *thirdside* that will prompt the user and read in values for b , c , and α (in degrees), and then calculate and print the value of a with three decimal places. (**Note:** To convert an angle from degrees to radians, multiply the angle by $\pi/180$.) The format of the output from the script should look exactly like this:

```
>> thirdside
Enter the first side: 2.2
Enter the second side: 4.4
Enter the angle between them: 50
The third side is 3.429
```

For more practice, write a function to calculate the third side, so the script will call this function.

Practice Problem

If a random variable X is distributed normally with zero mean and unit standard deviation, the probability that $0 \leq X \leq x$ is given by the standard normal function $\Phi(x)$. This is usually looked up in tables, but it may be approximated as follows:

$$\Phi(x) = 0.5 - r(at + bt^2 + ct^3),$$

where $a=0.4361836$, $b=-0.1201676$, $c=0.937298$,
 $r = \exp(-0.5x^2)/\sqrt{2\pi}$, and $t=1/(1+0.3326x)$.

Write a function to compute $\Phi(x)$, and use it in a program to write out its values for $0 \leq x \leq 4$ in steps of 0.1. Check: $\Phi(1)=0.3413$.

Practice Problem

The Fibonacci numbers are generated by the sequence:

$$1, 1, 2, 3, 5, 8, 13, \dots$$

Can you work out what the next term is? Write a recursive function $f(n)$ to compute the Fibonacci numbers F_0 to F_{20} , using the relationship:

$$F_n = F_{n-1} + F_{n-2},$$

given that $F_0 = F_1 = 1$.

Practice Problem

The first three Legendre polynomials are $P_0(x) = 1$, $P_1(x) = x$, and $P_2(x) = (3x^2 - 1)/2$. There is a general *recurrence* formula for Legendre polynomials, by which they are defined recursively:

$$(n + 1)P_{n+1}(x) - (2n + 1)xP_n(x) + nP_{n-1}(x) = 0.$$

Define a recursive function $p(n, x)$ to generate Legendre polynomials, given the form of P_0 and P_1 . Use your function to compute $p(2, x)$ for a few values of x , and compare your results with those using the analytic form of $P_2(x)$ given above.

Practice Problem

Write your own MATLAB function to compute the exponential function directly from the Taylor series:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

The series should end when the last term is less than 10^{-6} . Test your function against the built-in function `exp`, but be careful not to make x too large—this could cause a rounding error.

Practice Problem

- a. Generate a random sized array of random numbers using:
`x = 10*rand(ceil(10*rand)+2,1)`
- b. Use “for” loop to add up all the values in the array and assign the result to the variable mysum.
 - i. For example, if the array is `x = [1 1 1 1 1 1 1 1 1 2]`, then the sum of all the elements would be `mysum = 11`.
- c. Check your answer using the built-in MATLAB `sum()` function by adding the following code snippet to the end of your script.

```
if mysum == sum(x)
    disp('Congratulations!!, you did it right')
    load handel; sound(y,Fs)
else
    fprintf('Sorry, %.2f ~= %.2f. Please try
again.\n',mysum,sum(x))
end
```