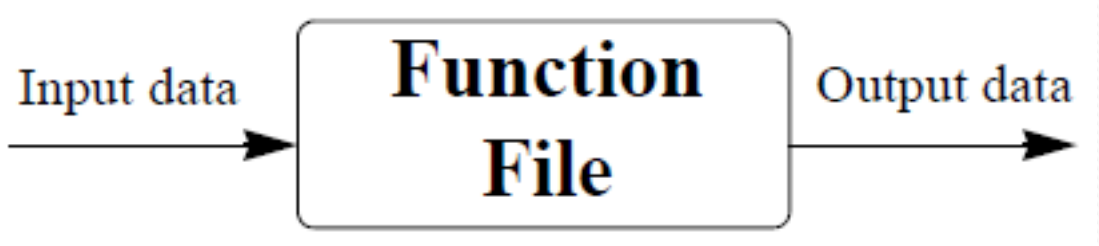
Chapter 7

User-Defined Functions and Function Files

MATLAB An Introduction With Applications, 6th Edition
Dr. Amos Gilat
The Ohio State University

Slide deck by
Dr. Greg Reese
Miami University

A *function* is a MATLAB program that can accept inputs and produce outputs. Some functions don't take any inputs and/or produce outputs.



A function is *called* or *executed* (run) by another program or function. That program can pass the called function input and receive the called function's output

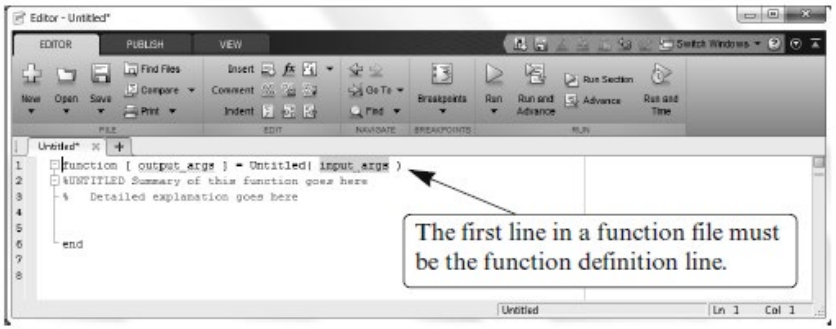
Why use functions?

- Use same code in more than one place in program without rewriting code
- Reuse code by calling in different programs
- Make debugging easier by putting all of one kind of functionality in one place

Code for a function is in an *m-file*

- Can make the file in any text editor but easiest to do it with MATLAB Editor Window

To create a new function m-file, on MATLAB desktop, click on New icon, then select Function. MATLAB opens a window opens that looks like



Can open Editor Window with blank file by clicking New Script icon on MATLAB desktop

Can also open Editor Window from Command Window by typing `edit` followed optionally by a filename not in quotes

- If file in current directory, MATLAB opens it
- If not in current directory, MATLAB creates it

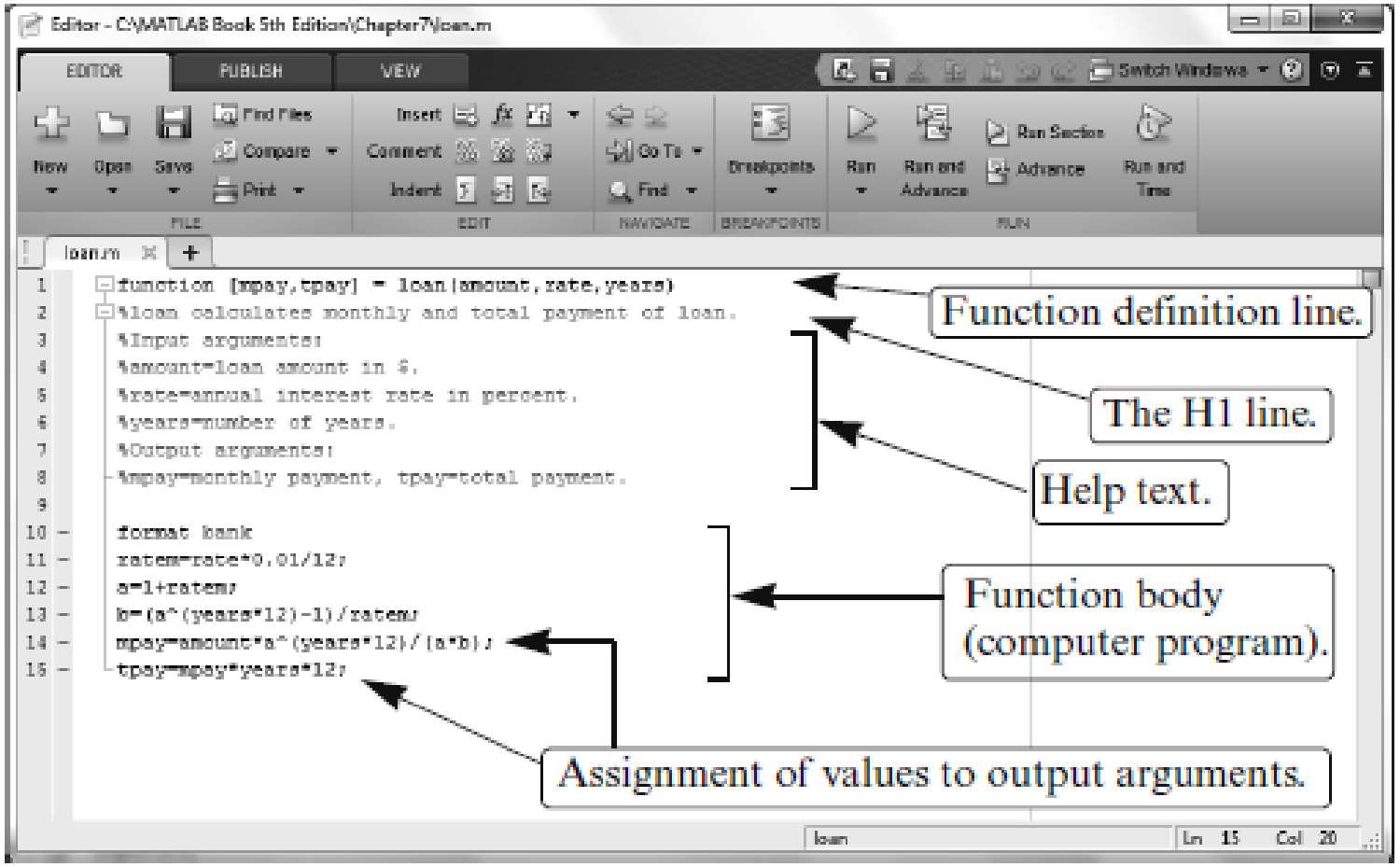


Figure 7-2: Example of a complete function file

Purpose

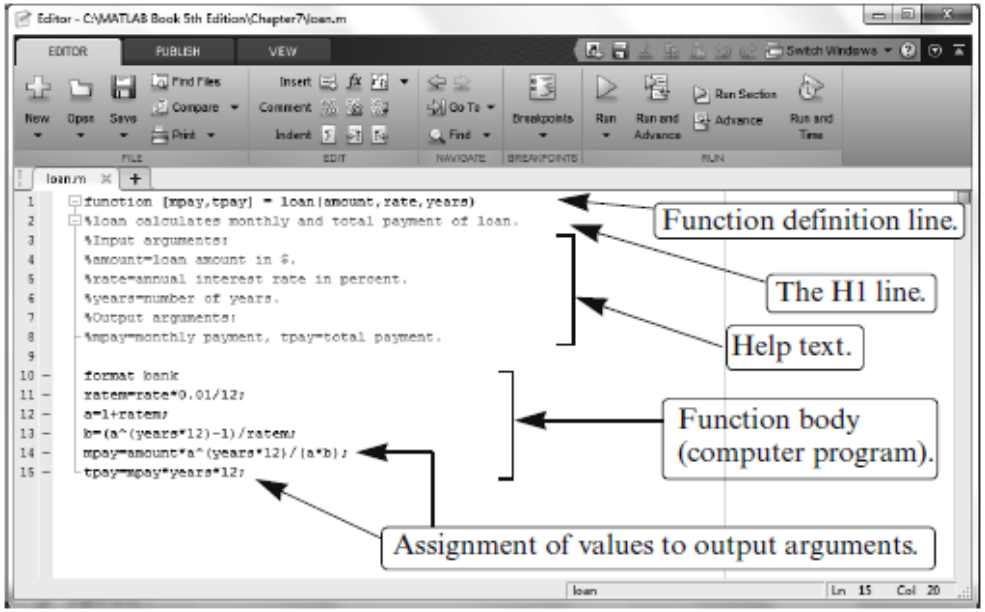
- Calculate monthly and total loan payments

Inputs

- Amount of loan, interest rate, duration of loan

Outputs

- Monthly and total payments



An *executable line* is a line of code that MATLAB can run

- It is a line that is not a comment line, a blank line, or a line with just spaces and tabs

A *function definition line*

- Must be first executable line in function file
  - If not, MATLAB considers file to be a script file
- Defines file as function file
- Defines name of function
- Defines number and order of input and output arguments

Form of function definition line is

```
function [output arguments] = function_name(input arguments)
```

The word "function" must be the first word, and must be typed in lowercase letters.

A list of output arguments typed inside brackets.

The name of the function.

A list of input arguments typed inside parentheses.

Function name

- Made up of letters, digits, underscores
- Cannot have spaces
- Follows same rules as variable names

Avoid making function names that are same as names of built-in functions

Input arguments

- Used to transfer data into function from calling program
- Can be zero or more input arguments
  - Separate multiple arguments with commas
- Can be scalars, vectors, or arrays
- Have values specified by calling program

Output arguments

- Used to transfer data out of function to calling program
- Can be zero or more output arguments
  - In function definition line
    - If zero arguments, can omit assignment operator (=)
    - If only one output argument, can omit brackets around it
    - If multiple arguments, separate them with commas
- Can be scalars, vectors, or arrays
- Function must assign values to all output arguments before it finishes running

Examples of function definition lines

Function definition line	Comments
function[mpay,tpay]= loan(amount,rate,years)	Three input arguments, two output arguments.
function [A] = RectArea(a,b)	Two input arguments, one output argument.
function A = RectArea(a,b)	Same as above; one output argument can be typed without the brackets.
function [V, S] = SphereVolArea(r)	One input variable, two output variables.
function trajectory(v,h,g)	Three input arguments, no output arguments.

H1 line and help text lines are comment lines (ones that start with a percent sign (%))

- Used to provide user with information about function
- Both types of lines are optional

If want name of MATLAB command that does some particular computation, can search for name by specifying a word or phrase likely to describe that computation. Do this with `lookfor` command:

```
lookfor 'word or phrase'
```

- Put word or phrase inside quote marks

EXAMPLE

Suppose want to find command that computes square root of a number. A likely word in description of computation is "square"

```
>> lookfor 'square'
magic - Magic square.
realsqrt - Real square root.
sqrt - Square root.
lscov - Least squares with known covariance.
sqrtm - Matrix square root.
cgs - Conjugate Gradients Squared Method.
```

and many others. From reading descriptions, can tell command you want is "sqrt"

EXAMPLE

A more precise, likely phrase is "square root"

```
>> lookfor 'square root'
realsqrt - Real square root.
sqrt - Square root.
sqrtm - Matrix square root.
```

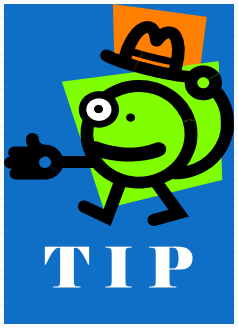
and just a few others

Good news – easy to make `lookfor` work on functions you write, not just MATLAB functions, by using H1 lines

The *H1 line* is the first comment line after the function definition line

- Optional
- Only one line long
- Usually contains function name and short definition of function

`lookfor 'word'` searches in H1 line of all the files that it knows about (including ones you write) for "word". If it finds "word" it prints out file name and H1 line.



When you write an H1 line, put in words that people will likely use in `lookfor`

- Good – uses "square" and "root"

```
% mySqrt(x): computes square root of x using Reese algorithm
```

- Bad – uses "half" and "power"

```
% mySqrt(x): computes x to half power using Reese algorithm
```

`help` command works by displaying H1 line and all following comment lines up to next blank line or code line in function file.

- Will work on files you write too

To make a blank line appear in `help` output, use a line with just a percent sign

EXAMPLE – function file lowpass.m with

```
function y = lowpass( x, fHigh )
% LOWPASS - lowpass filter with specified cutoff frequency
% USAGE: y = lowpass( x, fHigh )
*%
% AUTHOR: Joe Blow
 X = fft( x );
...
```

```
>> help lowpass
LOWPASS - lowpass filter with specified cutoff frequency
USAGE: y = lowpass( x, fHigh )

AUTHOR: Joe Blow
```

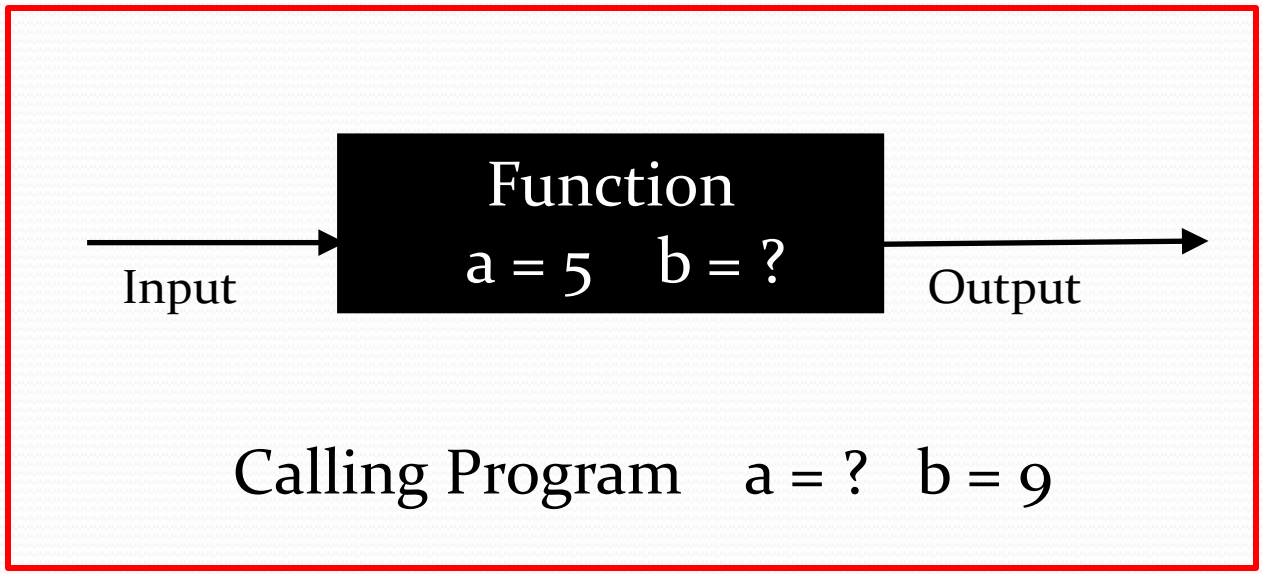
← Blank line produced by line * above

Function body

- Contains code that does computations
- Code can use
  - All MATLAB programming features studied before
  - Special MATLAB commands relevant only to functions

Variables declared inside a function are *local* to that function, i.e., can only be used inside that function

- Calling program can't see (access) any of the variables inside the function
- Function can't see any variables in the calling program



Variables inside and outside function can have same names but are still different variables. Changing one doesn't change other

EXAMPLE – test.m

```
function test( n )
fprintf( 'On entering function n = %d\n', n );
n = 10;
fprintf( 'Before leaving function n = %d\n', n );
```

```
>> n = 5
n = 5
>> test(n)
On entering function n = 5
Before leaving function n = 10
>> n
n = 5
```

← n still 5 after function call

By default, variables are local. To make a variable be recognized among various functions and/or the workspace, you must declare it to be *global*. Do this with the line

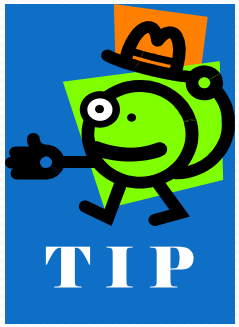
```
global variable_name
```

You can declare multiple global variables in one line by separating with spaces, e.g.,

```
global v1 v2 v3
```

- You must declare variable to be global in every function file that you want to use it in
  - Variable is then common only to these files
- `global` command must appear before the variable is used
  - It's good practice to put command at top of file
- `global` command has to be entered in Command Window and/or in script file for variable to be recognized in workspace

- Variable value can be assigned or changed anywhere it is recognized (declared common)
- It is common to use long descriptive names (or all capital letters) for global variables in order to distinguish them from regular variables



NOTE – use of global variables no longer considered good practice. Avoid them when possible

You must save a file before you can use it

- Click on Save icon
- Strongly recommend to give file same name as function in it but followed by ".m"
- Strongly recommend to use file extension of .m
  - If omit file extension, MATLAB automatically adds .m

Examples

Function definition line	File name
function [mpay,tpay] = loan(amount,rate,years)	loan.m
function [A] = RectArea(a,b)	RectArea.m
function [V, S] = SphereVolArea(r)	SphereVolArea.m
function trajectory(v,h,g)	trajectory.m

User-defined functions called same way as built-in functions

- From Command Window
- From script file
- From another function

To use function its file must be in current folder or in search path

Input arguments can be constants, expressions, or variables (that have values assigned to them)

Input and output passed by calling function assigned in order shown in function definition line,

e.g.,

```
function [month total]=loan(amount,rate,years)
```

EXAMPLE

How much money will I pay monthly and in total between now and when my kid turns 18 on a 6% loan of $10,000?

In Fig. 7-2, function definition line is

```
function [mpay tpay] = loan(amount,rate,years)
```

Example use:

```
>> kidAge = 11;
>> rate = 6;
>> [mpay tpay] = loan( 10000, rate, 18-kidAge )
```

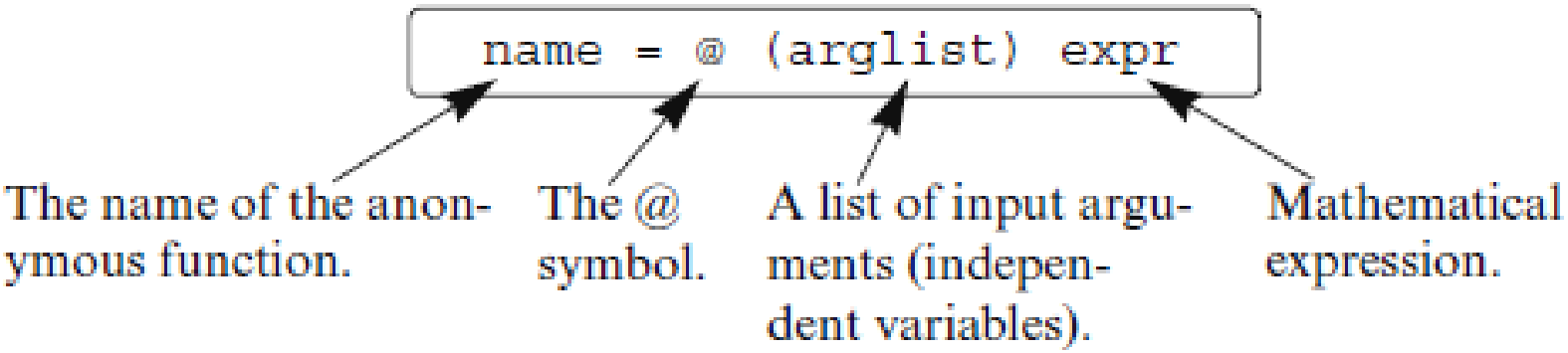
Constant Variable Expression

Characteristic	Function	Script
File extension	.m	.m
First executable line	Function definition line	Any
Variables recognized in Command Window?	No	Yes
Can use variables defined in workspace?	No	Yes
Can accept input arguments?	Yes	No
Can return output arguments?	Yes	No

An *anonymous function* is a function defined without using a separate function file

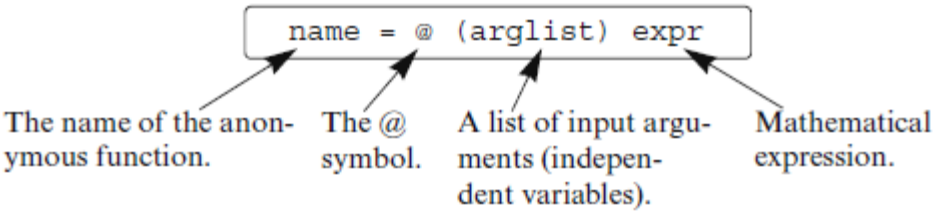
- For short (one line!) functions only
- Usually written by user
- Specified within another function or body of code, not in separate file
- Often used with MATLAB functions that operate on other functions, e.g.,
  - Find area under a function
  - Find derivatives of a function
  - Find optimal value of a function

Can make anonymous function in Command Window, script file, or inside user-defined function. Form is:



For example

```
cube = @(x) x^3
```



- `name` – name of function, using rules for names of user-defined functions
- `@` - a *function handle*, an object that has information about the function
- `arglist` – zero or more function arguments, defined like those for user-defined functions
- `expr` – a single MATLAB expression

Anonymous function

- Can include any built-in or user-defined functions
- Called same way as user-defined functions

```
>> triple = @(n) 3 * n
triple =
    @(n)3*n
>> triple( 4 )
ans = 12
>> x = 1.5;
>> y = triple( x )
y = 4.5000
```

An anonymous function can use in its body any (visible) variables that have been defined before the function itself is defined

Values that function uses are those that the variables have when function is defined (not when it is called)

Problem – variable undefined before anonymous function defined

```
>> p = 5;
>> combo = @(x,y) p*x + q*y;
>> combo(2,8)
??? Undefined function or variable 'q'.
Error in ==> @(x,y)p*x+q*y
```

Solution

```
>> p = 5;
>> q = 4;
>> combo = @(x,y) p*x + q*y;
>> combo(2,8)
ans = 42
```

Anonymous function unaffected by variable change after function defined

```
>> p = 5;
>> q = 4;
>> combo = @(x,y) p*x + q*y;
>> combo( 2, 8 )
ans = 42
>> p = 0
p = 0
>> q = 0
q = 0
>> combo( 2, 8 )
ans = 42
```

EXAMPLE

Write an anonymous function to compute *sin(2πt)* and use the MATLAB command `quad` to find the area under the function from 0 to 1.

```
>> mysine = @(t) sin( 2*pi*t );
>> quad( mysine, 0, 1 )
ans =
      0
```

A MATLAB function that accepts another function as an input is called a *function function* (seriously!)
For example

- `quad` finds area under passed function
- `fzero` finds zeros of passed function
- `fminbnd` finds minimum value of passed function

Function handle:

Function handles used for passing functions to function functions

- Passed functions can be built-in, user defined, or anonymous

To make a function handle for built-in or user-defined function, put the @ symbol in front of function name

- e.g., function handle for MATLAB `cos` function is `@cos`

If going to use function handle more than once, convenient to assign it to a variable, e.g.,

```
>> cosHandle = @cos;
```

Writing a function function that accepts a function handle as an input argument:

To write a function function

- Use any (legal) name for the function handle that is to be passed in
  - Don't put @ in name
  - Name you pick often called *dummy* name
  - Passed function name can be anything, just like passed variable can have any name
  - Example function names: f, fun, F, Func
- In function function, call the passed function same way as normal function, e.g., `Func(x)`

IMPORTANT – document all function requirements so user knows how to write function to be passed

- Inputs - how many, what order, what data type (scalar; row, column, or either vector, array and its dimensions), what valid values
- Outputs - likewise

A name for the function that is passed in.

```
function xyout=funplot(Fun,a,b)
% funplot makes a plot of the function Fun which is passed in
% when funplot is called in the domain [a, b].
% Input arguments are:
% Fun:  Function handle of the function to be plotted.
% a:  The first point of the domain.
% b:  The last point of the domain.
% Output argument is:
% xyout: The values of x and y at x=a, x=(a+b)/2, and x=b
% listed in a 3 by 2 matrix.

x=linspace(a,b,100);
y=Fun(x);
xyout(1,1)=a; xyout(2,1)=(a+b)/2; xyout(3,1)=b;
xyout(1,2)=y(1);
xyout(2,2)=Fun((a+b)/2);
xyout(3,2)=y(100);
plot(x,y)
xlabel('x'), ylabel('y')
```

Using the imported function to calculate $f(x)$ at 100 points.

Using the imported function to calculate $f(x)$ at the midpoint.

Passing a user-defined function into a function function:

Pass user-defined function by using @ symbol followed by function name EXAMPLE

```
function y=Fdemo(x)
y=exp(-0.17*x).*x.^3-2*x.^2+0.8*x-3;
```

```
>> ydemo=funplot(@Fdemo,0.5,4)
ydemo =
    0.5000   -2.9852
    2.2500   -3.5548
    4.0000    0.6235
```

Enter a handle of the user-defined function Fdemo.

Passing an anonymous function into a function function:

Pass anonymous function name without @ symbol

EXAMPLE

```
>> FdemoAnony=@(x) exp(-0.17*x).*x.^3-2*x.^2+0.8*x-3
FdemoAnony =
    @(x) exp(-0.17*x).*x.^3-2*x.^2+0.8*x-3

>> ydemo=funplot(FdemoAnony,0.5,4)
ydemo =
    0.5000   -2.9852
    2.2500   -3.5548
    4.0000    0.6235
```

Create an anonymous function for $f(x)$.

Enter the name of the anonymous function (FdemoAnony).

Using Function Name for Passing a Function into a Function Function

- This technique not used anymore. See book if you have to use it because you have an old version of MATLAB

Can have multiple functions within one function file

- Functions typed one after another
  - First function called the *primary function*
  - Remaining functions called *subfunctions*
- Each function starts with a function definition line
- Name of function file should be name of primary function (followed by .m)

- Every function in a file can call any other function in the file
- Functions cannot access variables inside other functions (unless those variables are declared global)
- Functions outside the file, or scripts, or Command Window can only call primary function

Subfunctions

- Commonly used to make primary function clearer by doing some of its computations
- Commonly not useful on their own or for being called by code other than primary function

EXAMPLE

```
function [me SD] = stat(v)
n=length(v);
me=AVG(v,n);
SD=StandDiv(v,me,n);

function av=AVG(x,num)
av=sum(x)/num;

function Sdiv=StandDiv(x,xAve,num)
xdif=x-xAve;
xdif2=xdif.^2;
Sdiv= sqrt(sum(xdif2)/(num-1));
```

The primary function.

Subfunction.

Subfunction.

```
>> Grades=[80 75 91 60 79 89 65 80 95 50 81];
>> [AveGrade StanDeviation] = stat(Grades)
AveGrade =
   76.8182
StanDeviation =
   13.6661
```

Nested function is a function written inside another function

- Nested function starts with a function definition line and ends with an end-statement
- Function containing nested function must also end with end-statement
- A nested function can contain another nested function, which can contain another nested function, …

One nested function

Format of a user-defined function A (called the *primary function*) that contains one nested function B is:

```
function y=A(a1,a2)
.......
    function z=B(b1,b2)
    .......
    end
.......
end
```

Note end-statements at the ends of B and A

```
function y=A(a1,a2)
.......
  function z=B(b1,b2)
  .......
  end
.......
end
```

Nested function B can access workspace of primary function A and vice versa

```
function y=A(a1,a2)
.......
  function z=B(b1,b2)
  .......
  end
.......
end
```

- Function A can call function B

```
function y=A(a1,a2)
fprintf( 'In function A\n' );

  function z=B(b1,b2)
  fprintf( 'In function B\n' );
  z = b1 + b2;
  end

fprintf( 'Calling B from A\n' );
y = B( a1, a2 );
end
```

```
>> y =  A( 1, 2 )
In function A
Calling B from A
In function B
y = 3
```

```
function y=A(a1,a2)
.......
  function z=B(b1,b2)
  .......
  end
.......
end
```

Function B <u>cannot</u> call primary function A (book is wrong)

```
function y=A(a1,a2)
fprintf( 'In function A\n' );
y = B( a1, a2 );

  function z=B(b1,b2)
  fprintf( 'In function B\n' );
  fprintf( 'Calling A from B\n' );
  z = A( b1, b2 );
  end

end
>> y =  A( 1, 2 )
In function A
In function B
Calling A from B
In function A
In function B
Calling A from B
...
Maximum recursion limit of
500 reached.
```

Two (or more) nested functions at the same level

Format of a user-defined function A that contains two nested functions B and C at the same level is:

```
function y=A(a1,a2)
.......
  function z=B(b1,b2)
  .......
  end
.......
  function w=C(c1,c2)
  .......
  end
.......
end
```

```
function y=A(a1,a2)
.......
  function z=B(b1,b2)
  .......
  end
.......
  function w=C(c1,c2)
  .......
  end
.......
end
```

- Function A can call functions B and C
- Function B can call function C but not function A (book is wrong)
- Function C can call function B but not function A (book is wrong)