MATLAB functions

This is Chapter 5 of the textbook plus 7.4 Function Handles

Disclaimer: *These abbreviated notes DO NOT substitute the textbook for this class. They should be used IN CONJUNCTION with the textbook and the material presented in class. If there is a discrepancy between these notes and the textbook, ALWAYS consider the textbook to be correct. Report such a discrepancy to the instructor so that he resolves it. These notes are only distributed to the students taking this class with A. Gerbessiotis in Fall 2014 ; distribution outside this group of students is NOT allowed.*

8.1 Elements of Program Design.

- **1.** Understand the problem that your plan to solve.
- **2.** Specify the inputs and outputs to the problem
- **3.** Establish the relationship between input(s) and output(s)
- **4.** Design an algorithm that implements the input(s) and output(s) relationship.
- **5.** Determine what data forms (i.e. data structures) you are going to use
- **6.** Write a MATLAB program that implements the proposed algorithm and realizes the proposed data structures.
- **7.** Iterate through steps 4-6 to refine the program (algorithm), the data, and the implementation.
- **8.** Design testing instances to test MATLAB code; use step 3 to figure out important test cases. Fix bugs.
- **9.** Maintenance (for future) releases.

In the realization of step 6 above, guided by steps 3-4 the following actions might need to be pursued.

- **6a.** Split MATLAB program into multiple smaller independent subtasks. Subtasks can be tested independently, and thus can help fixing problem in step 8. Try to have subtasks whose algorithms are known from other projects.
- **6b.** Subtasks that have been encountered before can **reuse** previously developed and test MATLAB code.

Each subtask can be implemented by one or more MATLAB functions which are functional units with their own workspace and variables; functions communicate with the MATLAB program through a **parameter** list. The calling program (or another MATLAB) function instantiates these parameters to specific values (the **arguments**) and these values of the arguments are used within the MATLAB program. Each function **hides** details of implementation plus internal data structures from the calling function or program.

(c) Copyright A. Gerbessiotis. All rights reserved.

MATLAB functions overview.

MATLAB functions come in a variety of flavors. In this subject we will examine the syntax of

- an `anonymous function`,
- a `(primary) function`,
- a `subfunction`,
- a `nested function`,
- a `private function`, and
- a `function function`.

8.1.1 `Anonymous` functions. An `anonymous` function is one that is defined completely in the command window/prompt of MATLAB. The characterization does not mean that a function will not have a name: we need to be able to refer to it somehow. Its definition (where we specify its parameters and the function itself) takes the form of `fname = @ arglist expression`, where `fname` is the name of the function, `arglist` is a list of arguments, separated by commas, and enclosed in parentheses, and an `expression` defines the list of commands/MATLAB functions defining the function.

8.1.1a An `Anonymous` function example. We define function $f(x) = x^2$ as follows `f= @ (x) (x * x)`. (In fact with this definition we allow f to operate on matrices as well, not just scalar values.)

8.1.2 `Primary` (or regular) functions. A `primary function` or simply `function` is visible to MATLAB through its filename that coincides (or supposed to coincide) with the name of the function and ends with `.m`. Thus a call to a primary function aptly named `fname` is resolved by MATLAB to locating a file with filenme `fname.m` in the local directory from which the function was called (other exceptions are discussed at the end of this Subject) and the first function in that file is called. It is strongly suggested (but not required) that this function be named `fname`. So the name of the primary function, gives a hint to MATLAB to look for it in a file with a related filename.

8.1.3 `Subfuctions`. A `subfunction` is defined within a primary function after the primary function's definition and is only visible within the primary function's file.

8.1.4 `Nested functions`. A `nested` function is nested within another function that can include primary, subfunctions and also nested functions (nesting with nesting).

8.1.5 `Private functions`. A `private` function is defined within a directory named `private`, is visible only by functions that can view that directory as it will be explained.

8.1.6 `function functions`. `Function functions` is a function that allows one or more of its parameters to be another function.

(c) Copyright A. Gerbessiotis. All rights reserved.

8.2.1 Script files vs functions. The commands we have written and executed in M-files (i.e. files that end with `.m`) contain one or more MATLAB commands (such as `pwd` or `whos`), operators (such as $+, -$) and functions (such as `length, size, sum`). Sometimes what we call commands are in reality functions or implemented as functions as well. When a script file is executed, the end result is the same as if a programmer typed the code for the command in the script file character by character. The commands in the script file share the workspace with data previously defined and inherit the values to data or code that will run after the script. The only way a script file can communicate with another script file is through the common Workspace; one can leave data in it, that can be read or written into by the other script. In contrast a MATLAB function is a special unit that runs in its own separate workspace and communicates with the `input argument list` and `output argument list`.

8.2.2 `function` definition. A function definition is the place in which the function is defined, i.e. the place where its parameters are listed and the code that implements the task implied by the function becomes available/defined. Communication with other functions or the command window is being done by the `input argument list` such as `inparg1, inparg2, ...` that reads data from the outside environment/world, and the `output argument list` such as `outarg1, outarg2, ...` that transfers/copies data to the outside/environment/world. Other than that a function's workspace is isolated from the workspace of the calling program or function. (There is one exception involving `global` variables.) Below, the return and end statements are in parentheses because they are (in general) optional. (The presence of an `end` is required if a nested function is defined in the file.)

```
function [outarg1 , outarg2 , ... ] = function_name (inparg1 , inparg2 , ...)
  Statement 1;
(return)
(end)
```

8.2.3 `function` name and corresponding filename storing the function definition. MATLAB finds function `fname` by looking for a file named `fname.m` The function is invoked within MATLAB by typing its name (i.e. file name minus the `.m`). MATLAB then understands that it will be looking for a file named `fname.m` to find the code to execute the function; if a file named `fname` (without the suffix) erroneously stores the function, MATLAB will ignore it and report an error instead. (In `fname.m` MATLAB will look for the first function definition and use its code for that of `fname`; it will not necessarily check to confirm that the function is named `fname`.)

(c) Copyright A. Gerbessiotis. All rights reserved.

MATLAB `function` definition continued

8.2.4 `function` syntax. `function [outarg1 , outarg2, ... ] = fname (inparg1, inparg2 , ...)` defines a function `fname` that accepts `inparg1, inparg2,` etc as inputs and returns `outarg1 , outarg2,` etc as outputs. The output variables can be the same as the input variables. Functions use their own local variables that exist while the function is active and die when control exits from the function. (An exception is `global, persistent` variables that are explicitly defined.) Function `fname` can be expressed in terms of existing functions that include built-in MATLAB (primary) functions (i.e. `length, sum, size`), nested, subfunctions etc. They all reside within an `M-file`.

A function name always start with an alphabetic character. Any variable defined within the body of the function is a local variable unless explicitly defined to be otherwise (e.g. global). One can define within a function, subfunctions that are only visible within the function in which they are defined, and invisible to outside programs/functions or have nested functions nested within the primary function.

A function can be terminated with an `end` statement but this is (mostly) optional. If however nested functions are used, then the presence of an `end` is required not just for the nested functions but for every function defined within the file.

A function returns when the end of the function is reached; the end of the function can be (i) an explicitly stated `end`, (ii) the last executed statement before the end of the file is reached or before another function begings, or (iii) an explicit `return` statement that forces the completion of the function.

In the example below `abs1, abs2` are defined as single function with one parameter x returning one value y, the absolute value of x. Both functions have an `end` statement, only the latter uses a `return`. In the former case the function terminates when it reaches the function `end` statement. In the latter case for $x < 0$ the function terminates at the `return` before reaching the `end` of the function. Function `abs3` is another incarnation of the absolute function. What can you say about `abs4`?

```
function y=abs1(x)
  if (x<0)
    y=-x;
  else
    y=x;
  end
end
```

```
function y=abs2(x)
 y=x;
 if (x<0)
    y=-x;
    return;
 end
end
```

```
function y=abs3(x)
 if (x < 0)
   y=-x;
   return;
 end
 y=x;
 end
```

```
function y=abs4(x)
% Does it work?
 if (x<0)
   y=-x;
 end
 y=x;
end
```

(c) Copyright A. Gerbessiotis. All rights reserved.

8.2.5 Calling a function. A function can be called by invoking its name followed by a parenthesized list of its arguments. If a function has no parameters we can have an empty list or use no list at all. Thus `h` or `h()` might call a function with no parameters that return no values but has only `side-effects`. The side-effect is the printing of the message `Hi`.

```
1 function h
2  disp('Hi');
3 end
```

For a function with no empty parameter list, we can call it by listing the values that these parameters will take. For function `f` that squares its input argument we can call it using $f(10)$ to get 100. If no return value is specified by the caller in MATLAB's command window, the return value will be assigned to `ans`. We can explicitly provide a return value by writing for example `z=f(10)`. In that case the value 100 returned will be assigned to the command window (local) variable z. If we define a variable $y = 10$ in the command window, we could also call it as in `z=f(y)` with the same end result.

```
1 % Hi.. This is my function f
2 function y=f(x)
3  y=x*x;
4 return
5 end
```

8.2.6 `H1` comment line. The first comment line in a MATLAB `M-file` (i.e. line 1 in `f.m`) is called the `H1` line. It is used by the MATLAB `help` system to retrieve helpful information about the function. Thus that line all the way until the first blank line are printed if a user invokes `help f`. In our case we get.

```
>> help f
  Hi.. This is my function f
```

8.2.7 `function` input/output parameters. The input and output parameters are used in defining the function. During the invocation these are instatiated (i.e. accept or return values) by the input and output argument. The input arguments can be scalars, vector, matrices or strings. So an input can be a scalar in one function call, and a vector in another one.

8.2.8 Understanding the argument possibilities in MATLAB. We define a MATLAB function `sub8addt` that takes as input two arguments (i.e. it has two parameters) and returns their sum. This is a very simple form of a function in which no output variable is defined. Try to call this functions with a variety of input variables as parameters: scalars, row vectors, mixed row and column vectors, matrices, strings of varying length, and strings of equal length. Understand why several cases work just fine and what value they return and also why some do not work.

```
function sub8addt(x,y)
  x+y
end
```

```
x=10; y=20
sub8addt(10,20);
x=[1 2]; y= [3 4];
sub8addt(x,y);
x=[1 2]; y= [3; 4];
sub8addt(x,y);
x=eye(4,4);y= -2*eye(4,4);
sub8addt(x,y);
x='Computer'; y='Science'
sub8addt(x,y);
x='Computer'; y='Science '
sub8addt(x,y);
```

(c) Copyright A. Gerbessiotis. All rights reserved.

2.9 Parameter passing by `value` in MATLAB. When we call $f(y)$ in line 2 below, MATLAB performs a number of steps to evaluate f at y and return a value. These include

- The argument of f is evaluated i.e. y is evaluated and its value 10 retrieved.
- Function f becomes active by activating and initializing its (input and output) parameters through the supplied arguments of the calling function. Since f has only one parameter x, this is the only variable that is defined. Variable x is a local variable to f; it exists within f, is becomes active the moment f is called and execution starts, and dies when f completes its execution. The value of x is copied from the calling environment (i.e. the command window). Thus the value of y of the command window is copied to become the value of x. Thus the local copy x has value 10. In addition a local variable y is activated; this is to hold the return value of f.
- The lines of f are then executed and local variables are updated as needed. For example y will hold the square of x i.e. y is set to 100.
- When execution terminates for f, the value of the return variable(s) is retrieved. Since the return variable is y its value 100 is retrieved. This value is the returned as the value of $f(y)$ and assigned to z. (If the call was $f(y)$ rather than $z = f(y)$ the return value would have been assigned to `ans` of the command window.)
- As soon as the return value is assigned to z function f terminates and variables x, y are destroyed and their contents erased. Thus a `x` to print x will generate an error (printed in red). A `y` will print the contents of the y defined in the command window not of function f.

```
>> y=10;
>> z=f(y);
>> z
>> x
??? Undefined function of variable 'x'
>> y
y = 10
```

Note. During the execution of the second line there are two variables y, z defined in the command window and two variables x and y defined in function f. The y of the former is different from the y of the latter. When the semicolon of line 2 is reached (i.e. 100 is copied to z) the latter y has ceased to exist (and so did the x). But the z, y of the command window are still there. In fact the y of the command window retains its original value.

8.2.10 more on functions. With reference to the same function f we perform know the following. This will convince you of the differences between the x, y of the command window and those of function f. In the last invocation the argument to f is not a scalar but a 2×2 matrix.

```
>> y=10;
>> x=f(y);
>> x
 x = 100
>> y
y = 10
>> f(y+y)
>> ans = 4000
>> f([ 1 1 ; 2 2])
>> ans = 3  3
         6  6
```

(c) Copyright A. Gerbessiotis. All rights reserved.

8.3.1 Example (Fibonacci) numbers. We rewrite the Fibonacci number example by using an independently callable function. We define a `sub8fibonacci` function that takes as input `n`, the index of the Fibonacci number that will be computed, and returns a value `value` which is the value of the n-th Fibonacci number. Given that the name of the function is `sub8fibonacci` we store this function in a file named `sub8fibonacci.m`. We will test this function by using a separate script file (no function definition inside) called `sub8testfibonacci.m`. The two files are shown below.

```
% This is sub8fibonacci.m
% sub8fibonacci (n)
function [value] = sub8fibonacci ( n )

f=zeros(1,n);
f(1)=1; f(2)=1;
for index = 3:1:n ;
 f(index)= f(index-1)+f(index-2);
end
value = f(n);
end
```

```
% This is sub8testfibonacci.m
n=input('Index of Fibonacci number = ');
value = sub8fibonacci(n);
fprintf('Fibonacci(%d) = %d \n',n,value);
```

8.3.2 Comment line. The result of `help sub8fibonacci` is shown below.

```
>> help sub8fibonacci
  This is sub8fibonacci.m
  sub8fibonacci (n)
```

(c) Copyright A. Gerbessiotis. All rights reserved.

8.3.3 Calling the function. There are two ways to call the `sub8fibonacci` function. One is through a prepared script. Thus below, by writing `sub8testfibonacci` we call the script `sub8testfibonacci.m` that resides in the directory from which we issued the `matlab` command. The script runs the commands of the three lines related to input, fibonacci and fprintf. It recognizes `sub8fibonacci` as a used specified function because it is not a MATLAB function; it finds a file `sub8fibonacci.m` with a same-named function inside it in the same local directory, and runs that function with the value provided by the user through the `input` function. The result is returned with the `fprintf` statement. Another alternative is to call `sub8fibonacci` directly by writing within MATLAB something like `x=sub8fibonacci(12)`, where the value returned is assigned to `x`.

```
>> sub8testfibonacci
 Index of Fibonacci number = 10
Fibonacci(10) = 55
>> x= sub8fibonacci(12)
x = 144
```

(c) Copyright A. Gerbessiotis. All rights reserved.

8.4.1 A Matlab file can contain data and usually have `.mat` suffixes, or can contain programs and functions: for this latter case we call such files `M-files`, because the file-name's suffix is `.m`. MATLAB `M-files` that are `script files` contain a sequence of MATLAB functions and commands. These are executed by calling from within MATLAB the file-name without using the suffix `.m`. Thus if a script file `filename.m` exists we can just call it from within MATLAB by writing `filename`. **NOTE that if `filename` only exists and from within MATLAB you call `filename` MATLAB will report an error because MATLAB would be looking for filename `filename.m`.** When MATLAB sees function `filename` and thinks it's a script file or a function it automatically adds the suffix `.m` and looks by default for `filename.m`. If you want to call a function `fibonacci`, then in the same location that you would store script files, you must also have a file named `fibonacci.m` with the `.m` suffix. Thus `fibonacci(12)` tells MATLAB that `fibonacci` is a function; then MATLAB uses a special sequence of rules to resolve this name that could lead to reading file `fibonacci.m` in which MATLAB expects to find a definition of a function. The first function in that file will be considered to be `fibonacci` even if it has a different name.

8.4.2 Default location of `M-file`. The default location that MATLAB looks for user specified files is the directory in which the user was located when the `matlab` command was issued from the command-line to start MATLAB and open a command window. If a user used a windows-based environment (windows might mean Windows as in Microsoft Windows but also X-Windows, or Apple OS X) then the default location is OS specific: for examples in Microsoft Windows the user might create a `shortcut` to run MATLAB and in that shortcut the user might describe the `Target` location for the `matlab.exe` executable. However the user might also specify that MATLAB `Start in:` some-other user-specified location, where the user's `.m` files are or will be located.

8.4.3 Changing location. The simplest way to call a function f located in some other place such as in directory `/afs/cad.njit.edu/u/UCID/dir1` is to go the by typing in the command window `cd /afs/cad.njit.edu/u/UCID/dir1`. Then the function can be invoked direcly by its name. Function/command `cd` also explained in Subject 0 is a UNIX/Windows command to `change directory`. To confirm location you can type in the command window `pwd` to print the current working directory, or `ls` to list the directory entries (files or subdirectories). The contents of an M-file `f.m` can be viewed by typing `type f.m`.

(c) Copyright A. Gerbessiotis. All rights reserved.

Changing the MATLAB location of `.m` files continued

8.4.4 Using `path` to invoke an `M-file` stored in a non-default location. The alternative to **8.4.3** is to reset the `path` variable in MATLAB that directs MATLAB to the (more than one) default locations for retrieving MATLAB related file (eg `M-files`). Thus the following might work as an alternative to the method in 8.4.3 of the previous page.

```
>> oldpath =  path;
>> path(oldpath, '/afs/cad.njit.edu/u/UCID');
>> filename
```

8.4.5 Using the `MATLABPATH` environment variable to invoke an `M-file` stored in a non-default location. On a UNIX seystem one can issue from the command line (i.e. from within a window terminal before the user invokes MATLAB from that terminal) a `setenv MATLABPATH /afs/cad.njit.edu/u/UCID`

In fact, the user can append several different paths (not just one) as long as the user separates the different paths with a colon `:` (as opposed to Microsoft Windows that uses a semicolon `;` UNIX systems use `:` for separate directory entries in path variables).

(c) Copyright A. Gerbessiotis. All rights reserved.

8.5.1 Local variables. In MATLAB functions, variables inside a function have their own workspace i.e. they are **local variable**. Consider the function `sub8testfunction` shown below.

```
function  x  = sub8testfunction(a)
% sub8testfunction(a)
  x=1;
  fprintf('1: a= %f  x=%f \n',a,x);
  y=sub8testnext(x);
  fprintf('2: a= %f  x=%f \n',a,x);
  x=y;
  fprintf('3: a= %f  x=%f \n',a,x);
end

function x= sub8testnext(x)
  x = x+20;
  fprintf('4: x=%f \n',x);
end
```

If one calls it by issuing a `sub8testfunction(20)`, one can observe that within `sub8testnext` its local copy of x changed its value to 21, yet the local copy of `sub8testfunction` still retains the 1 value, until it gets changed with the `x=y` statement.

```
>> sub8testfunction(20)
1: a= 20.000000  x=1.000000
4: x=21.000000
2: a= 20.000000  x=1.000000
3: a= 20.000000  x=21.000000
```

(c) Copyright A. Gerbessiotis. All rights reserved.

8.6.1 Subfunctions. (Primary) function `avg` defined in file `avg.m` return the average value of the elements of a vector x. (If x happens to be a scalar value, it returns that value itself.) Function `avg` call two built-in MATLAB functions `sum, length` to determine the sum of the values of the elements of x and the number of elements of x respectively. In `avg1.m` we define our own **subfunction** `mysum` to find the sum of the values of the elements of x. Function `avg` calls the subfunction `mysum` and function `length`. Note that neither of them includes an `end` statement. The primary function ends before line 3 and the subfunction start with line 4 and end by the end of file which is after line 5.

```
1 function y=avg(x)              % avg.m
2   y=sum(x)/length(x);
3 end
```

```
1 function y=avg1(x)             % avg1.m
2   y=mysum(x)/length(x);
3
4 function y=mysum(x)            % subfunction mysum
5    y=sum(x);
```

8.6.2 Nested functions. In `avg2.m` we perform the same operation execpt that instead of calling `sum, length` we call two **nested functions** `mysum, mylen` that respectively call `sum, length`. As one can observe all three functions of `avg2.m` are ended with an `end` statement.

```
1 function y=avg2(x)             %avg2.m
2   function z= mysum(x)
3       z= sum(x) ;
4   end
5   function z=mylen(x)
6       z= length(x) ;
7   end
8   y=mysum(x)/mylen(x);
9
10 end
```

(c) Copyright A. Gerbessiotis. All rights reserved.

8.6.3 Nested functions and subfunction with the same name. A nested function may have the same name as a subfunction defined in the same context (file). **A nested function always have priority over a subfunction.** Thus in `avg3.m` in function `avg3` the call to `mysym` in line 9 refers to the nested function of lines 2-5, not to the subfunction of the same name.

```
1  function y=avg3(x)              % avg3.m
2    function z= mysum(x)
3         z= sum(x) ;
4         fprintf('You are in the nested funtion\n');
5    end
6    function z=mylen(x)
7         z= length(x) ;
8    end
9    y=mysum(x)/mylen(x);
10 end
11 function z=mysum(x)
12   z=sum(x);
13   fprintf('You are in the subfunction\n');
14 end
```

8.6.4 Nested functions and variables. A nested function can access all the variables of the function in which it is nested. This is shown in `avg4` that access to variable `y` allows us to compute the average without using return values at all. At the same time all variable of nested functions are readable/writeable by the function(s) that enclose them

```
1  %The file is avg4.m but the function name is avg3 : You are looking for TROUBBBBBBLE
2  function y=avg3(x)
3    function z= my2sum(x)
4         z= sum(x) ;
5         fprintf('You are in the nested function mysum\n');
6    end
7    function z=mylen(x)
8         z= length(x) ;
9         fprintf('You are in the nested function mylen\n');
10   end
11   y=mysum(x)/mylen(x);
12
13 end
14
```

```
15 function z=mysum(x)
16    z=sum(x);
17    w=mylen(x);
18    fprintf('You are in the subfunction mysum\n');
19 end
```

A subfunction defined in `avg4.m` would not be able to call `mysum, mylen`.

8.6.5 Multiply Nested functions. When functions are nested as shown below, certain nested functions can or can not call other functions. One can call a nested function

- from a function one level above it (i.e. A can call B, D but not C, E),
- from a function with the same parent and the same nesting level (i.e. B and D can call each other),
- from a function at a lower level (C can call D and B but not E).

```
function A()           % file A.m
   function B()
      function C()
      end
   end
   function D()
      function E()
      end
   end
end
```

```
function A()
   function B()
      function C()
        fprintf('------->C\n');
        D();
      end
     fprintf('----->B\n');
     C();
   end
   function D()
      function E()
        fprintf('------->E\n');
      end
     fprintf('----->D\n');
     E();
   end

   fprintf('->A\n');
   B();D();
end
```

8.6.6 Nested functions and subfunction. Any nested function can call a subfunction of the same file.

```
function y=avg5(x)
  function z= zvg5(x)
      z= sum(x) ;
      fprintf('You are in the nested function mysum\n');
  end
  function z=mylen(x)
      z= length(x) ;
      fprintf('You are in the nested function mylen\n');
  end
  y=zvg5(x)/mylen(x);

end

function z=zvg5(x)
  z=sum(x);
  w=mylen(x);
  fprintf('You are in the subfunction mysum\n');
end
```

Collision resolution rule. If a function f is being called, an attempt is first being made to determine whether a nested function with that name is available. If the answer is yes, that nested function is called. Otherwise a subfunction is searched with name f. If the answer is yes (a subfunction exists), that subfunction is called. Otherwise a private function with the same name can be called, before a primary function is attempted. A complete sequence of the steps involved in resolving a function name is given at the end of this Subject. We present next `private functions.`

(c) Copyright A. Gerbessiotis. All rights reserved.

8.6.7 Private functions. Private functions are functions that reside in subdirectories with the special name `private`. They are visible to other functions also located in the `private` directory or to functions in the parent `directory` of `private` and those latter function are defined in an M-file. In addition, such functions can call not only the private function but also scripts calling the private function.

With reference to the example below, function `apriv` have access to a directory `private` that contains `bpriv`. Functions `apriv` calls `bpriv`. M-file `cpriv.m` contains a sequence of calls to `apriv, bpriv` but it is not a function. Each M-file involved prints a message that informs about its function name.

```
 apriv.m                          % calls bpriv
 bpriv.m                          % A primary function; prints a message and terminates
 cpriv.m                          % Script file calling bpriv
 dpriv.m                          % calls script file cpriv
 private
       bpriv.m                   % prints a message and terminates
 adir
    a.m                           % calls b,c
    b.m                           % calls a,c
    private
      b.m                         % prints a message and terminates
      c.m                         % calls b
 bdir
```

```
>> apriv
This is apriv. It will call bpriv
This is function bpriv in directory private
>> bpriv
This is  primary (NOT PRIVATE) function bpriv coexisting with apriv cpriv.
>> cpriv
This is script cpriv. It will call bpriv
This is  primary (NOT PRIVATE) function bpriv coexisting with apriv cpriv.
>> dpriv
This is dpriv. It will then call script cpriv
This is script cpriv. It will call bpriv
This is function bpriv in directory private
```

(c) Copyright A. Gerbessiotis. All rights reserved.

apriv From the command window we call `apriv`. Function `apriv` is defined in a file and calls `bpriv`. By following the definition of a private function, this is the default way to call a private function. Even though there is a primary function in the same directory with `apriv` the (private) function `bpriv` in directory `private` is being called.

bpriv We call `bpriv` from the command window. We can not call this way a private function. Thus the function called is the primary function `bpriv` co-located with `apriv` and `private`.

cpriv We call script `cpriv` from the command window that calls `bpriv`. A script can call a private function only if it is being called by a function such as `apriv` that is defined in an M-file. Script `cpriv` is called from the command window and thus the `bpriv` invocation in the script results in the primary function `bpriv` being called.

dpriv We call function `dpriv` defined in M-file `dpriv.m` that calls a script file `cpriv.m` that call a function `bpriv`. This is the way to call a script that can call a private function. In this case `cpriv` is executed and its call to `bpriv` resolves by calling the private function `bpriv` not the primary one (as it was the case in the previous case).

```
>> cd adir
>> a
This is a.m in adir
This is b.m in adir/private
This is c.m in adir/private
This is b.m in adir/private
>> b
This is b.m in adir
This is a.m in adir
This is b.m in adir/private
This is c.m in adir/private
This is b.m in adir/private
This is c.m in adir/private
This is b.m in adir/private
>>cd private/
>>c
This is c.m in adir/private
This is b.m in adir/private
```

(c) Copyright A. Gerbessiotis. All rights reserved.

With reference to the script of the previous page we have

a After getting into `adir` function `a` of `a.m` is called. Function a call prints the first line and then calls b, c. Function b co-located with a calls a, c, but function b in private only terminates. Which of the two b is called by a? The call to b by a is to the private function b not the primary function. This prints the second line. Then the call to c is also to the private function c in `private`, that calls a b. For c the b in `private` is just a primary function. This results in the fourth line being printed.

b A direct call from the command window to b can not call the private function b. It calls the primary function b i.e. of file `adir/b.m` that calls a and c. This is indeed so by the first line printout. The remaining 4 lines are those printed when a was called, and are the result of the invocation of function a. Finally c gets called after a which prints the last two lines (lines 6-7) similarly to the previous case.

c The invocation to c prints the two lines shown which were also printed similarly when `a` or `c` were called.

The intent of using private functions is to create for example our own separate copies of existing functions that can be used for testing etc. **NOTE that MATLAB first looks for private functions and then for other possibly same-named M-file functions.**

(c) Copyright A. Gerbessiotis. All rights reserved.

8.7.1 Odd-even transposition sort. We present a very simple, naive, and highly inefficient sorting algorithm that performs close to n^2 comparisons to sort n keys arranged as vector elements. However the algorithm is very elegant by its simplicity. It has a number of n rounds. The "odd" ones (i.e. indexed $1, 3, 5, 7$) and the "even" ones (i.e. indexed $2, 4, 6, 7$). In an odd-round an odd indexed key is compared to the key on its right (an even indexed one); if the former is greater than the latter they get swapped. An even round works the same way. In each one of the n rounds, roughly $n/2$ comparisons are performed. Thus overall approximately $n \times n/2$ comparisons are performed. We call this algorithmic a `quadratic` running time algorithm for that

```
function  x= sub8oddeven ( x)
 n = length(x);
 for ii= 1:n ;
  if mod(ii,2)==1
    for jj=1:2:n-1
        if (x(jj) > x(jj+1))
            temp=x(jj);
            x(jj)=x(jj+1);
            x(jj+1)=temp;
        end
    end
  else
    for jj=2:2:n-1
        if (x(jj) > x(jj+1))
            temp=x(jj);
            x(jj)=x(jj+1);
            x(jj+1)=temp;
        end
    end
  end
end
```

8.8.1 Within a function a sequence of MATLAB functions can be called that return information helpful for the function itself. Such functions include.

- `nargin` returns the number of actual input arguments that were used/provided to call the function. If a negative number is returned, this means that the function can accept a variable number of arguments.
- `nargout` returns the number of provided returned values for the function being called.
- `nargchk` that validates the number of input arguments provided.
- `nargoutchk` that validates the number of output arguments provided.
- `error` that prints an error message and aborts the function from which it is called.
- `warning` displays a warning message/string and continues execution in the function from which it is called.
- `inputname` returns the actual name of the variable that correspond to the particular argument number.

Note. Function `nargchk,nargoutchk` are usually used in conjunction with `error`.

```
function [mean n] =sub8narg(x)
  fprintf('Number of arguments/parameters used by caller   is %d\n',nargin);
  fprintf('Number of return values used by caller          is %d\n',nargout);
  n=length(x);
  mean=mysum(x)/n;
end

function z=mysum(x)
  z=sum(x);
end
```

The following example script shows the different output for `nargout` depending on the number of return variables provided by the calling program (respectively 0,1, and 2).

```
>> x=[10 20 30];
>> sub8narg(x)
Number of arguments/parameters used by caller  is 1
Number of return values used by caller         is 0
>> a= sub8narg(x);
Number of arguments/parameters used by caller  is 1
Number of return values used by caller         is 1
>> [a b] = sub8narg(x);
Number of arguments/parameters used by caller  is 1
Number of return values used by caller         is 2
```

```
1  function [mean n] =sub8nargc(x)
2    fprintf('Number of arguments/parameters used by caller is %d\n',nargin);
3    fprintf('Number of return values used by caller          is %d\n',nargout);
4    msg=nargchk(1,1,nargin,'string');
5    if length(msg) ~= 0
6      msg='Incorrect number of input argument\n';
7      warning(msg);
8    end
9    error(nargoutchk(1,2,nargout,'string'));
10   if (nargin >0)
11     n=length(x);
12     mean=mysum(x)/n;
13   else
14     n=0;
15     mean=0;
16   end
17 end
18
19 function z=mysum(x)
20   z=sum(x);
21 end
```

(c) Copyright A. Gerbessiotis. All rights reserved.

With reference to the previous page and `sun8argc` this is similar to `sub8narg` except that `nargin, nargout` are checked with functions `nargchk, nargoutchk` respectively. For `nargin` we check whether the number of provided input arguments is at least 1 and at most 1 (i.e. exactly 1). If this is so `msg` is empty and lines 5-8 are skipped. Otherwise a warning is printed through lines 6-7.

For `nargout` we check whether the number of return variables provided is at least 1 and at most 2. For any other case an error will be generated. The `'string'` can be omitted from both `nargchk, nargoutchk` since this is the default behavior.

8.8.2 Optional arguments. If one uses these special functions from within a function, the number of arguments to the function can even vary. By using `nargin` for example one can check how many arguments are present and thus set default values to the non present values. Of course in such a case one needs to know which argument is missing; thus a default mechanism needs to be assumed. This how functions such as `zeros(), zeros(n), zeros(n,m)` operate.

(c) Copyright A. Gerbessiotis. All rights reserved.

Variables in MATLAB

There are three main types of variables in MATLAB.

Local variables such as those defined in a function and the command window. A local variable exists as long as the command window is open, or a function is executing its code. As soon as the window closed or the function terminates the local variables cease to exist. In a command window the defined local variables can be checked with a `who, whos` and cleared (i.e. force the variable to cease to exist) by using `clear all`, `clear` or simply `clear variable` where `variable` is the name of the variable that will be erased.

Global variables can be defined in a function or the command window and can be accessed (read/written) from that function or other functions/command window.

Persistent variables can only be defined within a function and persist's that function's demise, i.e. even if a function terminates, the value of a persistent variable is retrievable and modifiable.

Note. A global or persistent variable needs to be **explicitly defined** using a keyword `global` or `persistent` as needed.

(c) Copyright A. Gerbessiotis. All rights reserved.

8.9.1 Global variables. One or more global variables in a MATLAB function is defined by writing `global var1 var2 var3;` to declare a global statement and thus define a set of global variables `var1, var2, var3`. The same `global` prefix can declare more than one variables that are separated by white space. If a global variable is defined it can be accessed from any workspace, not only that of the function's in which it was defined but also from any other function, and also from the workspace of MATLAB's command window.

For that to occur one needs to issue a `global` to create a connection from say the command window to the global variable. Global variables accessible through the command window can be cleared with a `clear` command; however this does not erase them completely, it only destroys the connection from the command windows to the global variable. One needs to use `clear global` or `clear all` to permanently erase global variables

```
function y=sub8g(x,y)
  global g1 g2;

  if (nargin == 2)
   g1=x;
   g2=y;
   fprintf('g1=%d g2=%d \n',g1,g2);
   return;
  end
  if (nargin == 1)
   g2= x;
   fprintf('g1=%d g2=%d \n',g1,g2);
  else
   g1=-1;
   g2=-1;
   fprintf('g1=%d g2=%d\n',g1,g2);
end
```

Global variables: example of usage

```
>> sub8g()                    % initialization call
g1=-1 g2=-1
>> whos global                %
   g1   1x1  8 double array (global)
   g2   1x1  8 double array (global)
>> global g1 g2               %  connect to g1 g2
>> g1                         %  print/read g1
g1 = -1
>> sub8g(100,200)             %  update through a function call (from withing sub8g)
g1=100 g2=200
>> g1=1;                      %  change g1 directly from command window
>> sub8g(9)                   %  change g2 from sub8g
g1=1 g2=9                     %  Verify changes
>> clear                      %  Destroy connection (global g1 g2 still exist)
>> g1
??? Undefined function or variable 'g1'.
>> global g1 g2               %  Reconnect
>> disp(g1), disp(g2)
   1
   9
>> clear global               %  Erase g1 g2 ; Also you can use clear all
>> global g1 g2
>> g1
   g1 = []                    %  Recreated in the command window not reconnected
```

Note. One can clean global variables with a `clear global` or `clear global var1` , if `var1` is he name of a global variable. A `clear all` is more general and will clear local variable and global variables (and more).

Note. After the connection global variable are visible with `whos`. Before the connection global variables can be viewed with a `whos global` only.

(c) Copyright A. Gerbessiotis. All rights reserved.

8.10.1 Persistent variables. Local variables within a function have their contents erased as soon as the function ends and returns the value(s) computed. Local variables can be preserved by using a `persistent` declaration (similar to a global declaration).

With respect to function `sub8p` defined in `sub8p.m`, we create two persistent variables `pers, pers2`. In `pers2` we will maintain how many times the function is called with a non empty numeric input argument, and in `pers` we copy that argument. Intialization occurs with an empty function call, i.e. `sub8p()` that also resets to 0 `pers`.

Note. Persistent variables can be cleared with a `clear all` or `clear functions`.

```
% First time you need to call sub8p() to set pers2 to 0
function y=sub8p(x)
 persistent  pers pers2
 if (nargin >0)
  pers =x;
  pers2=pers2+1;
 else
  pers =0;
  if (pers2 > 0)
   pers2=pers2;
  else
   pers2 =0;
  end
 end
 fprintf('pers=%d pers2=%d \n',pers,pers2);
end
```

Persistent variables continued

```
>> sub8p()    %initialize
pers=0 pres2=0
>> sub8p(10);
pers=10 pers2=1
>> sub8p(11);
pers=11 pers2=2
>> sub8p(12);
pers=12 pers2=3
>> sub8p()
pers=0  pers2=3
>> sub8p()
pers=0  pers2=3
>> sub8p(9)
pers=9  pers2=4
>> clear functions    % or clear all
```

(c) Copyright A. Gerbessiotis. All rights reserved.

8.11.1 Random numbers. In the attached code, function `sub8random0` generates random values in the range $(0, 1)$. A global variable `iseed` is used to seed the random number generator, i.e. "to start this (pseudo) random process". This variable `iseed` is a global variable. Another function `seed` sets this `seed` variable. A user can always set or reset the `seed` variable. Function `seed` performs the work outlined on page 230 of the textbook. The last four values of the second invocation of `sub8random0(1,10)` are the same as the first row of the second invocation of `sub8random0(4)` on page 232.

```
function ran= sub8random0(m,n)
% Generate a random number in [0,1]
% random0(m)    mxm array
% random0(m,n) mxn array
global iseed;

msg=nargchk(1,2,nargin);
error(msg);
if nargin < 2
  n=m;
end
ran=zeros(m,n);
for ii = 1:m
  for jj = 1:n
     iseed = mod(8121*iseed+28411,134456);
     ran(ii,jj) = iseed/134456;
  end
end
end
```

(c) Copyright A. Gerbessiotis. All rights reserved.

If we run an experiment that uses random values and then repeat and expect the same values to be generated, we need to reset generator by resetting its `iseed` value. You might have observed that we call this random number generator `pseudo random`. The reason is that the values generated are deterministically "determined" in lines 19 and 20 of `sub8random0`.

8.11.2 The `sub8seed`. The `sub8seed` function is straightforward. It is more useful to explain the example run below. First we need to set the value of the global variable `iseed` through the `sub8seed` function. We set it to `1024`. Then we fill a vector of length 10 with random values. Note that these values are the same as the one of the textbook of page 232, if we read the values of the 4×4 matrix there in row major form. In the fourth function invocation, we reset the seed and the same values can then be generated.

```
1 function  sub8seed(m)
2 % Seed for sub8random0
3 % sub8seed(m) sets iseed to m
4 global iseed;
5   newseed=round(m);
6   iseed=abs(newseed);
7 end
```

```
>> sub8seed(1024)
>> sub8random0(1,10)
ans = 0.0598    1.0000    0.0905    0.2060    0.2620    0.6432    0.6325    0.8392    0.6278    0.5463
>> sub8random0(1,10)
ans = 0.7551    0.4554    0.3177    0.9105    0.1289    0.6230    0.2266    0.3858    0.5876    0.7880
>> sub8seed(1024)
>> sub8random0(1,5)
ans = 0.0598    1.0000    0.0905    0.2060    0.2620
>> random(1,5)
ans = 0.6432    0.6325    0.8392    0.6278    0.5463
>> sub8random0(1,5)
ans = 0.7551    0.4554    0.3177    0.9105    0.1289
>> sub8random0(1,2)
ans = 0.6230    0.2266
```

(c) Copyright A. Gerbessiotis. All rights reserved.

8.12.1 Function functions. They are functions whose parameters can also be functions. A function called `fzero` that finds the zeroes of a function in an interval is such a function. In a call, its first argument is the input function, with the second argument being an interval in which a root belongs. Other than `fzero`, other such function functions include `fminbnd` that minimizes a function of one variable, `quad` that numerical integrates a function, `ezplot` an easy to use function plotter, and `fplot` a method to plot a function by name.

This MATLAB behavior is possible by way of functions `eval, feval`. The former evaluates a character string as if it had been typed in a Command Window. The latter `feval` evaluates a function at an input value. When a function argument is passed to a function function this argument needs to be preceded by `@` to avoid confusion. This is highlighted in Method 4 below, where function `k` is a primary function defined in `k.m`. Had we called with argument `k` instead of `@k` MATLAB would have attempted to evaluate `k`, i.e. call function `k` without an argument. This would have resulted in an error. To pass the function itself rather than the return value we need to write `@k`.

```
>> a=fzero('x^2-4',[0 4]);              % Method 1  : type function as a string
a = 2
>> b=fzero(@(x)(x^2 -4),[0,4]);         % Method 2  : use an anonymous function definition
b = 2
>> g = @ (x) (x^2 -4 );
>> c=fzero(g, [0 4]);                   % Method 3  : use an anonymous function (predefined)
c = 2
>> type k.m
function y=k(x)
  y= x^2 -4;
end
>> d= fzero(@k,[0,4]);                  % Method 4  : use a primary function note @fname needed
d = 2
>> eval('sin(pi/2)');
ans = 1
>> x=feval('sin',pi/2);
x = 1
>> x=fminbnd('x^2+2*x+1',-10,10)
x = -1.0000
>> quad('x.^2',0,3)   % function must accept vector as input; that's why we have .^ not just ^
ans = 9.0000
```

(c) Copyright A. Gerbessiotis. All rights reserved.

8.13.1 Scope of functions. The `scope` of a function in MATLAB is the location within MATLAB from which the function can be accessed. For an primary function this is the current working directory.

8.13.2 Order of evaluation. When a function with a given name is invoked, MATLAB, locates a function with that name as follows.

1. MATLAB first checks if there is a nested function with the given name. If so, it is executed.
2. MATLAB then checks if there is a subfunction with the given name. If so, it is executed.
3. MATLAB then checks if there is a private function with the given name. If so, it is executed.
4. MATLAB then checks if there is a function in the current working directory with the given name. If so, it is executed.
5. MATLAB then checks if there is a function in the MATLAB path with the given name. If so, the first one found is executed.
6. No function found; MATLAB will report an error.

(c) Copyright A. Gerbessiotis. All rights reserved.