E7: Introduction to Computer Programming for Scientists and Engineers

University of California at Berkeley, Spring 2017

Instructor: Lucas A. J. Bastien

Diary for lecture 05: More on Functions

Version: release

```
% This document presents and illustrates concepts related to:
%
% - Calling Matlab built-in functions from your own functions
% - Calling your own sub-functions from your own functions
% - Calling your own nested functions from your own functions
% - Function handles
% - Anonymous functions
%
% In this document, I call "user-defined functions" the functions that I
% define and implement myself. In this scenario, I am the user. The
% functions that you will write for your E7 assignments will also be
% "user-defined" functions. In that scenario, you are the user. In
% contrast, built-in functions are functions that are already defined in
% Matlab when you install it.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% CALLING MATLAB BUILT-IN FUNCTIONS FROM YOUR OWN FUNCTIONS %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% We will use the following variables to test our functions
>> a = [0, 0];
>> b = [0, 5];
>> c = [5, 5];
>> d = [5, 0];

% ----------------------------------------------------------------------------

% I define a function ("my_quadrilateral") that calculates the perimeter of
% a quadrilateral given the coordinates of its vertices. This function
% calls the Matlab built-in function "sqrt". Note that the command:
%
% type my_function.m
%
% displays the content of the m-file named "my_function.m"

>> type my_quadrilateral

function [perimeter] = my_quadrilateral(a, b, c, d)
% Calculate the perimeter of a quadrilateral "abcd" given the x- and y-
% coordinates of its four vertices: a, b, c, and d.
%
```

```
% Inputs are vectors (either row or column), each containing 2 elements:
% the x- and y- coordinates (in this order) of the corresponding vertex.
%
% Note: this function calls the Matlab built-in function "sqrt".

side1 = sqrt((a(1)-b(1))^2 + (a(2)-b(2))^2);
side2 = sqrt((b(1)-c(1))^2 + (b(2)-c(2))^2);
side3 = sqrt((c(1)-d(1))^2 + (c(2)-d(2))^2);
side4 = sqrt((d(1)-a(1))^2 + (d(2)-a(2))^2);

perimeter  = side1 + side2 + side3 + side4;

end

% Let's try!
>> perimeter = my_quadrilateral(a, b, c, d)

perimeter =

    20

% ----------------------------------------------------------------------------

% In the following example, I define two separate functions:
% "my_quadrilateral_separate" and "my_distance_separate". The function
% "my_quadrilateral_separate" calls the function "my_distance_separate"

>> type my_quadrilateral_separate

function [perimeter] = my_quadrilateral_separate(a, b, c, d)
% Calculate the perimeter of the quadrilateral "abcd".
%
% Inputs are vectors (either row or column), each containing 2 elements:
% the x- and y- coordinates (in this order) of the corresponding vertex.
%
% Note: this function calls the user-defined function "my_distance_separate".

side1 = my_distance_separate(a, b);
side2 = my_distance_separate(b, c);
side3 = my_distance_separate(c, d);
side4 = my_distance_separate(d, a);

perimeter  = side1 + side2 + side3 + side4;

end

>> type my_distance_separate

function [d] = my_distance_separate(a, b)
% Calculate the distance between point a and point b.
%
% Inputs are vectors (either row or column), each containing 2 elements:
```

```
% the x- and y- coordinates (in this order) of the corresponding vertex.

d = sqrt((a(1)-b(1))^2 + (a(2)-b(2))^2);

end

% Let's try!
>> perimeter = my_quadrilateral_separate(a, b, c, d)

perimeter =

    20

% I can also call my_distance_separate from the command prompt (or from
% other user-defined functions). For example:
>> my_distance_separate(a, b)

ans =

     5

% ---------------------------------------------------------------------------

% In the following example, I define the function "my_quadrilateral_sub",
% which calls the sub-function "my_distance_sub"

>> type my_quadrilateral_sub

function [perimeter] = my_quadrilateral_sub(a, b, c, d)
% Calculate the perimeter of the quadrilateral "abcd".
%
% Inputs are vectors (either row or column), each containing 2 elements:
% the x- and y- coordinates (in this order) of the corresponding vertex.
%
% Note: this function calls the user-defined sub-function "my_distance_sub".

side1 = my_distance_sub(a, b);
side2 = my_distance_sub(b, c);
side3 = my_distance_sub(c, d);
side4 = my_distance_sub(d, a);

perimeter  = side1 + side2 + side3 + side4;

end

function [d] = my_distance_sub(a, b)
% Calculate the distance between point a and point b.
%
% Inputs are vectors (either row or column), each containing 2 elements:
% the x- and y- coordinates (in this order) of the corresponding vertex.

d = sqrt((a(1)-b(1))^2 + (a(2)-b(2))^2);
```

```
end

% Let's try!
>> perimeter = my_quadrilateral_sub(a, b, c, d)

perimeter =

    20

% Note: the sub-function "my_distance_sub" cannot be directly called
% (without workarounds) from the command-line. Sub-functions can only be
% called by functions defined in the same m-file (either by the main
% function or by other sub-functions). There are workarounds to this
% limitation, but I will not talk about them
>> my_distance_sub(a, b)

Undefined function or variable 'my_distance_sub'.

% ------------------------------------------------------------------------------

% In the following example, I define the function
% "my_quadrilateral_nested", which calls the nested-function
% "my_distance_nested"

>> type my_quadrilateral_nested

function [perimeter] = my_quadrilateral_nested(a, b, c, d)
% Calculate the perimeter of the quadrilateral "abcd".
%
% Inputs are vectors (either row or column), each containing 2 elements:
% the x- and y- coordinates (in this order) of the corresponding vertex.
%
% Note: this function calls the user-defined nested function
% "my_distance_nested".

    function [d] = my_distance_nested(a, b)
        % Calculate the distance between point a and point b.
        %
        % Inputs are vectors (either row or column), each containing 2
        % elements: the x- and y- coordinates (in this order) of the
        % corresponding vertex.

        d = sqrt((a(1)-b(1))^2 + (a(2)-b(2))^2);

    end

side1 = my_distance_nested(a, b);
side2 = my_distance_nested(b, c);
side3 = my_distance_nested(c, d);
side4 = my_distance_nested(d, a);
```

```
perimeter  = side1 + side2 + side3 + side4;

end

% Let's try!
>> perimeter = my_quadrilateral_nested(a, b, c, d)

perimeter =

    20

% Note: the nested function "my_distance_nested" cannot be directly called
% (without workarounds) from the command-line. Nested functions can only be
% called by their parent function (i.e. the function within which they are
% defined). There are workarounds to this limitation, but I will not talk
% about them
>> my_distance_nested(a, b)

Undefined function or variable 'my_distance_nested'.

%%%%%%%%%%%%%%%%%%%%
% FUNCTION HANDLES %
%%%%%%%%%%%%%%%%%%%%

% A function handle is an association to a function that can be stored in
% variables. One can use @ to obtain a function handle to an existing
% function given its name. One can store the function handle in a variable
% and then use that variable to call the function. For example:
>> handle_to_cos = @cos

handle_to_cos =

  function_handle with value:

    @cos

>> cos(pi/3)

ans =

    0.5000

>> handle_to_cos(pi/3)

ans =

    0.5000

% handle_to_cos is not a function. Rather, it is a variable that contains a
% function handle. We can therefore assign the value of handle_to_cos to
% another variable, without using the @ sign
>> class(handle_to_cos)
```

```
ans =

function_handle

>> another_handle = handle_to_cos

another_handle =

  function_handle with value:

    @cos

>> another_handle(pi/3)

ans =

    0.5000

% A common application of function handles is to pass functions as input
% arguments to other functions. For example, consider the Matlab built-in
% function "fzero". This function can be called using the following syntax:
%
% x = fzero(f, initial_guess)
%
% where "f" is a function handle to a real-valued (class double) function
% of one variable, "initial_guess" is a scalar of class double, and "x" is
% also a scalar of class double.
%
% When using this syntax, "fzero" returns (when it works) a number x such
% that f(x) = 0. To increase the chances that "fzero" finds such a number,
% give "initial_guess" a value close the expected number "x"
%
% For example, find a number x such that cos(x) = 0:
>> fzero(@cos, 1)

ans =

    1.5708

% Isn't this number pi/2?
>> pi / 2

ans =

    1.5708

% We can find another number x such that cos(x) = 0 by giving
% "initial_guess" a different value
>> fzero(@cos, 7)

ans =
```

```
    7.8540

% Isn't this number (pi/2 + 2*pi)?
>> pi/2 + 2*pi

ans =

    7.8540

%%%%%%%%%%%%%%%%%%%%%%%
% ANONYMOUS FUNCTIONS %
%%%%%%%%%%%%%%%%%%%%%%%

% An anonymous function is a function that does not have a name, but that
% can be used with the help of a function handle associated with it. See
% the slides of this lecture for the syntax used to define anonymous
% functions. For example, define an anonymous function that, given x,
% calculates x.^2 - 2
>> my_function = @(x) x.^2 - 2

my_function =

  function_handle with value:

    @(x)x.^2-2

% Use the handle to this anonymous function and the built-in function
% "fzero" to calculate the value of the square root of 2
>> fzero(my_function, 2)

ans =

    1.4142

% Let's check our result
>> sqrt(2)


ans =

    1.4142

% One can use the values of existing variables (as is the case with "x0"
% and "y0" in the example below) when defining anonymous functions.
% Changing the values of these variables after the anonymous function has
% been defined has no effect on the anonymous function. Let us check this
% fact...
>> x0 = 2;
>> y0 = 5;

% Define a function handle to calculate the distance from the point of
```

```
% coordinates (x0,y0)
>> distance = @(x, y) sqrt((x-x0).^2 + (y-y0).^2)

distance =

  function_handle with value:

    @(x,y)sqrt((x-x0).^2+(y-y0).^2)

>> distance(4, 5)

ans =

     2

% Give different values to x0 and y0
>> x0 = 1000;
>> y0 = 2000;

% Did the anonymous function that we just defined change?
>> distance(4, 5)

ans =

     2

% Anonymous functions can call built-in functions (for example "sqrt") in
% the previous example, and user-defined functions. For example, another
% way to define a function handle to calculate the distance from the point
% of coordinates (x0,y0) is to use the function "my_distance_separate" that
% we defined earlier today
>> x0 = 2;
>> y0 = 5;
>> distance = @(x, y) my_distance_separate([x0, y0], [x, y])

distance =

  function_handle with value:

    @(x,y)my_distance_separate([x0,y0],[x,y])

>> distance(4, 5)

ans =

     2
```