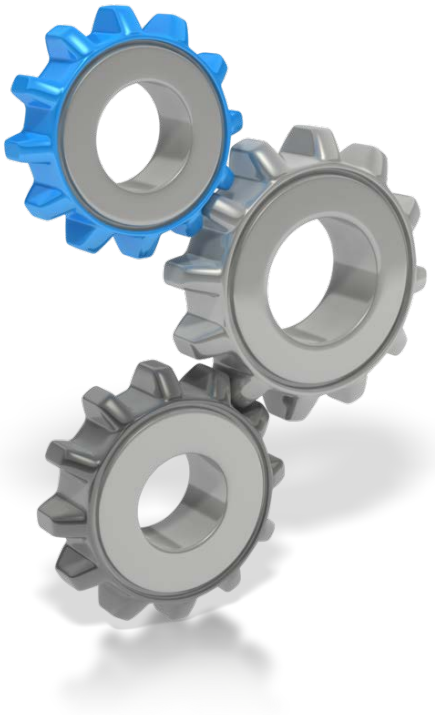
Module 07: Function

Chul Min Yeum
Assistant Professor
Civil and Environmental Engineering
University of Waterloo, Canada



UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING

Module 7: Learning Outcomes

- Explain the advantages of using functions
- Create the functions that supports various input and output variables
- Illustrate the difference between main and local functions
- Assess variable scope in the script having local functions.
- Program your own functions.

What is a Function?

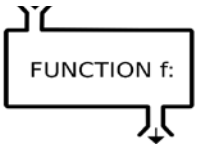
- Function is a type of procedure or routine that performs specific tasks.
- It is a block of **organized** and **reusable** scripts that is used to perform a single and related tasks.
- There are built-in functions provided by MATLAB, but users can write their own functions for reusing a certain operation.
- Advantage: **organized** (readable), **time saving** by avoiding mistake and reusing basic operations

Types of Functions

A function is a group of statements that together perform a task. In MATLAB, functions are defined in separate files. The name of the file and of the function should be the same.

Built-in Functions: Functions that are frequently used or that can take more time to execute are often implemented as executable files. These functions are called built-ins.

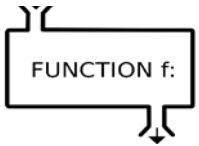
Single input



Single output

x = -1
y = abs(-1)

Multiple inputs



Single output

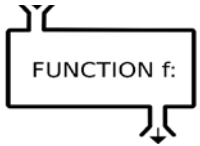
vec1 = randi(10,3,3)

Single/Multiple inputs

Multiple output

mat1 = [1 1; 1 1];
[cz, rz] = size(mat1)

Single/Multiple inputs



No return value

disp('hello')

Example: Built-in Functions

sum

R2020a

Sum of array elements

collapse all in page

Syntax

```
S = sum(A)
S = sum(A,'all')
S = sum(A,dim)
S = sum(A,vecdim)
S = sum( __ ,outtype)
S = sum( __ ,nanflag)
```

Description

example

`S = sum(A)` returns the sum of the elements of A along the first array dimension whose size does not equal 1.

- If `A` is a vector, then `sum(A)` returns the sum of the elements.
- If `A` is a matrix, then `sum(A)` returns a row vector containing the sum of each column.
- If `A` is a multidimensional array, then `sum(A)` operates along the first array dimension whose size does not equal 1, treating the elements as vectors. This dimension becomes `1` while the sizes of all other dimensions remain the same.

example

`S = sum(A,'all')` computes the sum of all elements of `A`. This syntax is valid for MATLAB® versions R2018b and later.

example

`S = sum(A,dim)` returns the sum along dimension `dim`. For example, if `A` is a matrix, then `sum(A,2)` is a column vector containing the sum of each row.

example

`S = sum(A,vecdim)` sums the elements of `A` based on the dimensions specified in the vector `vecdim`. For example, if `A` is a matrix, then `sum(A,[1 2])` is the sum of all elements in `A`, since every element of a matrix is contained in the array slice defined by dimensions 1 and 2.

example

`S = sum( __ ,outtype)` returns the sum with a specified data type, using any of the input arguments in the previous syntaxes. `outtype` can be `'default'`, `'double'`, or `'native'`.

example

`S = sum( __ ,nanflag)` specifies whether to include or omit `NaN` values from the calculation for any of the previous syntaxes. `sum(A,'includenan')` includes all `NaN` values in the calculation while `sum(A,'omitnan')` ignores them.

Generic Function Definition

```
1  function [output arguments] = functionname (input arguments)
2
3  Your script
4
5  end
```

📖: Texts in red and bold are syntax.

- The function header (line 1) includes
  - the reserved word **function**
  - output argument(s): listing multiple arguments separating with comma (Note that the square bracket in the output arguments is not needed if there is only one output
  - function name (optional: start with a capital letter for making difference from other variable names.)
  - Input argument(s): listing multiple arguments separating with comma
- The body of function include your script to compute values in the output arguments using input arguments.
- Finished with the reserved word **end**

Notes on Functions

- The function header and function call have to match up:
  - the name must be the same
  - the number of input arguments must be the same
  - the number of variables in the left-hand side of the assignment should be the same as the number of output arguments

📖: For example, if the function header is:

```
function [x,y,z] = fnname(a,b)
```

This indicates that the function is returning three outputs, so a call to the function might be (assuming a and b are numbers):

```
[g,h,t] = fnname(11, 4.3);
```

Or using the same names as the output arguments (**it doesn't matter since the workspace is not shared**):

```
[x,y,z] = fnname(11, 4.3);
```

Example: Make a Function to Compute an Absolute Value

```
% given an input scalar named
s1, compute the absolute value
of s1 and assign it to s2

s1 = 3;

if s1>=0
   s2 = s1;
end

if s1<0
   s2 = s1*-1;
end
```

Name	Value
s1	3
s2	3

Q. How to make a function called 'MyAbs' that can compute an absolute value like abs.

```
function x_abs = MyAbs(x)

if x>=0
    x_abs = x;
end

if s1<0
    x_abs = x*-1;
end

end
```

```
s1 = -3;
s2 = MyAbs(s1);
```

What are the Difference between Scripts and Functions?

Script: Script files are program files with a .m extension. In these files, you write series of commands, in which you want to execute together. Scripts do not accept the inputs and do not return any outputs. They operate on data in the base workspace.

```
Input

Series of processing

Output
```

my_script.m

Function: functions file are also program files with a .m extension. Functions can accept inputs and return outputs. Internal variables are local to the function. They use separate workspace.

```
Input data

Output = MyFun(input)

Output data
```

call

```
function output = MyFun(input)

Series of processing

end
```

my_script.m **MyFun.m**

How to Save and Call Functions

Save

- The function is stored in a code file with the extension .m
- The file name **must be the same** as the function name.
- The file should be placed at the same folder where the script use the function. Otherwise, you need to add the script folder to search its path.

Call

- Calling the function should be in an assignment statement with the input and output argument(s), which is the same as the number of the input and output arguments in the function header.
- You can use any input and output variable names when you call the function because they are not sharing the Workspace (different variable scope).

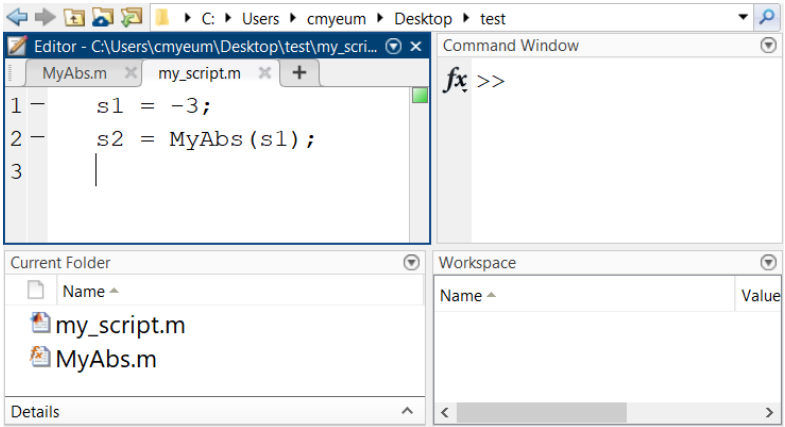
How to Use the Function

MyAbs.m

```
function x_abs = MyAbs(x)

if x>=0
    x_abs = x;
end

if s1<0
    x_abs = x*-1;
end

end
```

my_script.m

```
s1 = -3;
s2 = MyAbs(s1);
```



📖: Function and script files should be placed at the same folder. If not, we need add folders (`addpath`) where the functions are present so that MATLAB can search for the functions

Example: Bulls and Cows

Bulls and Cows is a mind game played by two players. In the game, a random, 4-digit number is chosen, and its values are compared to those of another trial number. **All four digits of the number must be different**. If any digit in the chosen number is the exact same value and in the exact same position as any digit in the trial number, this is called a bull. If the digit is present in both the trial number and chosen number, but is not in the same location, this is called a cow.

Example: Bulls and Cows – Compute Bulls

Q: Write a function named '`CompBulls`' to compute '`bulls`' when test and true numbers are given, which are named as '`x_true`' and '`x_test`'.

```
% This is the code that we
developed in the previous
module!

x_true = [1 2 3 4]; % true
x_test = [3 2 5 6]; % test

is_same = (x_true == x_test);
num_b = sum(is_same);
```

my_script.m

```
x_true = [1 2 3 4]; % true
x_test = [3 2 5 6]; % test

bulls = CompBulls(x_true, x_test);
```

CompBulls.m

```
function num_b = CompBulls(x_tr, x_ts)

is_same = (x_tr == x_ts);
num_b = sum(is_same);

end
```

Example: Bulls and Cows – Compute Cows

Q: Write a function named 'CompCows' to compute 'cows' when test and true numbers are given, which are named as 'x_true' and 'x_test'.

```
function cows = CompCows(x_tr, x_ts)

num_c = 0; % bulls + cows

for ii=1:4
    if any(x_tr == x_ts(ii))
        num_c = num_c + 1;
    end
end

is_same = (x_tr == x_ts);
num_b = sum(is_same); % bulls

cows = num_c - num_b; % cows

end
```

Option 1

```
x_true = [1 2 3 4]; % true
x_test = [3 2 5 6]; % test

cows = CompCows(x_true, x_test);
```

```
function cows = CompCows(x_tr, x_ts)

num_c = 0; % bulls + cows
for ii=1:4
    if any(x_tr == x_ts(ii))
        num_c = num_c + 1;
    end
end

num_b = CompBulls(x_tr, x_ts)
cows = num_c - num_b; % cows

end
```

Option 2

Example: Bulls and Cows – Compute Bulls and Cows

Q: Write a function named 'CompBC' to compute '`bulls`' and '`cows`' when test and true numbers are given, which are named as '`x_true`' and '`x_test`'.

```
function [bulls, cows] = CompBC(x_true, x_test)

bulls = CompBulls(x_true, x_test);
cows = CompCows(x_true, x_test);

end
```

```
function num_b = CompBulls(x_tr, x_ts)

is_same = (x_true == x_test);
num_b = sum(is_same);

end
```

📖: Assume that each function is stored in its m-file with the file name identical to its function name.

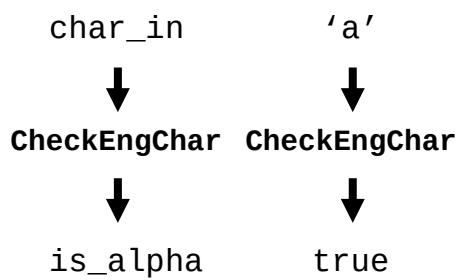
```
function cows = CompCows(x_tr, x_ts)

num_c = 0; % bulls + cows
for ii=1:4
    if any(x_tr == x_ts(ii))
        num_c = num_c + 1;
    end
end

num_b = CompBulls(x_tr, x_ts)
cows = num_c – num_b; % cows

end
```

Example: Is this Character English Alphabet?

Q: Given a character named 'one_char', write a function to check if 'one_char' is in the English alphabet. If yes, assign `true` to 'is_alpha', otherwise `false`.

```
one_char = 'a'; % input character

char_db = double(one_char);
cond1 = and(65 <= char_db, char_db <=90); % upper case
cond2 = and(97 <= char_db, char_db <=122); % lower case

is_alpha = or(cond1, cond2); % output
```

```
function is_alpha = CheckEngChar(char_in)

char_db = double(char_in);
cond1 = and(65 <= char_db, char_db <=90);
cond2 = and(97 <= char_db, char_db <=122);
is_alpha = or(cond1, cond2); % output

end
```



Example: How Many English Alphabets in a Character Vector?

```
char_vec = 'asbdec#43!@3';
num_char = numel(char_vec);

num_alpha = 0;
for ii=1:num_char

    one_char = char_vec(ii);
    char_db = double(one_char);
    cond1 = and(65 <= char_db, char_db <=90);
    cond2 = and(97 <= char_db, char_db <=122);

    is_alpha = or(cond1, cond2); % output

    if is_alpha == 1
        num_alpha = num_alpha + 1;
    end
end
```

Q: Given a character vector named 'char_vec', write a script to count the number of English alphabet characters in 'char_vec'. Assign the number to 'num_alpha'.

😂: If you pack the script as a function, the code can be shortened and readable.

```
char_vec = 'asbdec#43!@3';
num_char = numel(char_vec);

num_alpha = 0;
for ii=1:num_char
    is_alpha = CheckEngChar(char_vec(ii));

    if is_alpha
        num_alpha = num_alpha + 1;
    end
end
```

Example: Is There Non-Alphabet Character in a Character Vector?

```
char_vec = 'asbdec#43!@3';
num_char = numel(char_vec);

is_all_alpha = true;
for ii=1:num_char

    one_char = char_vec(ii);
    char_db = double(one_char);
    cond1 = and(65 <= char_db, char_db <=90);
    cond2 = and(97 <= char_db, char_db <=122);

    is_alpha = or(cond1, cond2);

    if is_alpha ~= 1
        is_all_alpha = false;
        break;
    end
end
```

Q: Given a character vector named 'char_vec', write a script to check if the vector only contain English alphabet characters. If yes, assign `true` to 'is_all_alpha', otherwise assign `false`.

```
char_vec = 'asbdec#43!@3';
num_char = numel(char_vec);

is_all_alpha = true;
for ii=1:num_char
    is_alpha = CheckEngChar(char_vec(ii));

    if is_alpha ~= 1
        is_all_alpha = false;
        break;
    end
end
```

Example: Vectorization

```
function is_alpha = CheckEngChar(char_in)

char_db = double(char_in);

cond1 = and(65 <= char_db, char_db <=90);
cond2 = and(97 <= char_db, char_db <=122);
is_alpha = or(cond1, cond2); % output

end
```

📖: Here, the input does not have to be a single character. The script supports a character vector as an input. Then, the output will be the same size as its input vector.

Q. How many English alphabet?

```
char_vec = 'asbdec#43!@3';

is_alpha = CheckEngChar(char_vec);

num_alpha = sum(is_alpha);
```

Q. Is there non-alphabet character in a character vector?

```
char_vec = 'asbdec#43!@3';

is_alpha = CheckEngChar(char_vec);

is_all_alpha = all(is_alpha);
```

Example: How Many Upper-Case and Lower-Case Letters?

```
function [num_up, num_low] = CheckCharCase(char_in)

char_db = double(char_in);

lg_up = and(65 <= char_db, char_db <=90); % index upper case
lg_low = and(97 <= char_db, char_db <=122); % index lower case

num_up = sum(lg_up);
num_low = sum(lg_low);

end
```

```
>> char_vec = 'asbdEc#43!@3';
>> [num_up, num_low] = CheckCharCase(char_vec)

num_up =

     1


num_low =

     5
```

📖: The function designed here computes numbers of both upper and lower-case letters when one input vector is provided.

Example: How Many Lower-Case and Upper-Case Letters? (Continue)

Challenging

```
function num_char = CheckCharCase(char_in, lt_case)

char_db = double(char_in);

if isequal(lt_case, 'upper')
    lg_up = and(65 <= char_db, char_db <=90);
    num_char = sum(lg_up);

elseif isequal(lt_case, 'lower')
    lg_low = and(97 <= char_db, char_db <=122);
    num_char = sum(lg_low);

else
   error('wrong second argument of lt_case');
end

End
```

📖: We can make the function to selectively compute either lower- or upper-case letter by providing the second argument.

```
>> char_vec = 'asbdEc#43!@3';
>> num_low_char = CheckCharCase(char_vec, 'lower')

num_low_char  =

     5
```

Local Function

MATLAB® program files can contain code for more than one function. In a function **script** file, the first function in the file is called the **main function**. This function is visible to functions in other files, or you can call it from the command line. Additional functions within the file are called **local functions**, and they can occur in any order after the main function.

MyFun.m

```
function out = MyFun(in)

call MyFunSub1
call MyFunSub2

processing using outputs
from sub functions

end
```

MyFunSub1.m

```
function out = MyFunSub1 (in)

a series of processing

end
```

MyFunSub2.m

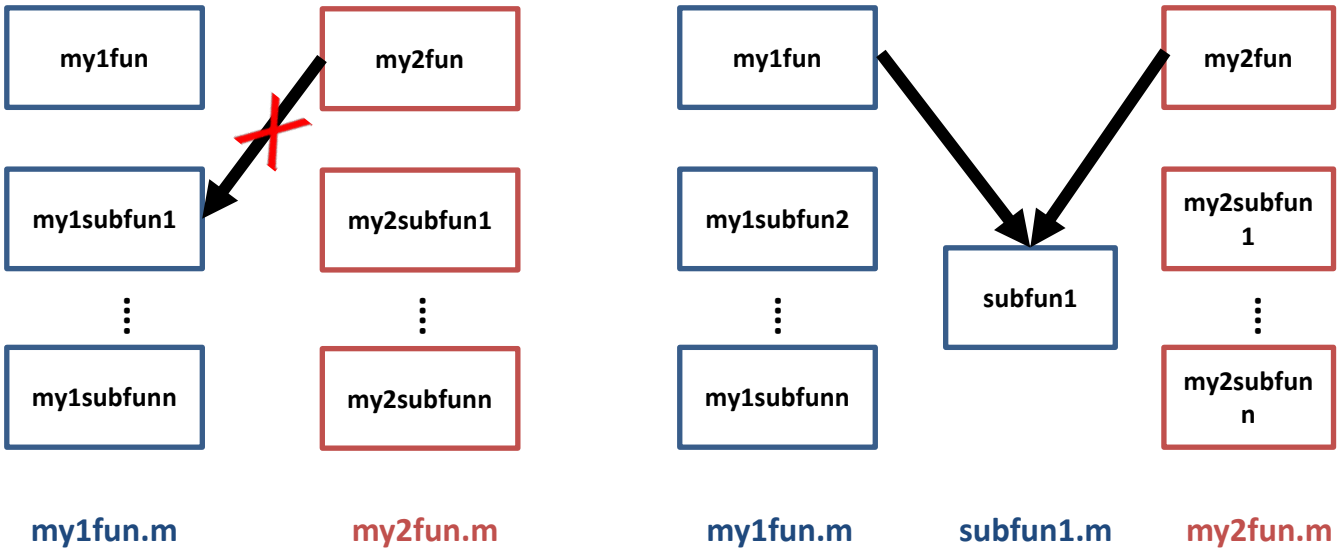
```
function out = MyFunSub2 (in)

a series of processing

end
```

MyFunAll.m

```
function out = MyFunAll(in)        <-- main function
call MyFunSub1
call MyFunSub2
processing using outputs from sub functions
end

function out = MyFunSub1 (in)      <-- local function
a series of processing
end

function out = MyFunSub2 (in)      <-- local function
a series of processing
end
```

Call Local Functions

Local functions are only visible to other functions **in the same file**. They are equivalent to subroutines in other programming languages and are sometimes called subfunctions.



Example: Bulls and Cows – Compute Bulls and Cows

CompBC.m

```
function [bulls, cows] = CompBC(x_true, x_test)
bulls = CompBulls(x_true, x_test);
cows = CompCows(x_true, x_test);
end

function num_b = CompBulls(x_tr, x_ts)

is_same = (x_tr == x_ts);
num_b = sum(is_same);

end

function num_c = CompCows(x_tr, x_ts)

num_c = 0;
for ii=1:4
    if any(x_tr == x_ts (ii))
        num_c = num_c + 1;
    end
end
bulls = CompBulls(x_tr, x_ts);
num_c = num_c - bulls;

end
```

```
>> x_tr = [1 2 3 6]; % true
>> x_ts = [3 2 5 6]; % test
>> [bls, cws] = CompBC(x_tr, x_ts)
bls =

     2


cws =

     1
```

my_script.m

```
x_tr = [1 2 3 6]; % true
x_ts = [3 2 5 6]; % test

[bls, cws] = CompBC(x_tr, x_ts)
```

Add Functions to Scripts

- MATLAB® scripts, including live scripts, can **contain code to define functions.** These functions are called local functions. Local functions are useful if you want to reuse code within a script. By adding local functions, you can avoid creating and managing separate function files. They are also useful for experimenting with functions, which can be added, modified, and deleted easily as needed. Functions in scripts are supported **in R2016b or later**.
- Local functions are only visible within the file where they are defined, both to the script code and other local functions within the file. They are not visible to functions in other files and cannot be called from the command line. They are equivalent to subroutines in other programming languages and are sometimes called subfunctions.

Example: Add Functions to Your Script

```
x_tr = [1 2 3 6]; % true
x_ts = [3 2 5 6]; % test
[bulls, cows] = CompBC(x_tr, x_ts)

function [bulls, cows] = CompBC(x_true, x_test)

bulls = CompBulls(x_true, x_test);
cows = CompCows(x_true, x_test);

end

function num_b = CompBulls(x_tr, x_ts)

is_same = (x_tr == x_ts);
num_b = sum(is_same);

end

function num_c = CompCows(x_tr, x_ts)
num_c = 0;
for ii=1:4
    if any(x_tr == x_ts (ii))
        num_c = num_c + 1;
    end
end
bulls = CompBulls(x_tr, x_ts);
num_c = num_c - bulls;

end
```

📖: We can write both script and function at the same m-file. In this case, all functions become local function, which means these cannot be accessed from the other files.

😂: This is very useful when you test your function. Before 2016, we need to have a separate script to test function or using a command window.

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***.
- Scripts also create variables in the base workspace. That means that variables created in the Command Window can be used in scripts and vice versa
- Functions do not use the base workspace. Every function has **its own function workspace.** Each function workspace is separate from the base workspace and all other workspaces to protect the integrity of the data. Even local functions in a common file have their own workspaces. Variables specific to a function workspace are called local variables. Typically, local variables do not remain in memory from one function call to the next.

Change Workspace and Variable Scope

Challenging

```
1   x_tr = [1 2 3 6]; % true
2   x_ts = [3 2 5 6]; % test
3
4   [bls, cws] = CompBC(x_tr, x_ts);
5
6   disp('Done!');
7
8   function [bulls, cows] = CompBC(x_true, x_test)
9   bulls = CompBulls(x_true, x_test);
10  cows = CompCows(x_true, x_test);
11  end
12
13  function num_b = CompBulls(x_tr, x_ts)
14  is_same = (x_tr == x_ts);
15  num_b = sum(is_same);
16  end
17
18  function num_c = CompCows(x_tr, x_ts)
19  num_c = 0;
20  for ii=1:4
21      if any(x_tr == x_ts(ii))
22          num_c = num_c + 1;
23      end
24  end
25  bulls = CompBulls(x_tr, x_ts);
26  num_c = num_c - bulls;
27  end
```

Workspace - CompBulls

Name	Value
is_same	*1x4 logical*
num_b	2
x_tr	[1,2,3,6]
x_ts	[3,2,5,6]

Stop at line 16

Workspace - CompBC

Name	Value
bulls	2
x_test	[3,2,5,6]
x_true	[1,2,3,6]

Stop at line 10

Workspace - CompCows

Name	Value
ii	4
num_c	3
x_tr	[1,2,3,6]
x_ts	[3,2,5,6]

Stop at line 25

Workspace - CompBulls

Name	Value
is_same	*1x4 logical*
num_b	2
x_tr	[1,2,3,6]
x_ts	[3,2,5,6]

Stop at line 16

Workspace - CompBC

Name	Value
bulls	2
cows	1
x_test	[3,2,5,6]
x_true	[1,2,3,6]

Stop at line 11

Workspace - my_script

Name	Value
bls	2
cws	1
x_tr	[1,2,3,6]
x_ts	[3,2,5,6]

Stop at line 6

Example: Variable Scope (Which is a Valid Code ?)

```
in_vec = [1 2 3 4];
n_elem = numel(in_vec);
mean_val = myfun_mean(in_vec);

function mean_val = myfun_mean(vec)

mean_val = sum(vec)/n_elem;

end
```

Error

```
vec = [1 2 3 4];
n_elem = numel(vec);
val1 = myfun_mean(vec);

function mean_val = myfun_mean(vec)

mean_val = sum(vec)/n_elem;

end
```

Error

```
vec = [1 2 3 4];
n_elem = numel(vec);
m_val = mf_mean(vec);

function m_val = mf_mean(vec, n_elem)

m_val = sum(vec)/n_elem;

end
```

Error

```
vec = [1 2 3 4];
n_elem = numel(vec);
mean_val = myfun_mean(vec);

function mean_val = myfun_mean(vec)

n_elem = numel(vec);
mean_val = sum(vec)/n_elem;

end
```

No error