

MATLAB Programs

6.1 MORE TYPES OF USER-DEFINED FUNCTIONS

We have already seen how to write a user-defined function that calculates and returns one value. This is just one type of function. It is also possible to **have functions that return more than one value, and functions that do not return any values**. We will categorize functions as follows:

- Functions that calculate and return one value
- Functions that calculate and return more than one value
- Functions that just accomplish a task, such as printing, without returning any values

In general, any function in MATLAB consists of the following:

- The function header (the first line); this has:
 - the reserved word `function`
 - if the function returns values, the name(s) of the output argument(s), followed by the assignment operator (`=`)
 - the name of the function (important: this should be the same as the name of the file in which this function is stored to avoid confusion)
 - the input arguments in parentheses, if there are any (separated by commas if there is more than one).
- A comment that describes what the function does (this is printed if help is used).
- The body of the function, which includes all statements, including putting values in all output arguments if there are any.
- `end` at the end of the function.

6.1.1 Functions That Return More Than One Value

Functions that return one value have one output argument, as we saw previously. Functions that return more than one value must, instead, have more than one output argument in the function header in square brackets. That means that in the body of the function, values must be put in all output arguments listed in the function header. The general form of a function definition for a function that calculates and returns more than one value looks like this:

`functionname.m`

```
function [output arguments]=functionname(input arguments)
% Comment describing the function
% Format of function call

Statements here; these must include putting values in all of the
output arguments listed in the header
end
```

In the vector of output arguments, the output argument names are by convention separated by commas.

Choosing **New**, then **Function** brings up a template in the Editor that can then be filled in. If this is not desired, it may be easier to start with **New Script**.

For example, here is a function that calculates two values, both the area and the circumference of a circle; this is stored in a file called `areacirc.m`:

`areacirc.m`

```
function [area, circum]=areacirc(rad)
% areacirc returns the area and
% the circumference of a circle
% Format: areacirc(radius)

area=pi * rad .* rad;
circum=2 * pi * rad;
end
```

Because this function is calculating two values, there are two output arguments in the function header (`area` and `circum`), which are placed in square brackets `[]`.

Therefore, somewhere in the body of the function, values have to be stored in both.

Because the function is returning two values, it is important to capture and store these values in separate variables when the function is called. In this case, the first value returned, the area of the circle, is stored in a variable **a** and the second value returned is stored in a variable **c**:

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

If this is not done, only the first value returned is retained, in this case, the area:

```
>> disp(areacirc(4))
50.2655
```

QUICK QUESTION!

What would happen if a vector of radii was passed to the function?

Answer: As the `*` operator is used in the function to multiply *rad* by itself, a vector can be passed to the input argument *rad*. Therefore, the results will also be vectors, so the variables on the left side of the assignment operator would become vectors of areas and circumferences.

```
>> [a, c]=areacirc(1:4)
a =
    3.1416    12.5664    28.2743    50.2655
c =
    6.2832    12.5664    18.8496    25.1327
```

QUICK QUESTION!

What if you want only the second value that is returned?

Answer: Function outputs can be ignored using the tilde:

```
>> [~, c]=areacirc(1:4)
c =
    6.2832    12.5664    18.8496    25.1327
```

The **help** function shows the comment listed under the function header:

```
>> help areacirc
```

This function calculates the area and the circumference of a circle
Format: areacirc(radius)

The **areacirc** function could be called from the Command Window, as has been shown, or from a script. Here is a script that will prompt the user for the radius of just one circle, call the **areacirc** function to calculate and return the area and circumference of the circle, and print the results:

calcareacirc.m

```
% This script prompts the user for the radius of a circle,  
% calls a function to calculate and return both the area  
% and the circumference, and prints the results  
% It ignores units and error-checking for simplicity  
  
radius=input('Please enter the radius of the circle: ');  
[area, circ]=areacirc(radius);  
fprintf('For a circle with a radius of %.1f,\n', radius)  
fprintf('the area is %.1f and the circumference is %.1f\n',...  
        area, circ)
```

```
>> calcareacirc  
Please enter the radius of the circle: 5.2  
For a circle with a radius of 5.2,  
the area is 84.9 and the circumference is 32.7
```

PRACTICE 6.1

Write a function *perimarea* that calculates and returns the perimeter and area of a rectangle. Pass the length and width of the rectangle as input arguments. For example, this function might be called from the following script:

calcareaperim.m

```
% Prompt the user for the length and width of a rectangle,  
% call a function to calculate and return the perimeter  
% and area, and print the result  
% For simplicity it ignores units and error-checking  
  
length=input('Please enter the length of the rectangle: ');  
width=input('Please enter the width of the rectangle: ');  
[perim, area]=perimarea(length, width);  
fprintf('For a rectangle with a length of %.1f and a', length)  
fprintf(' width of %.1f,\nthe perimeter is %.1f,', width, perim)  
fprintf(' and the area is %.1f\n', area)
```

As another example, consider a function that calculates and returns three output arguments. The function receives one input argument representing a total number of seconds and returns the number of hours, minutes, and remaining seconds that it represents. For example, 7515 total seconds is 2 hours, 5 minutes, and 15 seconds because $7515 = 3600 \cdot 2 + 60 \cdot 5 + 15$.

The algorithm is as follows.

- Divide the total seconds by 3600, which is the number of seconds in an hour. For example, $7515/3600$ is 2.0875. The integer part is the number of hours (e.g., 2).
- The remainder of the total seconds divided by 3600 is the remaining number of seconds; it is useful to store this in a local variable.
- The number of minutes is the remaining number of seconds divided by 60 (again, the integer part).
- The number of seconds is the remainder of the previous division.

breaktime.m

```
function [hours, minutes, secs]=breaktime(totseconds)
% breaktime breaks a total number of seconds into
% hours, minutes, and remaining seconds
% Format: breaktime(totalSeconds)

hours=floor(totseconds/3600);
remsecs=rem(totseconds, 3600);
minutes=floor(remsecs/60);
secs=rem(remsecs,60);
end
```

An example of calling this function is:

```
>> [h, m, s] = breaktime(7515)
h =
    2
m =
    5
s =
   15
```

As before, it is important to store all values that the function returns by using three separate variables on the left of the assignment.

6.1.2 Functions That Accomplish a Task Without Returning Values

Many functions do not calculate values but rather accomplish a task, such as printing formatted output. Because these functions do not return any values, there are no output arguments in the function header.

The general form of a function definition for a function that does not return any values looks like this:

functionname.m

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

For example, the following function just prints the two arguments, numbers, passed to it in a sentence format:

printem.m

```
function printem(a,b)
% printem prints two numbers in a sentence format
% Format: printem(num1, num2)

fprintf('The first number is %.1f and the second is %.1f\n',a,b)
end
```

As this function performs no calculations, there are no output arguments in the function header and no assignment operator (=). An example of a call to the *printem* function is:

```
>> printem(3.3, 2)
The first number is 3.3 and the second is 2.0
```

Note that, because the function does not return a value, it cannot be called from an assignment statement. Any attempt to do this would result in an error, such as the following:

```
>> x=prntem(3, 5) % Error!!  
Error using prntem  
Too many output arguments.
```

We can therefore think of the call to a function that does not return values as a statement by itself, in that the function call cannot be imbedded in another statement such as an assignment statement or a statement that prints.

The tasks that are accomplished by functions that do not return any values (e.g., output from an fprintf statement or a plot) are sometimes referred to as side effects. Some standards for commenting functions include putting the side effects in the block comment.

PRACTICE 6.2

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

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

6.1.3 Functions That Return Values Versus Printing

A function that calculates and returns values (through the output arguments) does not normally also print them; that is left to the calling script or function. It is good programming practice to separate these tasks.

If a function just prints a value, rather than returning it, the value cannot be used later in other calculations. For example, here is a function that just prints the circumference of a circle:

calccircum1.m

```
function calccircum1(radius)  
% calccircum1 displays the circumference of a circle  
% but does not return the value  
% Format: calccircum1(radius)  
  
disp(2*pi*radius)  
end
```

Calling this function prints the circumference, but there is no way to store the value so that it can be used in subsequent calculations:

```
>> calccircum1(3.3)  
20.7345
```

Because no value is returned by the function, attempting to store the value in a variable would be an error:

```
>> c=calccircum1(3.3)  
Error using calccircum1  
Too many output arguments.
```

By contrast, the following function calculates and returns the circumference so that it can be stored and used in other calculations. For example, if the circle is the base of a cylinder, and we wish to calculate the surface area of the cylinder, we would need to multiply the result from the calccircum2 function by the height of the cylinder.

calccircum2.m

```
function circle_circum=calccircum2(radius)  
% calccircum2 calculates and returns the  
% circumference of a circle  
% Format: calccircum2(radius)  
  
circle_circum=2*pi*radius;  
end
```

```
>> circumference=calccircum2(3.3)  
circumference=  
20.7345  
  
>> height=4;  
>> surf_area=circumference*height  
surf_area=  
82.9380
```

One possible exception to this rule of not printing when returning is to have a function return a value if possible but throw an error if not.

6.1.4 Passing Arguments to Functions

In all function examples presented thus far, at least one argument was passed in the function call to be the value(s) of the corresponding input argument(s) in the function header. The call-by-value method is the term for this method of passing the values of the arguments to the input arguments in the functions.

In some cases, however, it is not necessary to pass any arguments to the function. Consider, for example, a function that simply prints a random real number with two decimal places:

printrand.m

```
function printrand()
% printrand prints one random number
% Format: printrand or printrand()

fprintf('The random # is %.2f\n',rand)
end
```

Here is an example of calling this function:

```
>> printrand()
The random # is 0.94
```

As nothing is passed to the function, there are no arguments in the parentheses in the function call and none in the function header, either. The parentheses are not even needed in either the function or the function call. The following works as well:

printrandnp.m

```
function printrandnp
% printrandnp prints one random number
% Format: printrandnp or printrandnp()

fprintf('The random # is %.2f\n',rand)
end
```

```
>> printrandnp
The random # is 0.52
```

In fact, the function can be called with or without empty parentheses, whether or not there are empty parentheses in the function header.

This was an example of a function that did not receive any input arguments, nor did it return any output arguments; it simply accomplished a task.

The following is another example of a function that does not receive any input arguments, but, in this case, it does return a value. The function prompts the user for a string (meaning, actually, a character vector) and returns the value entered.

stringprompt.m

```
function outstr=stringprompt
% stringprompt prompts for a string and returns it
% Format stringprompt or stringprompt()

disp('When prompted, enter a string of any length.')
outstr=input('Enter the string here: ', 's');
end
```

```
>> mystring=stringprompt
When prompted, enter a string of any length.
Enter the string here: Hi there

mystring=
'Hi there'
```

PRACTICE 6.3

Write a function that will prompt the user for a string of at least one character, loop to error-check to make sure that the string has at least one character, and return it as a character vector.

QUICK QUESTION!

It is important that the number of arguments passed in the call to a function must be the same as the number of input arguments in the function header, even if that number is zero. Also, if a function returns more than one value, it is important to “capture” all values by having an equivalent number of variables in a vector on the left side of an assignment statement. Although it is not an error if there are not enough variables, some of the values returned will be lost. The following question is posed to highlight this.

Given the following function header (note that this is just the function header, not the entire function definition):

```
function [outa, outb]=qq1(x, y, z)
```

Which of the following proposed calls to this function would be valid?

- a) [var1, var2]=qq1(a, b, c);
- b) answer=qq1(3, y, q);

- c) [a, b]=myfun(x, y, z);
- d) [outa, outb]=qq1(x, z);

Answer: The first proposed function call, [a], is valid. There are three arguments that are passed to the three input arguments in the function header, the name of the function is *qq1*, and there are two variables in the assignment statement to store the two values returned from the function. Function call [b] is valid, although only the first value returned from the function would be stored in *answer*; the second value would be lost. Function call [c] is invalid because the name of the function is given incorrectly. Function call [d] is invalid because only two arguments are passed to the function, but there are three input arguments in the function header.




6.2 MATLAB PROGRAM ORGANIZATION

Typically, a MATLAB program consists of a script that calls functions to do the actual work.




6.2.1 Modular Programs

A modular program is a program in which the solution is broken down into modules, and each is implemented as a function. The script that calls these functions is typically called the main program.

To demonstrate the concept, we will use the very simple example of calculating the area of a circle. In [Section 6.3](#), a much longer example will be given. For this example, there are three steps in the algorithm to calculate the area of a circle:

-  Get the input (the radius)
-  Calculate the area
-  Display the results

In a modular program, there would be one main script (or, possibly a function instead) that calls three separate functions to accomplish these tasks:

-  A function to prompt the user and read in the radius
-  A function to calculate and return the area of the circle
-  A function to display the results

Assuming that each is stored in a separate code file, there would be four separate code files altogether for this program; one script file and three function code files, as follows:

calcandprintarea.m

```
% This is the main script to calculate the
%   area of a circle
% It calls 3 functions to accomplish this
radius=readradius;
area=calcarearea(radius);
printarea(radius,area)
```

readradius.m

```
function radius=readradius
% readradius prompts the user and reads the radius
% Ignores error-checking for now for simplicity
% Format: readradius or readradius()

disp('When prompted, please enter the radius in inches.')
radius=input('Enter the radius: ');
end
```

calcarearea.m

```
function area=calcarearea(rad)
% calcarea returns the area of a circle
% Format: calcarea(radius)

area=pi*rad .* rad;
end
```

printarea.m

```
function printarea(rad,area)
% printarea prints the radius and area
% Format: printarea(radius, area)

fprintf('For a circle with a radius of %.2f inches.\n',rad)
fprintf('the area is %.2f inches squared.\n',area)
end
```

When the program is executed, the following steps will take place:

- ✚ the script **calcandprintarea** begins executing
- ✚ **calcandprintarea** calls the **readradius** function
 - **readradius** executes and returns the radius
- ✚ **calcandprintarea** resumes executing and calls the **calcarearea** function, passing the radius to it
 - **calcarearea** executes and returns the area
- ✚ **calcandprintarea** resumes executing and calls the **printarea** function, passing both the radius and the area to it
 - **printarea** executes and prints
- ✚ the script finishes executing

Running the program would be accomplished by typing the name of the script;
this would call the other functions:

```
>> calcandprintarea
When prompted, please enter the radius in inches.
Enter the radius: 5.3
For a circle with a radius of 5.30 inches,
the area is 88.25 inches squared.
```

Note how the function calls and the function headers match up. For example:
readradius function:

```
function call: radius=readradius;
function header: function radius=readradius
```

In the **readradius** function call, no arguments are passed so there are no input arguments in the function header. The function returns one output argument so that is stored in one variable.

calcare function:

```
function call: area=calcare(radius);  
function header: function area=calcare(rad)
```

In the *calcare* function call, one argument is passed in parentheses so there is one input argument in the function header. The function returns one output argument so that is stored in one variable.

printarea function:

```
function call: printarea(radius,area)  
function header: function printarea(rad,area)
```

In the *printarea* function call, there are two arguments passed, so there are two input arguments in the function header. The function does not return anything, so the call to the function is a statement by itself; it is not in an assignment or output statement.

PRACTICE 6.4

Modify the *readradius* function to error-check the user's input to make sure that the radius is valid. The function should ensure that the radius is a positive number by looping to print an error message until the user enters a valid radius.

6.2.2 Local Functions

Thus far, every function has been stored in a separate code file. However, it is possible to have more than one function in a given file. For example, **if** one function calls another, the first (calling) function would be the **main function** and the function that is called is a **local function**, or sometimes a **subfunction**. These functions would both be stored in the same code file, first the main function and then the local function. The name of the code file would be the same as the name of the main function, to avoid confusion.

To demonstrate this, a program that is similar to the previous one, but calculates and prints the area of a rectangle, is shown here. The script first calls a function that reads the length and width of the rectangle, and then calls a function to print the results. This function calls a local function to calculate the area.

rectarea.m

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

readlenwid.m

```
function [l,w]=readlenwid  
% readlenwid reads & returns the length and width  
% Format: readlenwid or readlenwid()  
  
l = input('Please enter the length: ');  
w = input('Please enter the width: ');  
end
```

printrectarea.m

```
function printrectarea(len, wid)
% printrectarea prints the rectangle area
% Format: printrectarea(length, width)

% Call a local function to calculate the area
area = calcrectarea(len,wid);
fprintf('For a rectangle with a length of %.2f\n',len)
fprintf('and a width of %.2f, the area is %.2f\n', ...
    wid, area);

end

function area=calcrectarea(len, wid)
% calcrectarea returns the rectangle area
% Format: calcrectarea(length, width)
area = len * wid;
end
```

An example of running this program follows:

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

Note how the function calls and function headers match up. For example:

readlenwid function:

```
function call: [length, width]=readlenwid;
function header: function [l,w]=readlenwid
```

In the *readlenwid* function call, no arguments are passed so there are no input arguments in the function header. The function returns two output arguments so there is a vector with two variables on the left side of the assignment statement in which the function is called.

printrectarea function:

```
function call: printrectarea(length, width)
function header: function printrectarea(len, wid)
```

In the *printrectarea* function call, there are two arguments passed, so there are two input arguments in the function header. The function does not return anything, so the call to the function is a statement by itself; it is not in an assignment or output statement.

calcrectarea local function:

```
function call: area=calcrectarea(len,wid);
function header: function area=calcrectarea(len, wid)
```

In the *calcrectarea* function call, two arguments are passed in parentheses so there are two input arguments in the function header. The function returns one output argument so that is stored in one variable.

The help command can be used with the script *rectarea*, the function *readlenwid*, and with the main function, *printrectarea*. To view the first comment in the local function, as it is contained within the *printrectarea.m* file, the operator *>* is used to specify both the main and local functions:

```
>> help rectarea
This program calculates & prints the area of a rectangle

>> help printrectarea
printrectarea prints the rectangle area
Format: printrectarea(length, width)

>> help printrectarea>calcrectarea
calcrectarea returns the rectangle area
Format: calcrectarea(length, width)
```

So, local functions can be in script code files, and also in function code files

PRACTICE 6.5

For a right triangle with sides a , b , and c , where c is the hypotenuse and θ is the angle between sides a and c , the lengths of sides a and b are given by:

$$\begin{aligned}a &= c \cdot \cos(\theta) \\ b &= c \cdot \sin(\theta)\end{aligned}$$

Write a script *righttri* that calls a function to prompt the user and read in values for the hypotenuse and the angle (in radians), and then calls a function to calculate and return the lengths of sides a and b , and a function to print out all values in a sentence format. For simplicity, ignore units. Here is an example of running the script; the output format should be exactly as shown here:

```
>> righttri
Enter the hypotenuse: 5
Enter the angle: .7854
For a right triangle with hypotenuse 5.0
and an angle 0.79 between side a & the hypotenuse,
side a is 3.54 and side b is 3.54
```

For extra practice, do this using two different program organizations:

- One script that calls three separate functions, each stored in separate code files
 - One script that calls two functions; the function that calculates the lengths of the sides will be a local function to the function that prints
-

APPLICATION: MENU-DRIVEN MODULAR PROGRAM

Many longer, more involved programs that interact with the user are menu driven, which means that the program prints a menu of choices and then continues to loop to print the menu of choices until the user chooses to end the program. A modular menu-driven program would typically have a function that presents the menu and gets the user's choice, as well as functions to implement the action for each choice. These functions may have local functions. Also, the functions would error-check all user input.

As an example of such a menu-driven program, we will write a program to explore the constant e .

The constant e , called the natural exponential base, is used extensively in mathematics and engineering. There are many diverse applications of this constant. The value of the constant e is approximately 2.7183... Raising e to the power of x , or e^x , is so common that this is called the exponential function. In MATLAB, as we have seen, there is a function for this, `exp`.

One way to determine the value of e is by finding a limit.

$$e = \lim_{n \rightarrow \infty} \left(1 + \frac{1}{n}\right)^n$$

As the value of n increases toward infinity, the result of this expression approaches the value of e .

An approximation for the exponential function can be found using what is called a Maclaurin series:

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

We will write a program to investigate the value of e and the exponential function. It will be menu-driven. The menu options will be:

- Print an explanation of e .
- Prompt the user for a value of n and then find an approximate value for e using the expression $(1+1/n)^n$
- Prompt the user for a value for x . Print the value of $\exp(x)$ using the builtin function. Find an approximate value for e^x using the Maclaurin series just given.
- Exit the program.

The algorithm for the script main program follows:

- Call a function `eoption` to display the menu and return the user's choice.
- Loop until the user chooses to exit the program. If the user has not chosen to exit, the action of the loop is to:
- Depending on the user's choice, do one of the following:

- Call a function **explaine** to print an explanation of e .
- Call a function **limite** that will prompt the user for n and calculate an approximate value for e
- Prompt the user for x and call a function **expfn** that will print both an approximate value for e^x and the value of the built-in $\exp(x)$. Note that because any value for x is acceptable, the program does not need to error-check this value.
- Call the function **eoption** to display the menu and return the user's choice again.

The algorithm for the **eoption** function follows:

- Display the four choices.
- Error-check by looping to display the menu until the user chooses one of the four options.
- Return the integer value corresponding to the choice

The algorithm for the **explaine** function is:

- Print an explanation of e , the \exp function, and how to find approximate values.

The algorithm for the **limite** function is:

- Call a local function **askfor** to prompt the user for an integer n .
- Calculate and print the approximate value of e using n .

The algorithm for the local function **askfor** is:

- Prompt the user for a positive integer for n .
- Loop to print an error message and reprompt until the user enters a positive integer.
- Return the positive integer n .

The algorithm for the **expfn** function is:

- Receive the value of x as an input argument.
- Print the value of $\exp(x)$.
- Assign an arbitrary value for the number of terms n (an alternative method would be to prompt the user for this).
- Call a local function **appex** to find an approximate value of $\exp(x)$ using a series with n terms.
- Print this approximate value.

The algorithm for the local function **appex** is:

- Receive x and n as input arguments.
- Initialize a variable for the running sum of the terms in the series (to 1 for the first term) and for a running product that will be the factorials in the denominators.
- Loop to add the n terms to the running sum.
- Return the resulting sum.

The entire program consists of the following script file and four function code files:

eapplication.m

```
% This script explores e and the exponential function
% Call a function to display a menu and get a choice
choice=eoption;
% Choice 4 is to exit the program
while choice ~= 4
    switch choice
        case 1
            % Explain e
            explaine;
        case 2
            % Approximate e using a limit
            limite;
        case 3
            % Approximate exp(x) and compare to exp
            x=input('Please enter a value for x: ');
            expfn(x);
    end
    % Display menu again and get user's choice
    choice=eoption;
end
```

eoption.m

```
function choice=eoption
% eoption prints a menu of options and error-checks
% until the user chooses one of the options
% Format: eoption or eoption()

printchoices
choice=input("");
while ~any(choice==1:4)
    disp('Error - please choose one of the options.')
    printchoices
    choice=input("");
end
end

function printchoices
fprintf('Please choose an option:\n\n');
fprintf('1) Explanation\n')
fprintf('2) Limit\n')
fprintf('3) Exponential function\n')
fprintf('4) Exit program\n\n')
end
```

explaine.m

```
function explaine
% explaine explains a little bit about e
% Format: explaine or explaine()

fprintf('The constant e is called the natural')
fprintf(' exponential base.\n')
fprintf('It is used extensively in mathematics and')
fprintf(' engineering.\n')
fprintf('The value of the constant e is ~ 2.7183\n')
fprintf('Raising e to the power of x is so common that')
fprintf(' this is called the exponential function.\n')
fprintf('An approximation for e is found using a limit.\n')
fprintf('An approximation for the exponential function')
fprintf(' can be found using a series.\n')
end
```

limite.m

```
function limite
% limite returns an approximate of e using a limit
% Format: limite or limite()

% Call a local function to prompt user for n
n = askforn;
fprintf('An approximation of e with n=%d is %.2f\n', ...
    n, (1+1/n) ^ n)
end

function outn=askforn
% askforn prompts the user for n
% Format askforn or askforn()
% It error-checks to make sure n is a positive integer

inputnum=input('Enter a positive integer for n: ');
num2=int32(inputnum);
while num2 ~= inputnum || num2 < 0
    inputnum=input('Invalid! Enter a positive integer:');
    num2=int32(inputnum);
end
outn=inputnum;
end
```

expfn.m

```
function expfn(x)
% expfn compares the built-in function exp(x)
% and a series approximation and prints
% Format expfn(x)

fprintf('Value of built-in exp(x) is %.2f\n',exp(x))

% n is arbitrary number of terms
n = 10;
fprintf('Approximate exp(x) is %.2f\n', appex(x,n))
end

function outval=appex(x,n)
% appex approximates e to the x power using terms up to
% x to the nth power
% Format appex(x,n)

% Initialize the running sum in the output argument
% outval to 1 (for the first term)
outval=1;

for i=1:n
    outval=outval+(x^i)/factorial(i);
end
end
```



Running the script will bring up the menu of options.

```
>> eapplication
Please choose an option:

1) Explanation
2) Limit
3) Exponential function
4) Exit program
```

Then, what happens will depend on which option(s) the user chooses. Every time the user chooses, the appropriate function will be called and then this menu will appear again. This will continue until the user chooses 4 for 'Exit Program'. Examples will be given of running the script, with different sequences of choices.

In the following example, the user

-  Chose 1 for 'Explanation'.
-  Chose 4 for 'Exit Program'.

```
>> eapplication
Please choose an option:




1) Explanation
2) Limit
3) Exponential function
4) Exit program

1
The constant e is called the natural exponential base.
It is used extensively in mathematics and engineering.
The value of the constant e is ~ 2.7183
Raising e to the power of x is so common that this is called the
exponential function.
An approximation for e is found using a limit.
An approximation for the exponential function can be found using a
series.
Please choose an option:

1) Explanation
2) Limit
3) Exponential function
4) Exit program

4
```

In the following example, the user

-  Chose 2 for 'Limit'.
-  When prompted for n, entered two invalid values before finally entering a valid positive integer.
-  Chose 4 for 'Exit Program'.

```
>> eapplication
Please choose an option:
1) Explanation
2) Limit
3) Exponential function
4) Exit program

2
Enter a positive integer for n:-4
Invalid! Enter a positive integer: 5.5
Invalid! Enter a positive integer: 10
An approximation of e with n=10 is 2.59
Please choose an option:
1) Explanation
2) Limit
3) Exponential function
4) Exit program

4
```

To see the difference in the approximate value for e as n increases, the user kept choosing 2 for 'Limit', and entering larger and larger values each time in the following example (the menu is not shown for simplicity):

```
>> eapplication
Enter a positive integer for n: 4
An approximation of e with n=4 is 2.44
Enter a positive integer for n: 10
An approximation of e with n=10 is 2.59
Enter a positive integer for n: 30
An approximation of e with n=30 is 2.67
Enter a positive integer for n: 100
An approximation of e with n=100 is 2.70
```

In the following example, the user

- Chose 3 for 'Exponential function'.
 - When prompted, entered 4.6 for x .
- Chose 3 for 'Exponential function' again.
 - When prompted, entered -2.3 for x .
- Chose 4 for 'Exit Program'.

Again, for simplicity, the menu options and choices are not shown.

```
>> eapplication
Please enter a value for x: 4.6
Value of built-in exp(x) is 99.48
Approximate exp(x) is 98.71
Please enter a value for x:-2.3
Value of built-in exp(x) is 0.10
Approximate exp(x) is 0.10
```