

# ENGR101 Lab 4

[Part 1: File Input/Output](#)

[Part 2: Image Functions](#)

[Part 3: Debugging](#)

[Exercise 1: File Input/Output](#)

[Exercise 2: Changing the image](#)

[Exercise 3: Debugging master](#)

## Submission Requirements:

*All labs are due at 11:59pm the day after your lab section.*

- Exercise 1: Save your work as lab4\_1.m
- Exercise 1: Save your outputs as lab4\_1.mat
- Exercise 2: Save your work as lab4\_2.m
- Exercise 2: Save your output as lab4\_2.jpg
- Exercise 3: Save your work as daysLeftInMonth.m
- Exercise 3: Save your work as isLeapYear.m
- Exercise 3: Save your output as lab4\_3.txt

## Reminder from Lab 3

Recall from Lab 3, we experienced writing our user-defined functions with specified inputs and specified outputs, working with logical arrays, and using logical arrays to select data. In this lab, we will be exploring more selection and some operations with file input and output.

## Part 1: Basic File I/O

Not all files are .m files and Matlab gives us the flexibility to work with a wide variety of different files. This includes files like .txt, .xlsx, and .csv just to name a few. In order for Matlab to be able to use data from these files, we first need to import the data stored in these files into our workspace. In this lab, we'll learn about the **load**(filename) function for importing data to Matlab and the **save**(filename) function for exporting data from Matlab.

In the example below, **load**(filename) loads data from filename 'C.mat'. If the filename is a MAT-file, then **load** will copy variables from the MAT-file into the MATLAB® workspace. MAT-files are a data type that Matlab uses to store information; text, numbers and a mix of both can be saved to and loaded from a .mat file.

As shown below, **save(filename)** saves all variables from the current workspace in a MATLAB® formatted binary file (MAT-file) called 'C.mat'. If the filename exists, **save** overwrites the file.

```
>> clear %cleans out everything in the workspace
>> A = [3:5:80]; %creates a vector A
>> B = [4:16:79]; %creates a vector B
>> save('C.mat') %this saves everything currently in the
                  %workspace(A and B) as 'C.mat'

>> clear
>> load('C.mat')
>> whos
```

Name	Size	Bytes	Class	Attributes
A	1x16	128	double	
B	1x5	40	double	

## Part 2: Image Functions

Each color that we see on the computer screen is actually a combination of the colors red, green, and blue (RGB). Each pixel of an image is actually a 1x3 vector of its RGB values. For example, [0, 0, 0] corresponds to black, [255, 255, 255] corresponds to white, [255, 0, 0] represents red, [0, 255, 0] represents green, and [0, 0, 255] represents blue. Matlab has the ability to read in images as a 3D array using the function **imread(filename)**. Using the image *download.jpg* below, we can use **imread** to see the corresponding RGB values for the section of the image to which we are referring. When using the **imread** function, be sure to use a semicolon! This ensures that the matrix is suppressed and not printed in the command window.



*download.jpg*

The output from the **imread** function is a 3D matrix, with size [X, Y, 3]. X is the height of the image, Y is the width, and 3 is the depth of the image corresponding to the RGB values at each row by column element. In the example below, the first 10x10 matrix gives the R(ed) value of rows 50 to 60 and columns 200 to 210, second for G(reen) and third for B(lue). We can apply any matrix calculation on these matrices. **Remember:** there are differences between regular matrix operations (\*, /, ^) and element-wise operations(.\*, ./, .^).

```
>> C = imread('download.jpg');
>> C(50:60, 200:210, :)
```

```
ans(:,:,1) =
```

13	11	11	11	10	9	8	8	8	9	9
11	11	11	11	10	10	9	9	9	9	9
11	11	11	11	10	10	9	9	9	9	9
10	10	10	10	9	10	9	9	9	9	9
9	10	10	10	9	10	9	9	9	9	9
9	9	9	10	9	10	9	9	9	9	9
9	9	9	10	9	10	9	9	9	9	9
6	14	17	17	10	16	19	4	3	9	9
17	13	10	11	8	11	16	16	19	9	9
14	14	11	15	18	9	4	5	1	9	9
21	14	5	2	7	0	1	21	22	10	10

```
ans(:,:,2) =
```

9	9	9	9	8	8	7	6	7	8	8
9	8	8	8	7	6	5	4	5	7	8
8	8	8	8	7	6	5	5	5	8	8
10	10	10	10	9	8	8	8	7	8	8
10	9	9	10	9	9	8	8	8	8	8
8	8	8	8	7	7	6	7	7	8	8
8	8	8	7	7	6	5	6	6	8	8
6	12	15	15	8	14	17	2	1	8	8
16	12	9	10	6	9	14	14	17	8	8
12	12	9	15	18	9	4	7	2	8	8
17	10	4	1	8	0	1	23	23	9	9

```
ans(:,:,3) =
```

10	10	10	12	11	14	13	17	15	16	14
14	15	17	19	18	21	20	24	20	18	14
15	15	17	19	18	20	19	20	19	16	14
12	12	12	12	9	9	6	6	8	13	14
15	15	14	12	9	7	4	4	6	13	14
24	24	22	21	18	16	13	12	12	14	14
24	26	24	24	21	21	19	17	15	16	14
14	25	28	29	22	28	31	16	14	16	14
21	18	17	18	17	20	25	25	28	16	14
15	15	10	15	18	9	4	6	4	13	14
18	9	2	0	3	0	0	22	25	14	15

We can also change images using matrix operations. For example, we can intensify the R(ed) values, make everything lighter or darker, or completely change the color scale of the original image. We can use the function ***imwrite***(C, newfilename) to save the image into the file named newfilename.

Function	Use Case	Example
<b><i>load</i></b> (filename)	Loads filename.	load('myfile.txt')
<b><i>save</i></b> (filename)	Saves filename.	save('myfile.txt')
<b><i>imread</i></b> (filename)	Reads the image as a 3D RGB matrix.	imread('myimg.jpeg')
<b><i>imwrite</i></b> (C, newfilename)	Saves the manipulated 3D matrix C as a new image.	imwrite(C, 'mynewimg.jpeg')

**Note:** We need single quotes around the filename when implementing the above functions

## Part 3: Debugging

There are many different errors that may occur when writing code: incorrect syntax, flawed logic, the list goes on and on. Some are easier to spot than others. Below are a few of the steps you can take in debugging your code. Ask your instructor for further demos, and you'll get to practice in one of the exercises today.

### Get your code to run:

You may have noticed that MATLAB has an automatic "spell-check" function in the editor. If you hover your cursor over the part of the code that is underlined, MATLAB will give you an explanation of why an error or a warning is appearing.

1. **Errors** are underlined in red and create a red line to the right side: these must be fixed or the code will not run.
2. **Warnings** are underlined in orange and create an orange line to the right side: **these do not have to be fixed**, but MATLAB is indicating that something may be incorrect.

For example:



```

1  function [hypotenuse] = pythag(leg1,leg2)
2      hyp = sqrt(leg1.^2 + leg2.^2);
3
4  end
  
```

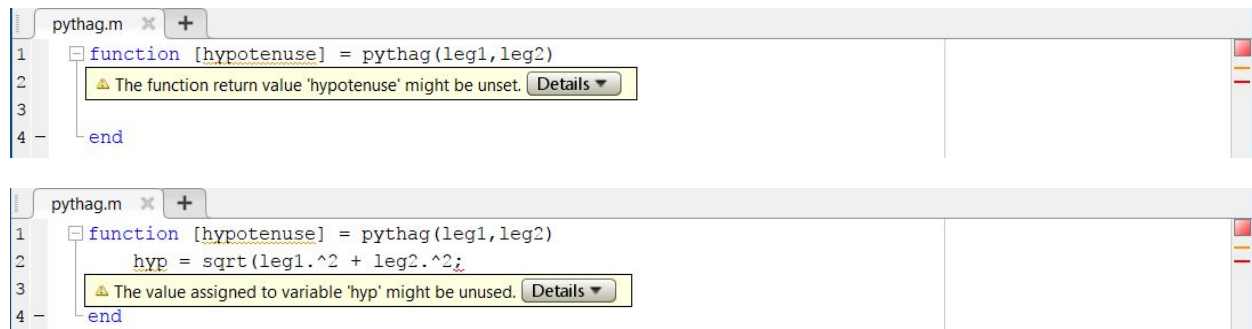
The screenshot shows the MATLAB editor window for a file named 'pythag.m'. The code is as follows:

```

1  function [hypotenuse] = pythag(leg1,leg2)
2      hyp = sqrt(leg1.^2 + leg2.^2);
3
4  end
  
```

On line 2, the variable 'hyp' is underlined in orange, indicating a warning. A red line is visible on the right side of the editor window.

The above function receives the following warnings:



The first screenshot shows the MATLAB code editor with the file 'pythag.m' open. The code is as follows:

```
1 function [hypotenuse] = pythag(leg1,leg2)
2
3
4 end
```

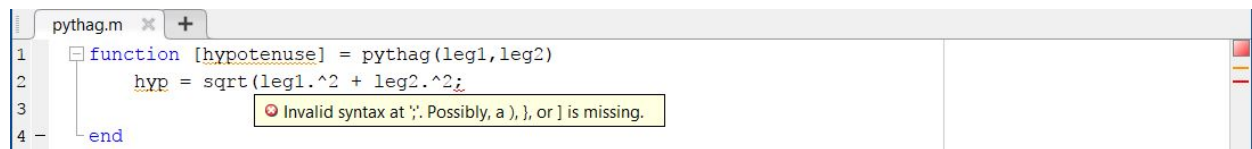
A warning message is displayed: "The function return value 'hypotenuse' might be unset." with a 'Details' button.

The second screenshot shows the same code editor with the following code:

```
1 function [hypotenuse] = pythag(leg1,leg2)
2     hyp = sqrt(leg1.^2 + leg2.^2);
3
4 end
```

A warning message is displayed: "The value assigned to variable 'hyp' might be unused." with a 'Details' button.

The above function receives the following errors:



The screenshot shows the MATLAB code editor with the file 'pythag.m' open. The code is as follows:

```
1 function [hypotenuse] = pythag(leg1,leg2)
2     hyp = sqrt(leg1.^2 + leg2.^2;
3
4 end
```

An error message is displayed: "Invalid syntax at ':'. Possibly, a ), }, or ] is missing."

From the warning and error messages, we now know that we can fix the *pythag* function by changing *hyp* to the correct output (*hypotenuse*), and adding the missing parenthesis in line 2.

Additionally, proper commenting and style are essential to successful debugging (i.e. if we can't tell what the code should be doing, then how can we expect to fix it?). In the code above, we should add a comment in line 2 that states what the next line is doing. For example: Use the Pythagorean theorem to calculate the hypotenuse length.

### Disp and assert:

So now your code is running, but it is not giving you correct output. You need to find the place(s) in your code where MATLAB is doing something different than you think. Oftentimes, when you have complicated code, it is useful to know where MATLAB is looking within your code and where the error is located.

You can print out values, either by using **disp(variable)** or removing semicolons. [Remember semicolons suppress output. So by removing them, output is displayed in the command window.]

If you can't identify the source of the problem, printing arbitrary numbers or the state of a variable can help! You can print numbers such as 1, 2, or 3 to check progress through the code. Better still, you can print the value of the variable at different points in the code and check if the code correctly calculated the value.

**assert(condition, charVector)** will check a condition. If the condition is false, it will end your function/script and print the message to the command window.

```
function [area, hypotenuse] = triangle(leg1,leg2)

    assert(leg1>0, 'leg1 is negative');
    assert(leg2>0, 'leg2 is negative');

    %calculate area of triangle
    area = 0.5 .* leg1 .* leg2;

    %pythagorean theorem
    hypotenuse = sqrt(leg1.^2 + leg2.^2);

end
```

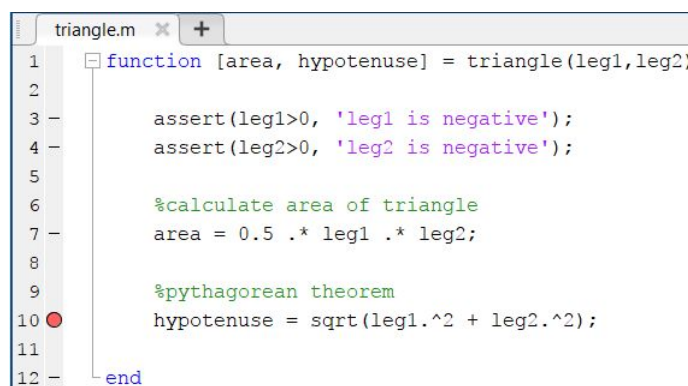
When trying to run it:

```
>> triangle(-5,10)
Error using triangle (line 3)
leg1 is negative
```

## The Debugger:

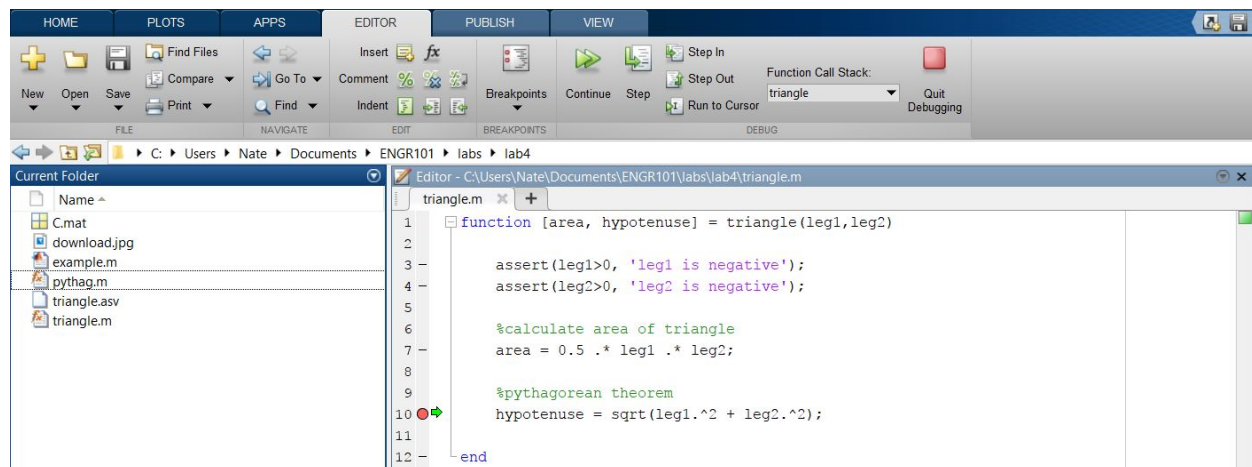
MATLAB has a handy debugging feature. It allows you to set “breakpoints” within your code and see the variables at that point before continuing. Remember, we don’t typically see the variable that a function knows at any given point - this allows us to look inside a MATLAB function (or script) while it is working!

Breakpoints are set by clicking once on the line to the left of your code. This will “pause” your code from being executed at the point just before that line of code (in the picture below, just before line 10).

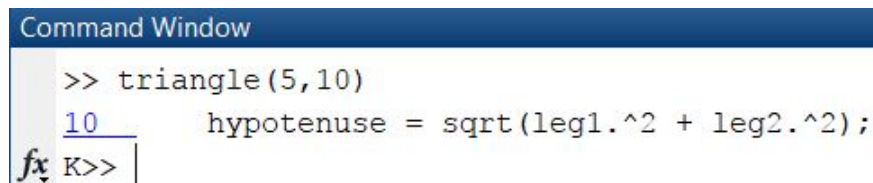


Once you have a breakpoint set, run your script/function. Remember, functions must run with the specified input. In this case we would test our function through the command window (i.e. *triangle(5, 10)*). If we just use the green “Run” button, we would receive an error.

While running the script/function, the editor menu will change to look like this:



- The green arrow in the script shows at which line of code MATLAB is currently “paused”
- The “Continue” double arrows will continue to run the code until it reaches any subsequent breakpoint.
- “Step” runs one and only one line of code: hitting step in the picture above would execute line 10 of the code and then pause.
- You’ll noticed that in Debugging mode, while the program is “paused”, the Command Line prompt changes from >> to K>>



- At any point while the program is “paused”, you can display or edit the variables in the current Workspace using the Command Line.
- If you want to exit this mode, use “Quit Debugging”.

Debugging is an acquired skill, and it takes lots of practice. It will get less painful as you accept the fact that it is inevitable, so take a deep breath and relax. You’ll get through it!

*Before beginning each exercise, be sure to clear your workspace!*

### **Exercise 1: File Input/Output (I/O)**

*Download `lab4_1_water.mat`, `lab4_1.m`.*

In this exercise you will be working with a set of data from the Flint Service Line, the data we got from Flint for their water crisis. The Michigan Data Science Team was given this data set to use machine learning and other methods to predict potential areas that have water quality issues. You will use a combination of file I/O and logical arrays to do the requested analysis. Detailed instructions can be found in `lab4_1.m`.

### **Exercise 2: Changing the image**

*Download `lab4_2.m`, `lab4_2.png`.*

As we saw in Lab 2, images can be quantified with different color space models (i.e. RGB, HSV, etc.). In this exercise, you will be using the image functions to manipulate the RGB values and to create a new image that has a different color scale. Detailed instructions can be found in `lab4_2.m`.

### **Exercise 3: Debugging Master**

*Download `isLeapYear.m`, `daysLeftInMonth.m`, and `lab4_3.txt`.*

The purpose of this exercise is to learn how to use the debugger to fix bugs in your programs. You must follow these steps exactly and answer the questions stated in `lab4_3.txt`. *Attach the completed files to your Canvas submission.*

**Make sure not to enter any extra lines into either function; otherwise the line numbers will not match and you might have to start over from the beginning.**

1. Before we start debugging, look over the code in both `daysLeftInMonth` and `isLeapYear` to get an idea of what they are trying to accomplish. If you find any bugs, **do not modify the code yet**.

*Fixing the first bug:*

2. Run `daysLeftInMonth` with the inputs shown below. If the function is working properly, it would return 27. However, since it isn't working correctly, we have found a bug that we need to fix.

```
>> d = daysLeftInMonth(7, 4, 1994)
```



- Put a breakpoint on line 14 of *daysLeftInMonth* and run your function again with the same inputs.

```
11 %Change February if leap year
12 daysInMonth(2) = 28 + leapYear;
13
14 daysLeft = daysInMonth(month) - day;
15 end

>> d = daysLeftInMonth(7, 4, 1994)
7 daysLeft = daysInMonth(month) - day;
K>>
```

We have now paused execution inside the function. Don't forget to write down these answers for *lab4\_3.txt*.

- What variables are currently stored in the MATLAB workspace?
  - What are the values currently stored in the *daysInMonth* vector?
- Remove all breakpoints and hit **continue** or **quit debugging** to return to a normal state.
  - Let's figure out how to fix the bug.
    - What values should the *daysInMonth* vector hold? (Hint: [how many days are in each month?](#))
    - Modify *daysLeftInMonth* and run again with the same inputs to confirm that *daysInMonth* now holds the correct value for the month of July and returns the correct number of days left in the month (27).

*Fixing the second bug:*

- Remove all breakpoints and run *daysLeftInMonth* with the new inputs shown below. The function would return 15 if it was working properly.

```
>> d = daysLeftInMonth(2, 14, 2000)
```
- 2000 was a [leap year](#) (i.e. February had 29 days instead of 28), so we expect *leapYear* to be set to 1 on line 9 of *daysLeftInMonth*. Put a breakpoint on line 12 of *daysLeftInMonth* and run the function again with the same inputs.
  - What are the current values in the *daysInMonth* vector?
  - What is the current value of *leapYear*?
- Remove all breakpoints and hit **continue** or **quit debugging** to return to a normal state.

Note: Below are the characteristics of a leap year

- A year IS a leap year if the year is divisible by 4
- BUT, a year IS NOT a leap year if the year is divisible by 100
- EXCEPT, a year IS a leap year if the year is divisible by 400

For example:

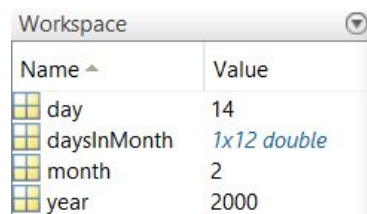
- **1996 WAS** a leap year because it is divisible by 4.
- **1900 WAS NOT** a leap year because it is divisible by 100 (but not 400).
- **1600 WAS** a leap year because it is divisible by 400.

9. We now know that the *isLeapYear* function is broken because it returned 0 when it should have returned 1 for the year 2000. We need to go inside that function to see what is going on. However, before we continue, let's do a quick exercise on scope.

- Set breakpoints at lines 9 and 14 in *daysLeftInMonth* and line 9 of *isLeapYear*
- Run the *daysLeftInMonth* function again with the below inputs:

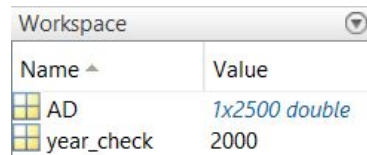
```
>> d = daysLeftInMonth(2, 14, 2000)
```

- Notice when we pause at line 9 that our workspace contains the following variables:



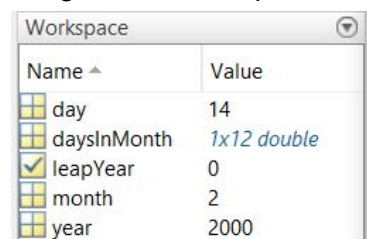
Name ^	Value
day	14
daysInMonth	1x12 double
month	2
year	2000

- Now click the continue button and see what happens. We have now entered the *isLeapYear* function. Notice how our workspace currently **only** contains variables used in the *isLeapYear* function.



Name ^	Value
AD	1x2500 double
year_check	2000

- Click continue again and observe how the workspace changes. Our workspace is back to how it looked at the start, but with a new variable (leapYear) that is assigned to the output of the *isLeapYear* function.



Name ^	Value
day	14
daysInMonth	1x12 double
leapYear	0
month	2
year	2000

See how the workspace no longer contains any variables used in the *isLeapYear* function? The output of the function has been assigned to the variable *leapYear*, but we no longer have access to any of the specific variables used inside of *isLeapYear*. This is why MATLAB does not allow us to reference specific variables outside of their assigned function.

10. Ok - let's get back to debugging. Hit **continue** or **quit debugging** to return to a normal state.

11. Remove the breakpoints at lines 9 and 14 of *daysLeftInMonth*, but keep the breakpoint at line 9 in *isLeapYear*. Run the *daysLeftInMonth* function again with the below inputs:

```
>> d = daysLeftInMonth(2, 14, 2000)
```

- a. What is the current value of AD(2000)?  
*Hint:* (type *AD(2000)* in your command window to find out)
- b. What should the value of AD(2000) be?
- c. According to the characteristics in step 8, will the year 2400 be a leap year?
- d. According to the characteristics in step 8, will the the year 2100 a leap year?

*Hint:* [Exactly Which Years are Leap Years?](#)

12. Modify *isLeapYear* such that the leap year values are correct, remove all breakpoints, and run the function again to ensure it is now working properly.