# Lab 2 MATLAB and Image Processing

In this Lab, we'll explore more functions/tools in MATLAB which facilitate image processing and analysis. This lab will contain:

- More basic knowledge in MATLAB
  - Control structures
  - Graph plotting
- Basics of image processing

## Part 1 Control Structures

- Changing the flow of control is common in programming. They are necessary for two reasons.

  (1) **Selection structures**: You may want to execute a part of the code under a certain condition only;

  (2) **Iteration structures**: You may want to repeat a code segment for certain number of times.

- MATLAB uses the occurrence of keywords to define the extent of code blocks. Keywords like if, switch, while, for, case, else, elseif, and end are identified with blue coloring by the MATLAB text editor. Below is an example:

ControlStructure_Tutorial.m

```
1  nrows=4;
2  ncols=6;
3  A=ones(nrows,ncols);
4
5  for c=1:ncols
6      for r=1:nrows
7          if r==c
8              A(r,c)=2;
9          elseif abs(r-c)==1
10             A(r,c)=-1;
11         else
12             A(r,c)=0;
13         end
14     end
15 end
```

These keywords are not part of the code block, but they serve as

- instructions on what to do with the code block
- delimiters that define the extent of the code block

## Logical Operations

Logical operations can be performed element-by-element on two vectors as long as both vectors are the same length, or one vector is a scalar. The result will be a vector of logical values with the same length as the original vectors.

| | |
|---|---|
| **<** | **Less than** |
| <= | **Less than or equal to** |
| > | **Greater than** |
| >= | **Greater than or equal to** |
| == | **Equal to** |
| ~= | **Not equal to** |

In the command window, enter the followings:

```
>> A = [2, 5, 7, 1, 3]

>> B = [0, 6, 5, 3, 2]

>> A >= 5

>> A >= B
```

## Logical Operators: Short-Circuit &&, ||

```
% Syntax

expression1 && expression2

expression1 || expression2
```

- expression1 && expression2 represents a logical AND operation that employs short-circuiting behavior. That is, expr2 is not evaluated if expr1 is logical 0 (false). Each expression must evaluate to a scalar logical result.
- expression1 || expression2 represents a logical OR operation that employs short-circuiting behavior. That is, expr2 is not evaluated if expr1 is logical 1 (true). Each expression must evaluate to a scalar logical result.

Specify a logical statement where the second condition depends on the first. In the following statement, it doesn't make sense to evaluate the relation on the right if the divisor, b, is zero.

```
b = 1;

a = 20;
```

```
x = (b ~= 0) && (a/b > 18.5)
```

## More Logical Operations

| | |
|---|---|
| and | Find logical AND |
| not | Find logical NOT |
| or | Find logical OR |
| xor | Find logical exclusive-OR |
| all | Determine if all array elements are nonzero or true |
| any | Determine if any array elements are nonzero |
| False | Logical 0 (false) |
| true | Logical 1 (true) |
| Find | Find indices and values of nonzero elements |
| islogical | Determine if input is logical array |
| logical | Convert numeric values to logicals |

## 1.1 Selection Structures

Also known as a conditional structure, a selection structure is a programming feature that performs different processes based on whether a boolean condition is true or false. Selection structures use relational operators(logical) to test conditions. There are different types of selection structures that can be used to achieve different outcomes.

### 1.1.1 if statements

We use if, elseif else and end to execute statements if condition is true. The Syntax looks like: (logical expression should be )

```
if <logical expression 1>
    <code block 1>
elseif <logical expression 2>
    <code block 2>
    …
elseif <logical expression n>
    <code block n>
else
    <default code block>
end
```

For example, the following code segment determines whether a day is weekday or a weekend day.

```
day=2;
if day==6||day==7
    state='weekend';
elseif day==1||day==2||day==3||day==4||day==5
    state='weekday';
else
    state ='wrong number';
```

```
end
```

### 1.1.2 switch statements

We use switch, case, and otherwise to execute one of several groups of statements. The syntax looks like,

```
switch <parameter>
    case <case specification 1>
        <code block 1>
    case <case specification 2>
        <code block 2>
…
    case <case specification n>
        <code block n>
    otherwise
        <default code block>
end
```

Note that all tests refer to the value of the same parameter. ‘case’ specifications may be either a single value or a set of parameters enclosed in braces {…}. ‘otherwise’ specifies the code block to be executed when none of the case value apply.

For example, the following code segment determines the number of days in a month.

```
month=4;
leapYear=false;
switch month
    case {4,6,9,11} % Apr, June, Sept or Nov
        days=30;
    case 2 % Feb
        if leapYear % whether it is leap Year
```

```
            days=29;
        else
            days=28;
        end
    case{1,3,5,7,8,10,12} % Other months
        days=31;
    otherwise
        error('bad month index');
end
```

## 1.2 Iteration Structures

Iteration allows controlled repetition of a code block. Control statements at the beginning of the code block specify the manner and extent of the repetition.

- The *for* loop is designed to repeat its code block a fixed number of times and largely automates the process of managing the iteration.
- The *while* loop is more flexible. In contrast to the fixed repetition of the *for* loop, its code block can be repeated a variable number of times, depending on the values of data being processed.

### 1.2.1 for statements

```
for <variable specification>
    <code block>
end
```

The core concept in the MATLAB *for*-loop implementation is in the style of the <variable specification>, which is accomplished as follows:

<variable> = <vector>

where <variable> is the name of the loop control variable and <vector> is any vector.

Examine the following code example that finds the maximum value in a vector.

```
A = [6, 12, 6, 91, 13, 6];                    % initial vector
theMax = A(1);                                % set initial max value
for x = A
    if x > theMax
        theMax = x;
    end
end
fprintf('max(A) is %d\n', theMax);
```

Below shows another example of finding the maximum value in a list of random numbers.

```
A = floor(rand(1, 10) * 100);
theMax = A(1);
theIndex = 1;
for index = 1:length(A)
    x = A(index);
    if x > theMax
        theMax = x;
        theIndex = index;
    end
end
fprintf('the max value in A is %d at %d\n', theMax, theIndex);
```

### 1.2.2 while statements

```
<initialization>
While <logical expression>
    % must make some changes to the values in logical expression
    % to terminate the loop eventually
    <code block>
end
```

The following is a rewritten version using *while* loop of the previous example.

```
A = floor(rand(1, 10) * 100);
theMax = A(1);
theIndex = 1;
index = 1;
while index <= length(A)
    x = A(index);
    if x > theMax
        theMax = x;
        theIndex = index;
    end
    index = index + 1;
end
fprintf('the max value in A is %d at %d\n', theMax, theIndex);
```

## 1.3 Save and Load variables

We can save variables to '.mat'file by 'save' and load them from file by 'load'.

```
A = [1,2,3];
B = [4,5,6];
save('data.mat', 'A', 'B'); % save to .mat file
load('data.mat');      % load from .mat file someday
```

# Part 2 Graph plotting in MATLAB

In this part, we will discuss the fundamental concepts of plotting in MATLAB. Plotting facilitates our examination of the distribution of information in multimedia contents.

## 2.1 Figure: Plot Container

Enter the followings in the command window

```
>> x = 1:10

>> y = 2 * x

>> plot(x, y)

>> figure


>> plot(x, 2 * y)

>> clf

>> close all
```

The commands *clf* and *close all* are used to clear the current figure and remove all the figures respectively. Try enter the followings in the command window:

The basic function *plot(x, y)* creates a simple plot of *x* versus *y*. There are some useful functions to enhance plots and these functions have to be called AFTER the plot figure is created.

*axis <param>* provides a rich set of tools for managing the appearance of the axes:

- *axis equal* sets the x and y scales to the same value
- *axis square* makes the plot figure of equal width and height
- *axis off* does not show the axes at all
- *axis on* recovers the axes

Enter the followings in the command window

```
>> x = 1:10

>> y = 2 * x

>> plot(x, y)
```

```
>> axis equal

>> axis square

>> axis off

>> axis on
```

*axis([x1, x2, y1, y2])* overrides the automatic computation of the axis values, forcing the x-axis to reach from x1 to x2 and the y-axis from y1 to y2.

Enter the following in the command window:

```
axis([-10, 10, 0, 50])
```

*grid on* puts a grid on the plot; *grid off* (default) removes grid lines.

Enter the following in the command window:

```
grid on

grid off
```

*hold on* holds the existing data on the figure to allow subsequent plotting calls to be added to the current figure without first erasing the existing plot. *hold off* (the default) redraws the current figure, erasing the previous contents.

Enter the followings in the command window:

```
hold on

plot(x, 3 * y)

hold off

plot(x, 4 * y)
```

*text(x, y, str)* places the text provided at the specified (x, y) location on a 2-D plot.

Enter the followings in Editor and run the script:

```
x = 1:10;

y = 2 * x;

plot(x, y);
```

```
for i = 1:10
    text(x(i) + 0.2, y(i) + 0.1, num2str(i));
end
```

*title(…)* places the text provided as the title of the plot.

*xlabel(…)* sets the string provided as the label for the x-axis.

*ylabel(…)* sets the string provided as the label for the y-axis.

Enter the followings in the command window:

```
title('My figure')
xlabel('x axis')
ylabel('y axis')
```

## 2.2 Subplots: Multiple Plots on a Single Figure

Within the same figure, you can place multiple plots with the subplot command.

The function *subplot(r, c, n)*

- divides the current figure into *r* rows and *c* columns of equally spaced plot areas;
- establishes the *n*-th of these as the current figure.

For example, if the command *subplot(2, 2, 1)* is used,

- the window is divided into two rows and two columns;
- the plot is drawn in the upper left-hand window;
- the windows are numbered from left to right, top to bottom.

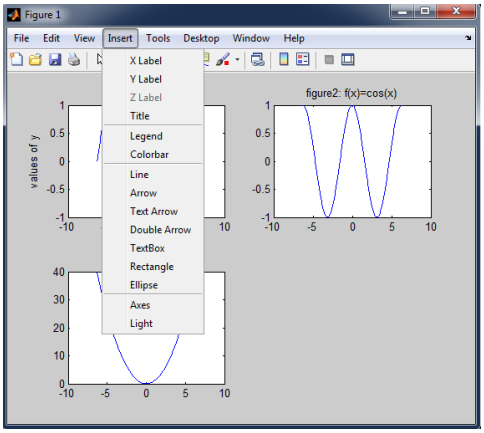
| | |
|---|---|
| n = 1 | n = 2 |
| n = 3 | n = 4 |

Examine the following script:

```
clf
x = -2*pi:0.05:2*pi;
```

```
subplot(2, 2, 1);

plot(x, sin(x));

title('figure1: f(x)=sin(x)');

xlabel('values of x');

ylabel('values of y');

subplot(2, 2, 2);

plot(x, cos(x));

title('figure2: f(x)=cos(x)');

subplot(2, 2, 3);

plot(x, x.^2);

title('figure3: f(x)=x^2');
```

When a figure has been created, you are able to manipulate many of its characteristics by using the menu items and tool bars. They provide the ability to resize the plot, change the view characteristics, and annotate it with axis labels, lines and text callouts.




## 2.3 Line, Color and Mark style

The basic function for plots is *plot(…)*. The usual way to use the function is to give it three parameters, *plot(x, y, str)*, where *x* and *y* are vectors of the same length containing the *x* and *y* coordinates respectively, and *str* is a string containing one or more optional line color and style control characters. If the vector *x* is omitted, MATLAB assumes that the x coordinates are *1:N*, where *N* is the length of the y vector. If the *str* is omitted, the default line and color style is solid blue. The function also permits multiple (*x*, *y*, *str*) data sets in a single function call.

Here are the Line, Color and Mark style:

| Line Type | Indicator | Point Type | Indicator | Color | Indicator |
|---|---|---|---|---|---|
| solid | - | point | . | blue | b |
| dotted | : | circle | o | green | g |
| dash-dot | -. | x-mark | x | red | r |
| dashed | -- | plus | + | cyan | c |
| | | star | * | magenta | m |
| | | square | s | yellow | y |
| | | diamond | d | black | k |

Enter the followings in the editor:

```
x = [0:2:18]
y = [0, 0.33, 4.13, 6.29, 6.85, 11.19, 13.19, 13.96, 16.33, 18.17]
plot(x ,y)
title('Lab Experiment 1')
xlabel('Time, sec')
ylabel('Distance, ft')
grid on
```

Enter the followings in the command window:

```
x = [0:2:18]
y = [0, 0.33, 4.13, 6.29, 6.85, 11.19, 13.19, 13.96, 16.33, 18.17]
plot(x, y, ':ok', x, y*2, '--xr', x, y/2, '-b')
```

# Part 3 Image Processing

## 3.1 Gray image

In Matlab, we can read an image file by imread, display an image by imshow, and write an image to file by imwrite:

```
grayLena = imread('lena.bmp'); % Read the image

imshow(grayLena); % Display the image

imwrite(grayLena, 'lena.png')
```

grayLena is an 512x512 2d-array storing all 8-bit gray-scale values of the image. The following should be displayed:


512x512 uint8

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 162 | 162 | 162 | 161 | 162 |
| 2 | 162 | 162 | 162 | 161 | 162 |
| 3 | 162 | 162 | 162 | 161 | 162 |
| 4 | 162 | 162 | 162 | 161 | 162 |
| 5 | 162 | 162 | 162 | 161 | 162 |
| 6 | 164 | 164 | 158 | 155 | 161 |
| 7 | 160 | 160 | 163 | 158 | 160 |
| 8 | 159 | 159 | 155 | 157 | 158 |
| 9 | 155 | 155 | 158 | 158 | 159 |
| 10 | 155 | 155 | 157 | 158 | 155 |

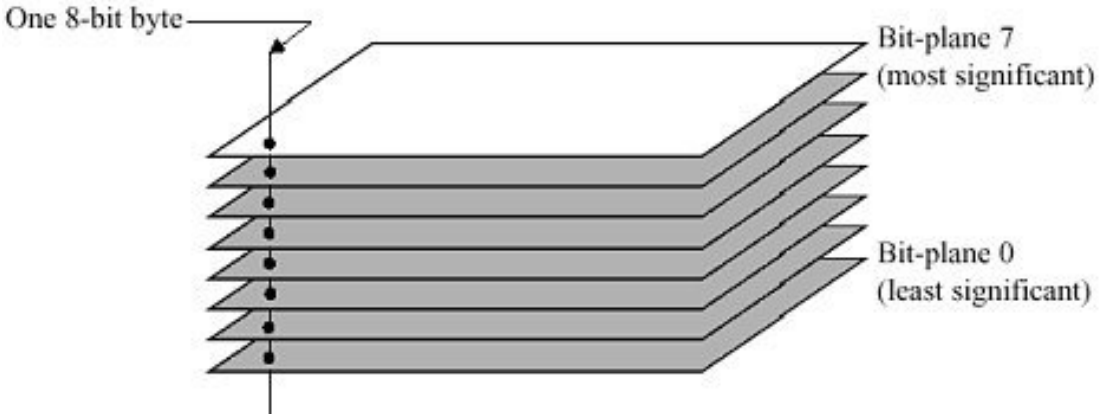
### 3.1.1 Complement image

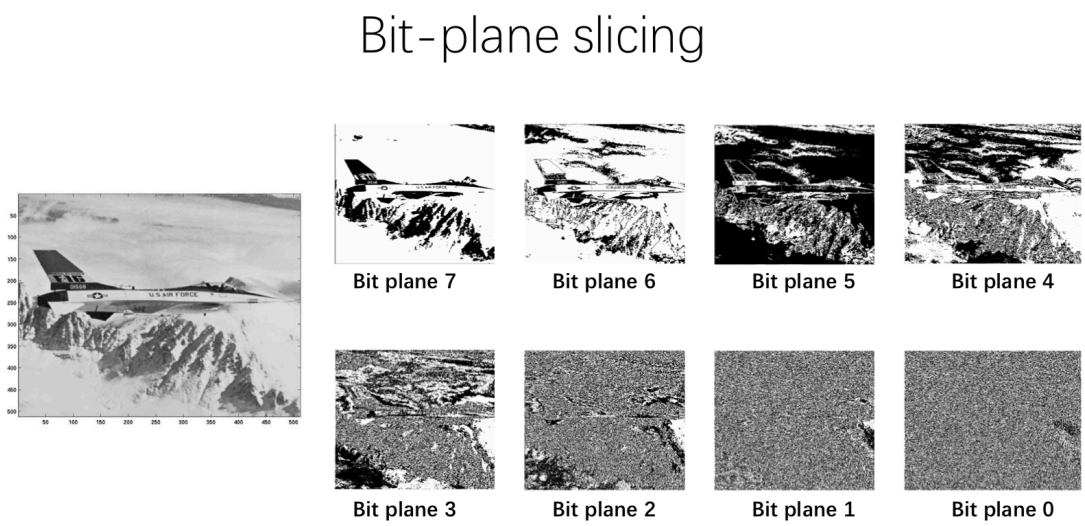
To complement the image in MATLAB, there are multiple solutions. Exam the following script.

```
subplot(1,3,1);
imshow(grayLena);


% Solution 1
invGrayLena1=255-grayLena;
imshow(invGrayLena1);


% Solution 2
invGrayLena2=imcomplement(grayLena);
imshow(invGrayLena2);
```

3.1.2 Bit plane

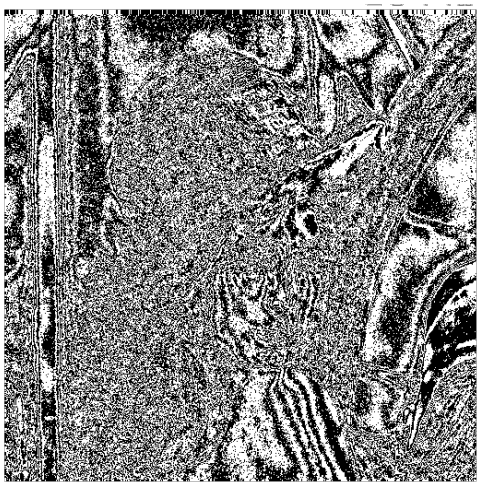
We can use bitget to get bit at specified position. Note that bit can be a scalar or an array of the same size as A. bit must be between 1 (the least-significant bit) and the number of bits in the integer class of A. Note that the position of bit in MATLAB starts at 1 instead of 0, so we have bit plane 1-8 instead of 0-7.




Here we extract bit plane 4 of gray Lena (MATLAB counts from 1, not from 0, so the eight bit-planes are numbered as 1-8, instead of 0-7):

```
plane4Lena=bitget(grayLena,4);

imshow(logical(plane4Lena));
```



## 3.2 Color image

A color image in computer is usually stored and processed as a series of matrix channel, simply we read in the image by using imread function. Create a new .m file to deal with our color image, type in the following in the editor.

```
colorLena=imread('colorLena.png');

imshow(colorLena);
```


By default, we will have a 512x512x3 3d-array that represent the red, green and blue channel, respectively. To check them out separately, exam the following script:

```
figure(2);

% extract the three channel(a 512*512 matrix)

redChannel = colorLena(:,:,1);

greenChannel = colorLena(:,:,2);

blueChannel = colorLena(:,:,3);
```


3.2.1 RGB to HSV

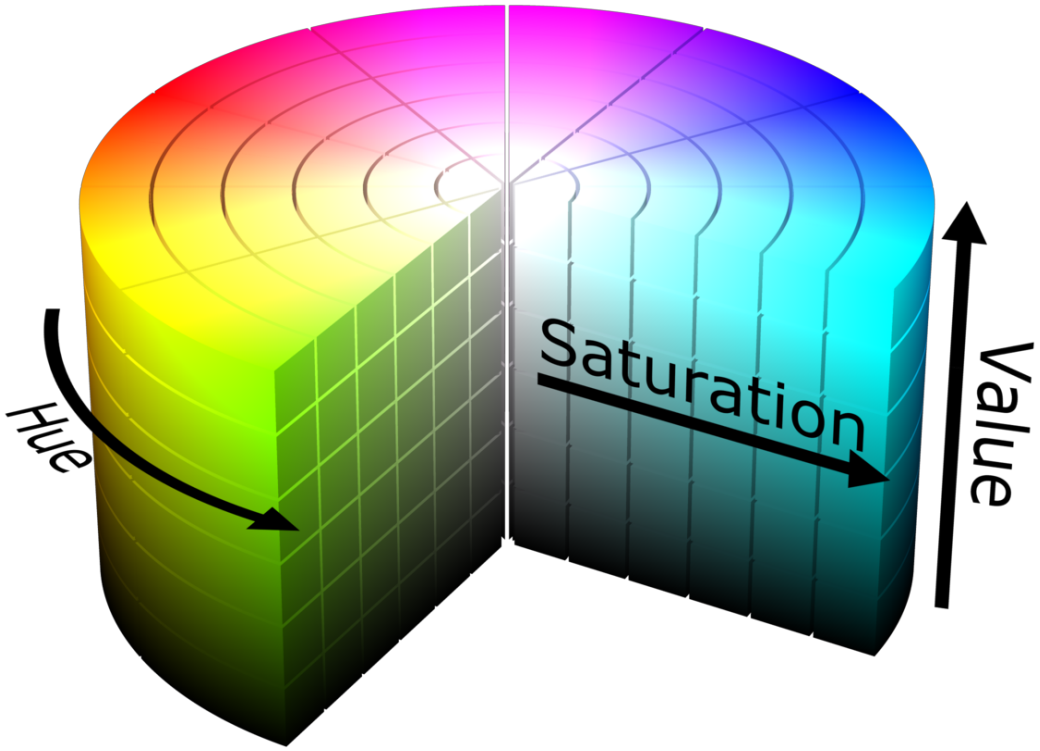
HSV is very commonly used in image processing. The three channels of HSV are Hue, Saturation, Value. Which stands for:

Hue: different kinds of color.

Saturation: how colorful or colorless it is.

Value: how light or dark it is.

Image below illustrates the HSV color space. In MATLAB, all three channels ranges from 0 to 1.




There are built-in function in MATLAB to convert a color image from RGB color model to HSV color model. Exam the following script in your editor.

```
hsvLena = rgb2hsv(colorLena);

hChannel = hsvLena(:,:,1);
```

```
sChannel = hsvLena(:,:,2);

vChannel = hsvLena(:,:,3);

figure(3);

subplot(1,3,1);

imshow(hChannel), title('H channel')

subplot(1,3,2);

imshow(sChannel), title('S channel')

subplot(1,3,3);

imshow(vChannel), title('V channel')
```


### 3.2.2 Image thresholding

Image thresholding is a simple, yet effective, way of partitioning an image into a foreground and background. This image analysis technique is a type of image segmentation that isolates objects by converting grayscale images into binary images. Image thresholding is most effective in images with high levels of contrast. Common image thresholding algorithms include histogram and multi-level thresholding. To conduct image thresholding, you can type in the following code in the editor.

```
figure(4);

threshold=200;

% if a pixel in redChannel is larger than the threshold

% it becomes true, otherwise it is false

BWR=redChannel>threshold;

imshowpair(redChannel,BWR,'montage');
```

### 3.2.3 Image segmentation

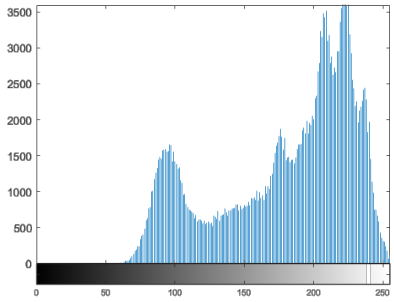
With a binary mask, we can then simply segment the region of the interest(ROI) from the image. To extract the image using the mask we generated above, exam the following script in the editor.

```
%% image segmentation

figure(5);

% convert BWR to unit8, the same data type as colorLena.
% dot multiply each channel with the mask
BWR = uint8(BWR);
segLena=colorLena;

segLena (:,:,1) = segLena (:,:,1).*BWR;

segLena (:,:,2) = segLena (:,:,2).*BWR;

segLena (:,:,3) = segLena (:,:,3).*BWR;

imshow(segLena);
```

### 3.2.4 Histogram

Histogram of a channel is a powerful tool to help you explore the feature of the image, to get the histogram of your image, exam the following script:

```
figure(5);

imhist(redChannel);
```




## 3.3 Image processing Toolbox

Play with image processing toolbox if you are interested!

https://www.mathworks.com/products/image.html

Image Processing Toolbox™ provides a comprehensive set of reference-standard algorithms and workflow apps for image processing, analysis, visualization, and algorithm development. You can perform image segmentation, image enhancement, noise reduction, geometric transformations, image registration, and 3D image processing.

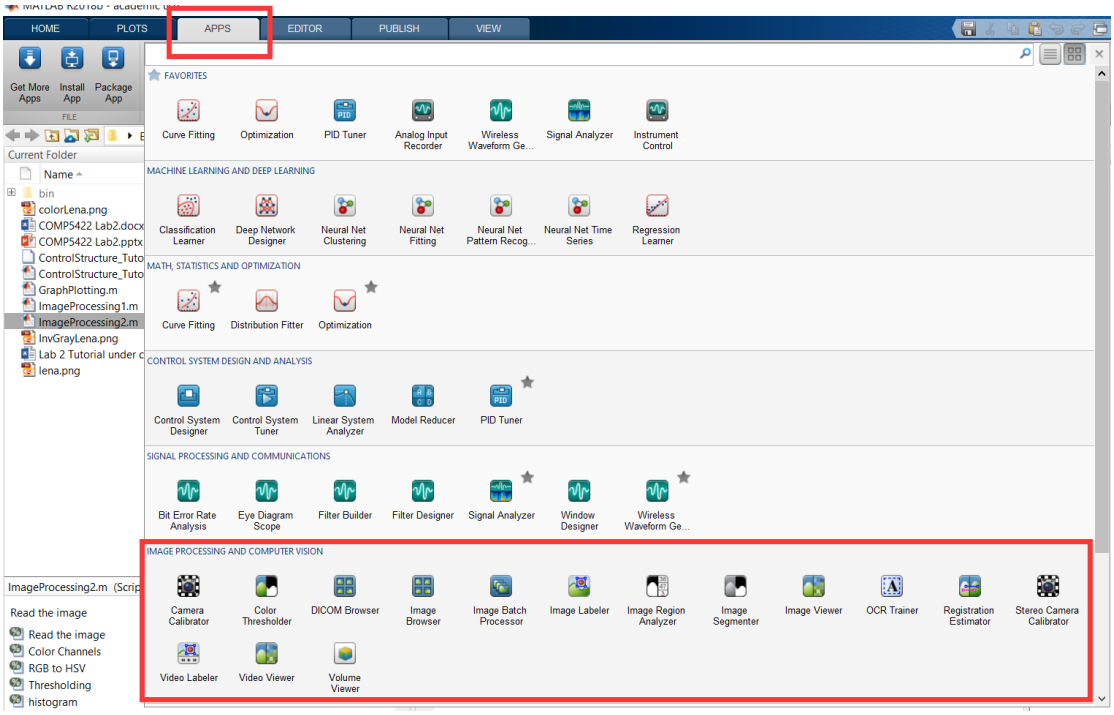



Image Processing Toolbox apps let you automate common image processing workflows. You can interactively segment image data, compare image registration techniques, and batch-process large datasets. Visualization functions and apps let you explore images, 3D volumes, and videos; adjust contrast; create histograms; and manipulate regions of interest (ROIs).

## Part 4 Assignment

Here is the task syllabus. Detailed instructions are in ‘task1.m’and ‘task2.m’, which are going to be filled by you.

- Task 1 Gray Image Processing
    1. Basic image read, write and display. (1 point)
    2. Bit-plane (1 point)
    3. Lossy Compression by Discarding Lower Bits. (0.5 point)
- Task 2 Color Image Processing
    1. Complement Image. (1 points)
    2. Modify image saturation. (1 points)
    3. Image Segmentation. (0.5 points)

Submit a zip file via blackboard, which includes:

1. A report (**Lab2_Your Student No.pdf**) with key code snippets. No length requirement, just clearly explain how you complete the above tasks.
2. The **image** files you produced.
3. **task1.m** and **task2.m**.

You could submit as many times as you want before **7:00 a.m. 12 Nov. (Monday Morning),** but only your **latest submission** will be graded.