Lab Objective and Outcome

Objective

The primary aim of this course is to provide essential and hands-on knowledge about computer vision components and to cultivate the necessary skills for image processing, applicable both in industry and research settings.

Key objectives of the course include:

1. Introducing fundamental terminology, technology, and its practical applications.
2. Exploring the concept of image processing and its essential operations.
3. Familiarizing students with Matlab, a versatile tool for image-related operations.
4. Introducing various datasets available on different platforms, widely used for analysis in the field of image processing.
5. Guiding students in the practical implementation of image processing techniques using both online and offline image datasets.

By the end of this course, students will be equipped with a solid foundation in computer vision, image processing techniques, and the practical tools required for success in this field.

Course Outcomes

After completion of this course, students will be able to –

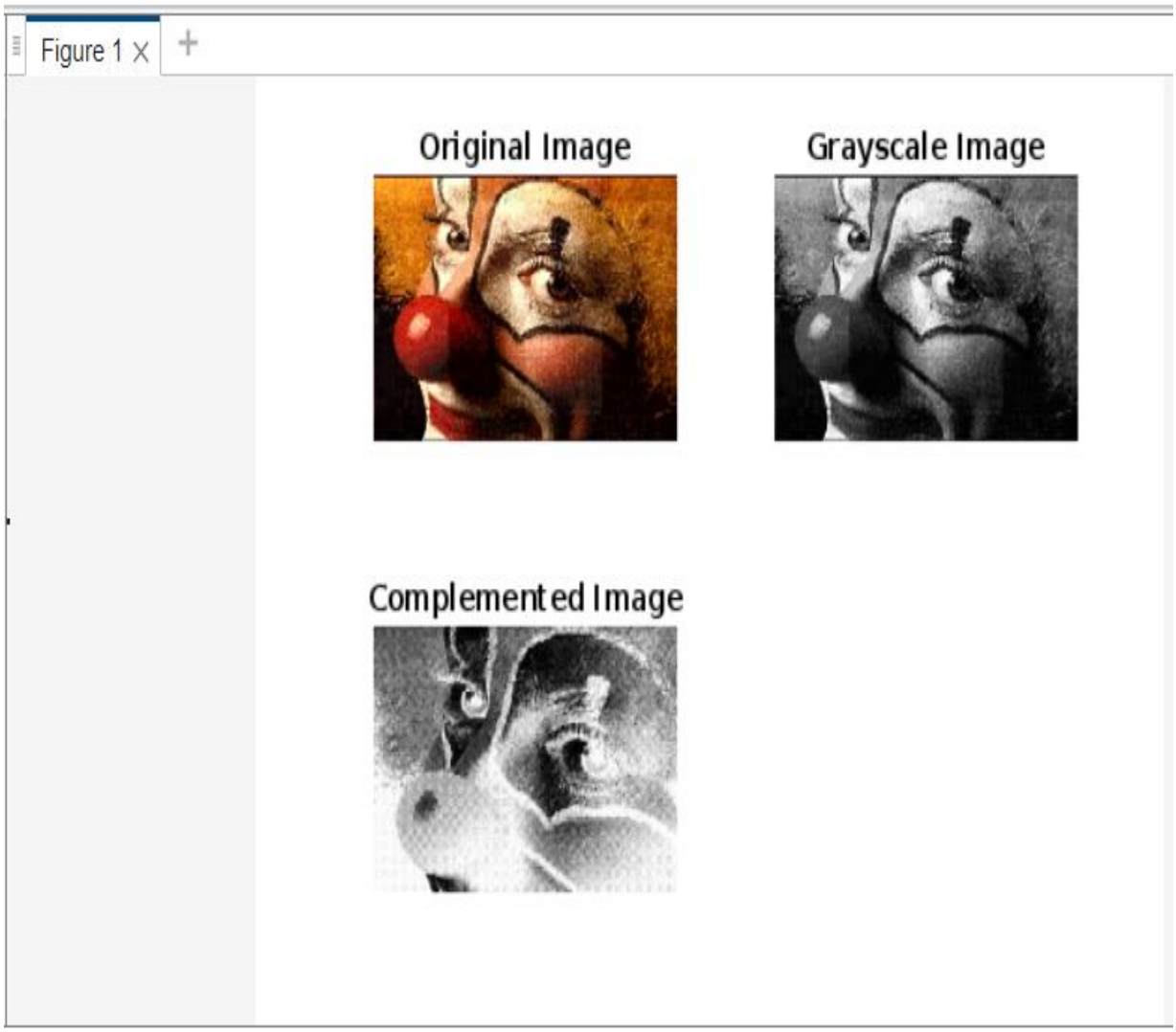
7CAI4-22.1	**Explain** the **concept** and Application of image processing for computer vision.
7CAI4-22.1	**Illustrate** key **technologies and implement** Histogram processing and Equalization for an image.
7CAI4-22.1	**Implement and Analyze the** Fourier transform for filtering the image.
7CAI4-22.1	**Application for performing and implementing** different methods of image segmentation
7CAI4-22.1	**Design** an Image processing application project using YOLO Model online datasets.

Experiment – 1

Aim: Reading images, writing image, conversion of images, and complementing of an image.

Matlab Code:

```
% Reading an image
originalImage = imread('example.jpg'); % Replace 'example.jpg' with your image file

% Display the original image
subplot(2, 2, 1);
imshow(originalImage);
title('Original Image');

% Writing the original image to a new file
imwrite(originalImage, 'original_image.jpg'); % Save as 'original_image.jpg'

% Convert the image to grayscale
grayImage = rgb2gray(originalImage);

% Display the grayscale image
subplot(2, 2, 2);
imshow(grayImage);
title('Grayscale Image');

% Writing the grayscale image to a new file
imwrite(grayImage, 'grayscale_image.jpg'); % Save as 'grayscale_image.jpg'

% Complementing (inverting) the grayscale image
complementedImage = imcomplement(grayImage);

% Display the complemented image
subplot(2, 2, 3);
imshow(complementedImage);
title('Complemented Image');

% Writing the complemented image to a new file
imwrite(complementedImage, 'complemented_image.jpg'); % Save as 'complemented_image.jpg'
```



In this code:

1. We read an image using **imread**.
2. We display the original image.
3. We write the original image to a new file using **imwrite**.
4. We convert the original image to grayscale using **rgb2gray**.
5. We display the grayscale image.
6. We write the grayscale image to a new file.
7. We complement (invert) the grayscale image using **imcomplement**.
8. We display the complemented image.
9. We write the complemented image to a new file.

Make sure to replace **'example.jpg'** with the path to your own image file. This code covers the basic operations of reading, writing, converting, and complementing images in MATLAB.

Experiment – 2

Aim: Implement contrast adjustment of an image. Implement Histogram processing and equalization.

Code

% Image Enhancement

```
I=imread('cancercell.jpg');
subplot(4,2,1); imshow(I); title('Original Image');

g=rgb2gray(I);
subplot(4,2,5); imshow(g); title('Gray Image');

J=imadjust(g,[0.3 0.7],[]);
subplot(4,2,3); imshow(J); title('Enhanced Image');

D= imadjust(I,[0.2 0.3 0; 0.6 0.7 1],[]);
subplot(4,2,4);imshow(D);title('Enhanced Image 2');
```

% Histogram and Histogram Equalization

```
subplot(4,2,7); imhist(g); title('Histogram of Gray Image');

m=histeq(g);
subplot(4,2,6); imshow(m); title('Equalized Image');
subplot(4,2,8); imhist(m); title('Histogram of Equalized
Image');
```

Original Image

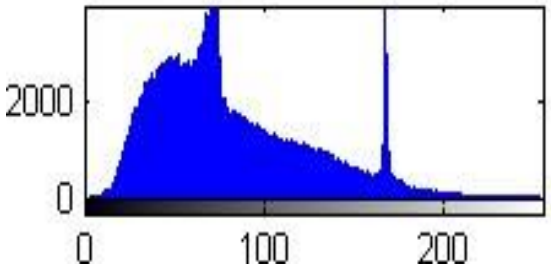
Enhanced Image

Enhanced Image 2

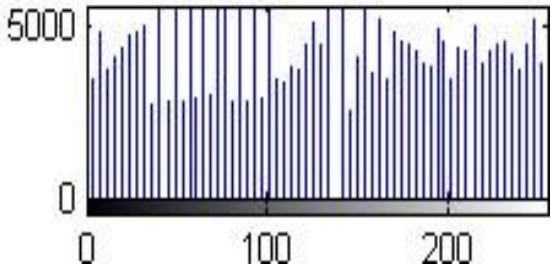
Gray Image

Equalized Image

Histogram of Gray Image



Histogram of Equalized Image



Experiment – 3

Aim: Use of Fourier transform for filtering the image.

Code:

```
% Read an image
originalImage = imread('example.jpg'); % Replace 'example.jpg' with your image file

% Convert the image to grayscale if it's in color
if size(originalImage, 3) == 3
   originalImage = rgb2gray(originalImage);
end

% Display the original image
subplot(2, 3, 1);
imshow(originalImage);
title('Original Image');

% Compute the 2D Fourier Transform of the image
fourierTransform = fft2(double(originalImage));

% Shift the zero frequency components to the center
fourierTransformShifted = fftshift(fourierTransform);

% Create a simple low-pass filter (e.g., a circular mask)
[M, N] = size(originalImage);
radius = 50; % Adjust the radius to control the filter size
[x, y] = meshgrid(1:N, 1:M);
centerX = N/2;
centerY = M/2;
mask = (x - centerX).^2 + (y - centerY).^2 <= radius^2;

% Apply the filter by multiplying it with the Fourier Transform
filteredTransform = fourierTransformShifted .* mask;

% Shift the frequency components back to their original positions
filteredTransformShifted = ifftshift(filteredTransform);
```

```
% Compute the inverse Fourier Transform to get the filtered image
filteredImage = ifft2(filteredTransformShifted);

% Display the filtered image
subplot(2, 3, 2);
imshow(abs(filteredImage), []);
title('Filtered Image');

% Display the magnitude of the Fourier Transform
subplot(2, 3, 3);
imshow(log(1 + abs(fourierTransformShifted)), []);
title('Magnitude of Fourier Transform');

% Display the filter mask
subplot(2, 3, 4);
imshow(mask, []);
title('Filter Mask');

% Display the phase of the Fourier Transform
subplot(2, 3, 5);
imshow(angle(fourierTransformShifted), []);
title('Phase of Fourier Transform');

% Display the phase of the filtered image
subplot(2, 3, 6);
imshow(angle(filteredImage), []);
title('Phase of Filtered Image');

% Wait for user input before closing the figure
pause;
close;
```

In the above code, we added two additional subplots to display the phase of both the Fourier Transform and the filtered image. This will help you visualize both the magnitude and phase changes resulting from the filtering process.

Original Image:

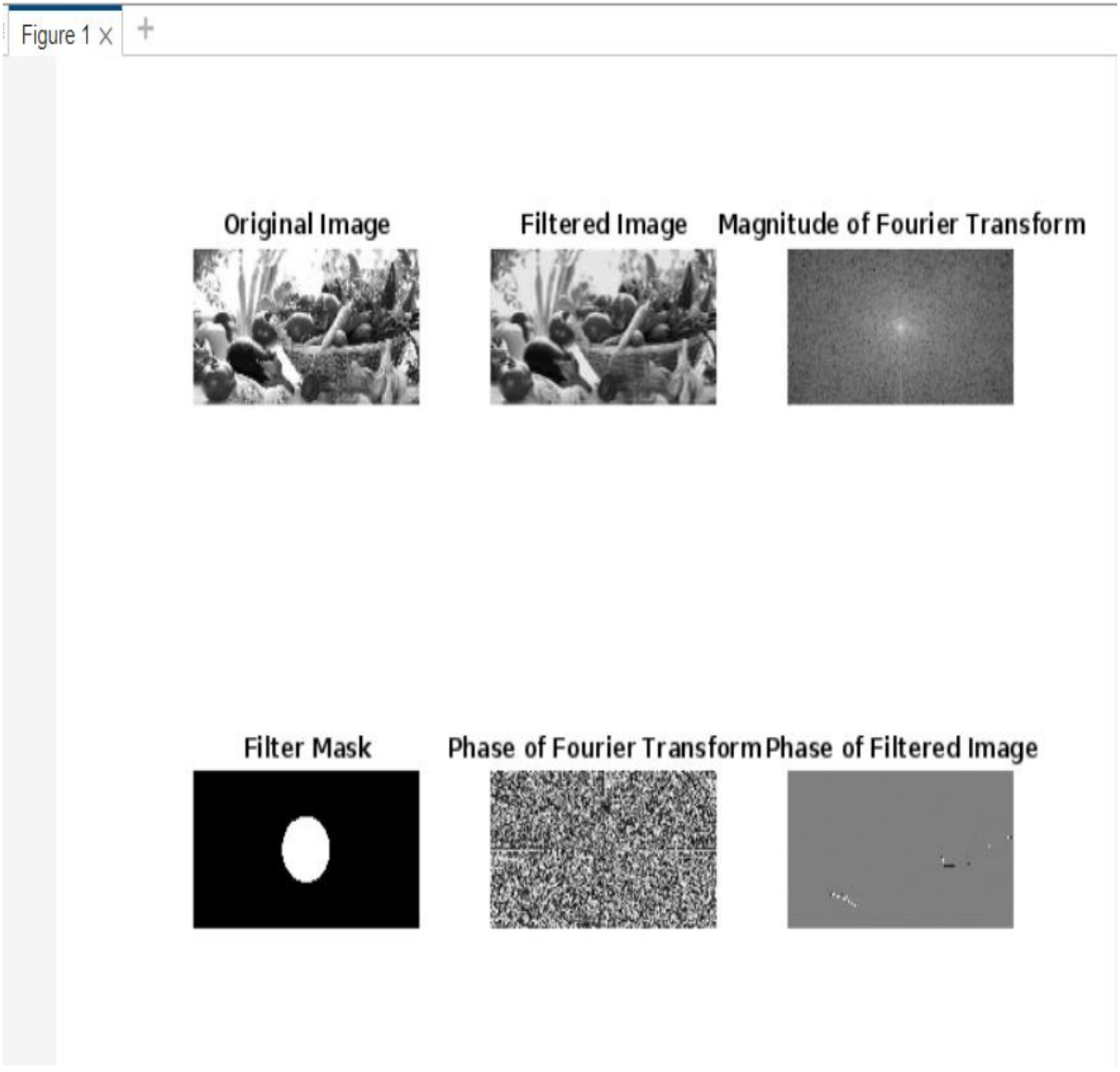


Figure 1
Original Image
Filtered Image
Magnitude of Fourier Transform
Filter Mask
Phase of Fourier Transform
Phase of Filtered Image

Experiment – 4

Aim: Utilization of SIFT and HOG features for image analysis.

MATLAB's Computer Vision Toolbox does not provide a built-in SIFT detector. Instead, you can use the VLFeat library for SIFT feature extraction. Here's an example of how to do it:
First, make sure you have the VLFeat library installed and added to MATLAB. You can download VLFeat from the official website (https://www.vlfeat.org/) and follow their installation instructions.
Once VLFeat is installed, you can use it for SIFT feature extraction as follows:

Code

```
% Load an image
img = imread('example.jpg'); % Replace 'example.jpg' with your image file

% Convert the image to grayscale if it's in color
if size(img, 3) == 3
    img = rgb2gray(img);
end

% Add the VLFeat toolbox to MATLAB (adjust the path as needed)
addpath('path_to_vlfeat/vlfeat-0.9.21/toolbox'); % Replace 'path_to_vlfeat' with the actual path

% Initialize VLFeat
run('vlfeat-0.9.21/toolbox/vl_setup');

% Extract SIFT features
```

```
[f, d] = vl_sift(single(img));

% Visualize SIFT features on the image
siftImage = insertMarker(img, f(1:2, :)', 'Color', 'red', 'Size', 5);

% Display the image with SIFT features
imshow(siftImage);
title('Image with SIFT Features');

% Create a HOG feature extractor
hogExtractor = vision.HOGDescriptor;

% Extract HOG features
hogFeatures = extractHOGFeatures(img);

% Display HOG features
figure;
subplot(1, 2, 1);
imshow(img);
title('Original Image');

subplot(1, 2, 2);
plot(hogFeatures);
title('HOG Features');

% Perform further analysis using SIFT and HOG features as needed
```

In this code:

- We use VLFeat to extract SIFT features.
- Make sure to replace '**path_to_vlfeat**' with the actual path to your VLFeat toolbox installation.
- We then visualize the SIFT features on the image and display them.
- The HOG feature extraction part remains the same as in the previous code.

Ensure that you have VLFeat properly installed and configured in MATLAB for this code to work.

Experiment – 5

Aim: Performing/Implementing image segmentation.

Implementing image segmentation in MATLAB typically involves using various segmentation algorithms, depending on the specific requirements of your task. One commonly used algorithm is k-means clustering for color-based segmentation. Here's an example of how to perform image segmentation using k-means clustering:

Matlab Code

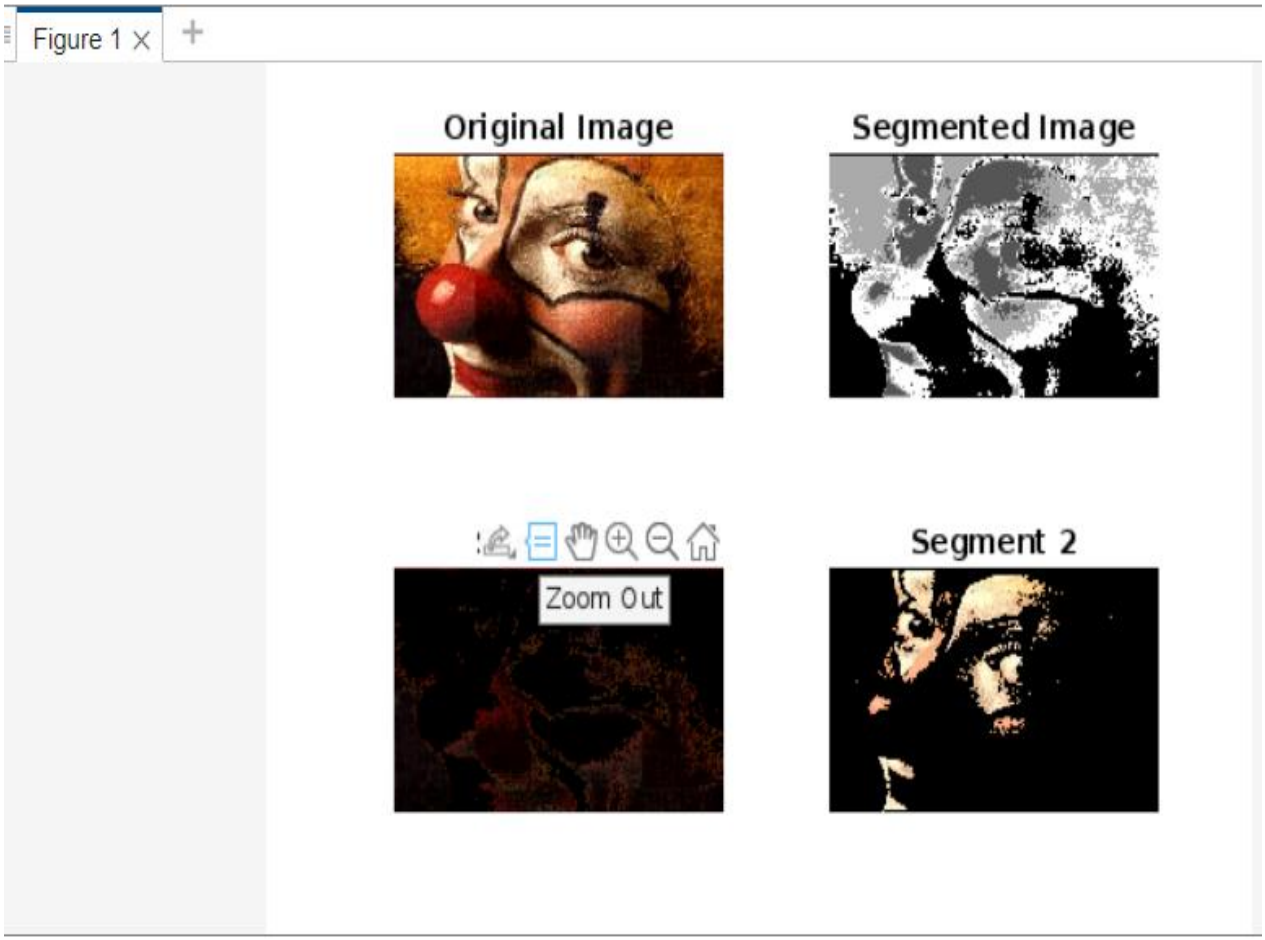
```
% Read an image
originalImage = imread('example.jpg'); % Replace 'example.jpg' with your image file

% Display the original image
subplot(2, 2, 1);
imshow(originalImage);
title('Original Image');

% Convert the image to double precision for k-means
imageData = double(originalImage);

% Reshape the image data into a 2D matrix
[m, n, ~] = size(imageData);
imageData = reshape(imageData, m * n, 3);

% Perform k-means clustering (adjust the number of clusters as needed)
numClusters = 4; % Change the number of clusters as per your requirement
[clusterIndices, clusterCenters] = kmeans(imageData, numClusters);

% Reshape the clusterIndices back to the original image size
segmented = reshape(clusterIndices, m, n);

% Display the segmented image
subplot(2, 2, 2);
imshow(segmented, []);
title('Segmented Image');

% Create a mask for each segment
segmentMasks = cell(numClusters, 1);
for i = 1:numClusters
    mask = (segmented == i);
```

```
    segmentMasks{i} = mask;
end

% Display each segmented region
for i = 1:numClusters
    subplot(2, 2, i + 2);
    segmentedRegion = originalImage;
    segmentedRegion(repmat(~segmentMasks{i}, [1, 1, 3])) = 0;
    imshow(segmentedRegion);
    title(['Segment ', num2str(i)]);
end
```

In this code:

1. We read an image and display it.
2. We convert the image to double precision for k-means clustering.
3. We reshape the image data into a 2D matrix to prepare it for clustering.
4. We perform k-means clustering with a specified number of clusters.
5. We reshape the cluster indices back to the original image size.
6. We display the segmented image.
7. We create masks for each segment.
8. We display each segmented region individually.

You can adjust the **numClusters** variable to control the number of segments in the output. Feel free to experiment with different segmentation algorithms and parameters based on your specific image segmentation needs.



Experiment – 6

Aim: Object detection and Recognition on available online image datasets using the YOLO Model.

Implementing object detection and recognition using the YOLO (You Only Look Once) model in MATLAB can be complex, as it often requires pre-trained models and a deep learning framework like TensorFlow or PyTorch. However, I can provide you with a general outline of the steps involved in performing object detection and recognition using a pre-trained YOLO model. You may need to adapt this code to your specific dataset and requirements.

1. Setup Deep Learning Framework: Ensure you have a deep learning framework like TensorFlow or PyTorch installed in MATLAB.

2. Download Pre-trained YOLO Model: Download a pre-trained YOLO model, such as YOLOv3 or YOLOv4, along with its configuration file and weights. You can find pre-trained YOLO models on the official YOLO website or other sources.
3. Load and Configure the Model: Load the pre-trained YOLO model in MATLAB using the deep learning framework. Configure the model according to the model's architecture and requirements.
4. Load and Preprocess the Image Dataset: Load the image dataset you want to perform object detection and recognition on. Preprocess the images to match the input format expected by the YOLO model (e.g., resizing, normalization).
5. Perform Object Detection: Use the pre-trained YOLO model to perform object detection on the images in the dataset. This will generate bounding boxes and class predictions for detected objects.
6. Display Detected Objects: Visualize the detected objects by drawing bounding boxes and labels on the images.

Here's a high-level example of how you might perform these steps using MATLAB and TensorFlow for object detection and recognition with a pre-trained YOLO model (assuming you have the model files and dataset ready):

Matlab Code:

```
% Load the pre-trained YOLO model and configure it (using TensorFlow)
net = importKerasNetwork('yolov3.h5'); % Replace with the path to your model
inputSize = [416 416 3]; % Input size expected by the model

% Load and preprocess the image dataset
imageFolder = 'path_to_dataset_folder'; % Replace with your dataset folder
imageFiles = dir(fullfile(imageFolder, '*.jpg')); % Assuming JPEG images
numImages = length(imageFiles);

for i = 1:numImages
    % Load and preprocess each image
    img = imread(fullfile(imageFolder, imageFiles(i).name));
    img = imresize(img, inputSize(1:2));
    img = img/255; % Normalize pixel values (if needed)

    % Perform object detection
    detections = detectYOLO(net, img);

    % Visualize and process detections (e.g., filter by confidence score)
```

```
    % Display detected objects with bounding boxes and labels
    imshow(img);
    hold on;
    for j = 1:numel(detections)
        bbox = detections(j).Location;
        label = detections(j).Label;
        score = detections(j).Confidence;
        if score > 0.5 % Adjust confidence threshold as needed
            rectangle('Position', bbox, 'EdgeColor', 'r', 'LineWidth', 2);
            text(bbox(1), bbox(2) - 10, label, 'Color', 'r', 'FontSize', 12);
        end
    end
    hold off;

    % Further processing or recognition for detected objects can be done here
end
```

Note: Please note that this is a simplified example, and you'll need to adapt it to your specific dataset and requirements. Additionally, the actual code may vary based on the pre-trained YOLO model you choose and the deep learning framework you're using in MATLAB.

Beyond Syllabus

Experiment – 7

Aim: Implementation of image restoring techniques

Image restoration is a broad field with various techniques, and the specific implementation depends on the method you want to use. Let's implement image restoration using MATLAB with an example of image denoising using the Total Variation (TV) regularization technique. In this example, we'll use a simple test image and add noise to it before restoring it.

Matlab Code:

```
% Load a sample image
originalImage = imread('cameraman.tif');  % Load a sample image

% Add noise to the image (you can adjust the noise level as needed)
noiseLevel = 25;
noisyImage = imnoise(originalImage, 'gaussian', 0, (noiseLevel/255)^2);

% Display the noisy image
subplot(1, 2, 1);
imshow(noisyImage);
title('Noisy Image');

% Image restoration using Total Variation regularization
lambda = 0.1;  % Regularization parameter (adjust as needed)
numIterations = 100;  % Number of iterations (adjust as needed)

% Perform TV regularization-based image restoration
restoredImage = tvdenoise(noisyImage, lambda, numIterations);

% Display the restored image
subplot(1, 2, 2);
imshow(restoredImage);
title('Restored Image');
```

```
% Define the TV denoising function (you can use an existing MATLAB implementation)
function denoisedImage = tvdenoise(inputImage, lambda, numIterations)
    u = inputImage;
    [Dux, Duy] = gradient(u);

    for k = 1:numIterations
        u_old = u;

        % Update u using TV denoising
        u = u - lambda * divergence(Dux, Duy);

        % Compute gradient
        [Dux, Duy] = gradient(u);

        % Projection step
        normDu = sqrt(Dux.^2 + Duy.^2);
        normDu(normDu == 0) = 1;
        Dux = Dux ./ normDu;
        Duy = Duy ./ normDu;

        % Ensure the solution is within the range [0, 255]
        u = min(max(u, 0), 255);

        % Check for convergence
        if norm(u - u_old, 'fro') / norm(u_old, 'fro') < 1e-6
            break;
        end
    end

    denoisedImage = u;
end
```

In this code:

1. We load a sample grayscale image (you can replace it with your own image).
2. We add Gaussian noise to the image, with the noise level adjustable by the noiseLevel variable.
3. We perform image restoration using Total Variation (TV) regularization, a popular technique for denoising.

4. The tvdenoise function implements the TV denoising algorithm. It iteratively updates the image to reduce noise while preserving edges.

Please note that this is a simplified example. For practical applications, you may need to fine-tune parameters, choose different restoration techniques, or use more advanced denoising algorithms depending on your specific needs and the characteristics **of your images.**

Experiment – 8

Aim: Implementation of Image Intensity slicing technique for image enhancement

Image intensity slicing is a simple technique used for enhancing specific regions of interest in an image by applying a custom intensity mapping function. Here's a MATLAB code example to implement image intensity slicing:

Matlab Code

```
% Load the image
originalImage = imread('sample_image.jpg');  % Load your image here

% Define the intensity range for enhancement
minIntensity = 100;  % Minimum intensity for enhancement
maxIntensity = 200;  % Maximum intensity for enhancement

% Create a copy of the original image for enhancement
enhancedImage = originalImage;

% Apply intensity slicing
enhancedImage(originalImage >= minIntensity & originalImage <= maxIntensity) = 255;

% Display the original and enhanced images side by side
subplot(1, 2, 1);
imshow(originalImage);
title('Original Image');

subplot(1, 2, 2);
imshow(enhancedImage);
title('Enhanced Image');
```

```
% Save the enhanced image if needed
imwrite(enhancedImage, 'enhanced_image.jpg');  % Save the enhanced image
```

In this code:

1. Load your input image (replace '**sample_image.jpg**' with the path to your image).
2. Define the minimum (**minIntensity**) and maximum (**maxIntensity**) intensity values for enhancement. Any pixel values within this range will be set to 255 (white) in the enhanced image.
3. Create a copy of the original image to preserve the original data.
4. Apply the intensity slicing technique by setting pixel values within the specified intensity range to 255 (white).
5. Display both the original and enhanced images side by side using subplots.
6. Optionally, save the enhanced image to a file using **imwrite** if needed.

This simple intensity slicing technique enhances the specified intensity range while keeping other parts of the image unchanged. You can adjust the **minIntensity** and **maxIntensity** values to target the specific intensity range you want to enhance.

Experiment – 9

Aim: Canny edge detection Algorithm

MATLAB code example for implementing the Canny edge detection algorithm with a sample image:

```
% Load the sample image
originalImage = imread('sample_image.jpg');  % Load your image here

% Convert the image to grayscale if it's not already
if size(originalImage, 3) == 3
   grayImage = rgb2gray(originalImage);
else
   grayImage = originalImage;
end

% Apply Gaussian smoothing to reduce noise and enhance edges
sigma = 1.5;  % Adjust the Gaussian filter's standard deviation as needed
```

```
smoothedImage = imgaussfilt(grayImage, sigma);

% Perform Canny edge detection
lowThreshold = 0.1;  % Adjust the low threshold as needed (suggested range: 0.1-0.4)
highThreshold = 0.3;  % Adjust the high threshold as needed (suggested range: 0.3-0.7)
edgeImage = edge(smoothedImage, 'Canny', [lowThreshold highThreshold]);

% Display the original and edge-detected images side by side
subplot(1, 2, 1);
imshow(grayImage);
title('Original Grayscale Image');

subplot(1, 2, 2);
imshow(edgeImage);
title('Canny Edge Detection');

% Save the edge-detected image if needed
imwrite(edgeImage, 'canny_edges.jpg');  % Save the edge-detected image
```

In this code:

1. Load your input image (replace **'sample_image.jpg'** with the path to your image).
2. If the input image is in color, it is converted to grayscale using **rgb2gray** because Canny edge detection is typically performed on grayscale images.
3. Apply Gaussian smoothing to the grayscale image to reduce noise and enhance edges. You can adjust the **sigma** parameter to control the degree of smoothing.
4. Perform Canny edge detection using the **edge** function. Adjust the **lowThreshold** and **highThreshold** values to control the sensitivity of edge detection. A larger range between the thresholds results in fewer detected edges.
5. Display both the original grayscale image and the Canny edge-detected image side by side using subplots.
6. Optionally, save the edge-detected image to a file using **imwrite** if needed.

You can experiment with different threshold values and the standard deviation of the Gaussian filter to fine-tune the edge detection results for your specific images.