MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

LAB MANUAL

COMPUTER VISION LAB [CSE 3181]

Fifth Semester BTech in CSE(AI&ML)

(JULY – DEC 2023)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL-576104

MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

CERTIFICATE

This is to certify that Ms./Mr. .....

Reg. No.: ..... Section: ..... Roll No.: .....

has satisfactorily completed the **LAB EXERCISES PRESCRIBED FOR COMPUTER VISION LAB (CSE 3181)** of Third Year B.Tech. degree in Computer Science and Engineering (AI & ML) at MIT, Manipal, in the Academic Year 2023– 2024.

Date: .....

Signature
Faculty in Charge

CONTENTS

Course Objectives

- Infer the concepts of image formation, colour models and linear filtering.
- Outline the mathematics behind feature detection and description methods.
- Demonstrate the fundamental concepts in camera calibration.
- Compare various object tracking algorithms.
- Build object and scene recognition and categorization from images.

Course Outcomes

At the end of this course, students will be able to

- Write OpenCV programs in python for computer vision tasks.
- Use various segmentation and feature extraction methods for high level tasks.
- Use different object tracking techniques for computer vision tasks.

Evaluation plan

- Internal Assessment Marks: 60
- Continuous Evaluation: 60
    - Continuous evaluation component (for each evaluation): 10 marks
    - The assessment will depend on punctuality, program execution, maintaining the observation note and answering the questions in viva voce.
- End semester assessment based on miniproject: 40 [20% Report + 20% Demonstration]

INSTRUCTIONS TO THE STUDENTS

Pre- Lab Session Instructions

1. Students should carry the Lab Manual Book and the required stationery to every lab session.
2. Be in time and follow the institution dress code.
3. Must Sign in the log register provided.
4. Make sure to occupy the allotted seat and answer the attendance
5. Adhere to the rules and maintain the decorum.
6. Students must come prepared for the lab in advance.

In- Lab Session Instructions

- Follow the instructions on the allotted exercises.
- Show the program and results to the instructors on completion of experiments.
- On receiving approval from the instructor, copy the program and results in the Lab record.
- Prescribed textbooks and class notes can be kept ready for reference if required.

General Instructions for the exercise in Lab

- Implement the given exercise individually and not in a group.
- Observation book should be complete with program, proper input output clearly showing the parallel execution in each process. Plagiarism (copying from others) is strictly prohibited and would invite severe penalty in evaluation.
- The exercises for each week are divided under three sets:
- Solved example
- Lab exercises - to be completed during lab hours
- Additional Exercises - to be completed outside the lab or in the lab to enhance the

- In case a student misses a lab class, he/ she must ensure that the experiment is completed during the repetition class with the permission of the faculty concerned but credit will be given only to one day's experiment(s).
- Questions for lab tests and examination are not necessarily limited to the questions in the manual, but may involve some variations and / or combinations of the questions.

THE STUDENTS SHOULD NOT

- Bring mobile phones or any other electronic gadgets to the lab.
- Go out of the lab without permission.

Lab No 1: **Date:**

Introduction to OpenCV using Python

Objectives:

In this lab, student will be able to

1. Understand the execution environment of PyCharm
2. Learn basic concepts of OpenCV with Python
3. Learn and use the basic functions of OpenCV

I. Introduction to OpenCV

OpenCV is an open-source software library for computer vision and machine learning. The OpenCV full form is Open-Source Computer Vision Library. It was created to provide a shared infrastructure for applications for computer vision and to speed up the use of machine perception in consumer products. OpenCV, as a BSD-licensed software, makes it simple for companies to use and change the code. There are some predefined packages and libraries that make our life simple and OpenCV is one of them.

The term Computer Vision (CV) is used and heard very often in artificial intelligence (AI) and deep learning (DL) applications. The term essentially means giving a computer the ability to see the world as we humans do.

Computer Vision is a field of study which enables computers to replicate the human visual system. As already mentioned above, it's a subset of artificial intelligence which collects information from digital images or videos and processes them to define the attributes. The entire process involves image acquiring, screening, analysing, identifying and extracting information. This extensive processing helps computers to understand any visual content and act on it accordingly.

II. OpenCV Installation

To install OpenCV, one must have Python and PIP, preinstalled on their system. To check if your system already contains Python, go through the following instructions: Open the Command line(search for cmd in the Run dialog(+ R). Now run the following command:

python --version

If Python is already installed, it will generate a message with the Python version available.

PIP is a package management system used to install and manage software packages/libraries written in Python. These files are stored in a large "on-line repository" termed as Python Package Index (PyPI). To check if PIP is already installed on your system, just go to the command line and execute the following command:

pip -v

The PIP can be downloaded and installed using the command line by going through the following steps:

Method: Using cURL in Python

Curl is a UNIX command that is used to send the PUT, GET, and POST requests to a URL. This tool is utilized for downloading files, testing REST APIs, etc.

Step 1: Open the cmd terminal

Step 2: In python, a curl is a tool for transferring data requests to and from a server. Use the following command to request:

curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py

python get-pip.py

Verification of the installation process

One can easily verify if the pip has been installed correctly by performing a version check on the same. Just go to the command line and execute the following command:

pip -v or pip --version

III. Simple Image and video reading/writing program:

Reading, displaying, and writing images are basic to image processing and computer vision. Even when cropping, resizing, rotating, or applying different filters to process images, you'll need to first read in the images. So it's important that you master these basic operations. Use the following link to read more information.

https://learnopencv.com/read-display-and-write-an-image-using-opencv/

Solved Exercise:

```
# import the cv2 library
import cv2

# The function cv2.imread() is used to read an image.
img_grayscale = cv2.imread('D:\CLASS MATERIAL\MY CODE\data\kid.jpg', 0)

# The function cv2.imshow() is used to display an image in a window.
cv2.imshow('graycsale image', img_grayscale)

# waitKey() waits for a key press to close the window and 0 specifies indefinite loop
cv2.waitKey(0)

# cv2.destroyAllWindows() simply destroys all the windows we created.
cv2.destroyAllWindows()

# The function cv2.imwrite() is used to write an image.
cv2.imwrite('d:\grayscale.jpg', img_grayscale)
```

```
import numpy as np
import cv2

cap = cv2.VideoCapture(0)
fourcc = cv2.VideoWriter_fourcc(*'XVID')
out = cv2.VideoWriter('d:\output.avi',fourcc, 20.0, (640,480))

while(True):
    ret, frame = cap.read()
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    out.write(frame)
    cv2.imshow('frame',gray)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

cap.release()
out.release()
cv2.destroyAllWindows()
```

Lab Exercises:

1. Write a simple program to read, display, and write an image.
2. Write a simple program to read and display a video file.
3. Write a simple program to Extracting the RGB values of a pixel.
4. Write a simple program to draw rectangle.
5. Write a simple program to Resizing the Image.
6. Write a simple program to Rotating the Image.

Lab No 2: **Date:**

Implementation of Image Enhancement Methods

Objectives:

In this lab, student will be able to

1. Understand the implement various spatial image enhancement methods
2. Plot histograms and use the concept for image enhacement

I. Image enhancement

Image enhancement techniques aim to improve the visual quality of an image by increasing the contrast, sharpening edges, adjusting brightness and colour balance, or reducing blurriness. The objective is to make the image visually appealing and improve its interpretability for human observers. Intensity transformations are applied on images for contrast manipulation or image thresholding. These are in the spatial domain, i.e. they are performed directly on the pixels of the image at hand, as opposed to being performed on the Fourier transform of the image.

The following are commonly used intensity transformations:

- Image Negatives (Linear)

- Log Transformations

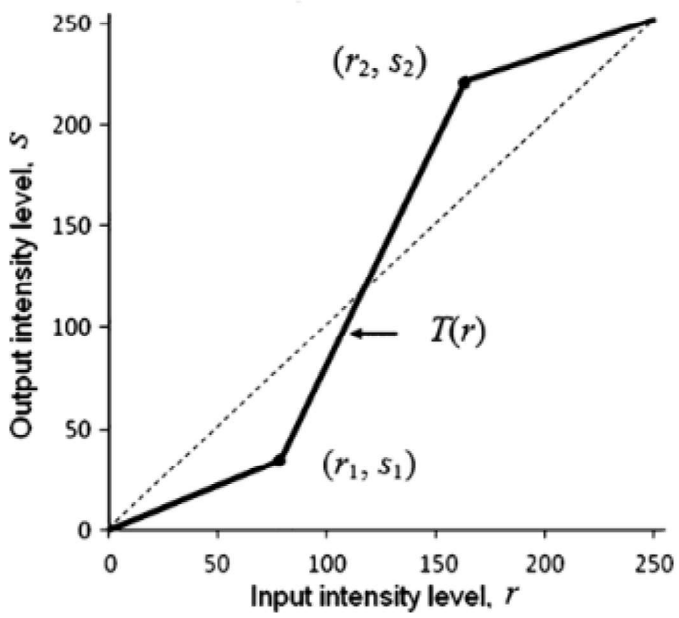
```
import cv2
import numpy as np

# Open the image.
img = cv2.imread('sample.jpg')

# Apply log transform.
c = 255/(np.log(1 + np.max(img)))
log_transformed = c * np.log(1 + img)

# Specify the data type.
log_transformed = np.array(log_transformed, dtype = np.uint8)

# Save the output.
cv2.imwrite('log_transformed.jpg', log_transformed)
```

- Power-Law (Gamma) Transformations

```
import cv2
import numpy as np

# Open the image.
img = cv2.imread('sample.jpg')

# Trying 4 gamma values.
for gamma in [0.1, 0.5, 1.2, 2.2]:

    # Apply gamma correction.
    gamma_corrected = np.array(255*(img / 255) ** gamma, dtype = 'uint8')

    # Save edited images.
    cv2.imwrite('gamma_transformed'+str(gamma)+'.jpg', gamma_corrected)
```

- Piecewise-Linear Transformation Functions

Contrast = (I_max - I_min)/(I_max + I_min)



```
import cv2
import numpy as np

# Function to map each intensity level to output intensity level.
def pixelVal(pix, r1, s1, r2, s2):
    if (0 <= pix and pix <= r1):
        return (s1 / r1)*pix
    elif (r1 < pix and pix <= r2):
        return ((s2 - s1)/(r2 - r1)) * (pix - r1) + s1
    else:
        return ((255 - s2)/(255 - r2)) * (pix - r2) + s2

# Open the image.
img = cv2.imread('sample.jpg')

# Define parameters.
r1 = 70
s1 = 0
r2 = 140
s2 = 255

# Vectorize the function to apply it to each value in the Numpy array.
pixelVal_vec = np.vectorize(pixelVal)

# Apply contrast stretching.
contrast_stretched = pixelVal_vec(img, r1, s1, r2, s2)

# Save edited image.
cv2.imwrite('contrast_stretch.jpg',          stretched)
```

Lab Exercises:

1. Write a program to read an image and perform histogram equalization.
2. Write a program to read an input image, reference image, and perform histogram specification.
3. Write a program to Resizing the Image and cropping an image.
   (https://learnopencv.com/image-resizing-with-opencv/)

Lab No 3: **Date:**

Implementation of Image Filtering Operations

Objectives:

In this lab, student will be able to

1. Implement linear and nonlinear filtering operations
2. Understand the importance of filters in preprocessing and feature detection

I. Image filtering

Image filtering operations are commonly used techniques in digital image processing to enhance or modify images. They involve applying a mathematical operation to each pixel or a neighbourhood of pixels in an image.

Here are some popular image filtering operations:

Gaussian Blur: It is a smoothing filter that reduces noise and details in an image. It applies a Gaussian function to each pixel, averaging its neighbouring pixels.

Median Filter: This filter replaces each pixel with the median value of the neighbouring pixels. It is effective in removing salt-and-pepper noise while preserving edges.

Sobel Operator: It is an edge detection filter that highlights edges in an image. The filter calculates the gradient magnitude at each pixel, emphasizing areas of rapid intensity change.

Laplacian Filter: This filter enhances the high-frequency components of an image, highlighting edges and details. It calculates the second derivative of the image intensity at each pixel.

Unsharp Masking: It is a sharpening filter that enhances edges and details in an image. It subtracts a blurred version of the image from the original, emphasizing high-frequency components.

Solved Exercise:

```python
# importing libraries
import cv2
import numpy as np

image = cv2.imread('                                    fruits.jpg')

cv2.imshow('Original Image', image)
cv2.waitKey(0)

# Gaussian Blur
Gaussian = cv2.GaussianBlur(image, (7, 7), 0)
cv2.imshow('Gaussian Blurring', Gaussian)
cv2.waitKey(0)

# Median Blur
median = cv2.medianBlur(image, 5)
cv2.imshow('Median Blurring', median)
cv2.waitKey(0)


# Bilateral Blur
bilateral = cv2.bilateralFilter(image, 9, 75, 75)
cv2.imshow('Bilateral Blurring', bilateral)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

```python
import cv2
import numpy as np

image = cv2.imread('test.jpg')

# Print error message if image is null
if image is None:
    print('Could not read image')

# Apply identity kernel
kernel1 = np.array([[0, 0, 0],
                    [0, 1, 0],
                    [0, 0, 0]])

identity = cv2.filter2D(src=image, ddepth=-1, kernel=kernel1)

cv2.imshow('Original', image)
cv2.imshow('Identity', identity)

cv2.waitKey()
cv2.imwrite('identity.jpg', identity)
cv2.destroyAllWindows()

# Apply blurring kernel
kernel2 = np.ones((5, 5), np.float32) / 25
img = cv2.filter2D(src=image, ddepth=-1, kernel=kernel2)

cv2.imshow('Original', image)
cv2.imshow('Kernel Blur', img)

cv2.waitKey()
cv2.imwrite('blur_kernel.jpg', img)
cv2.destroyAllWindows()
```

Lab Exercises:

1. Write a program to read an image and perform unsharp masking.
2. Write a program to obtain gradient of an image.
3. Write a program to compare box filter and gaussian filter image outputs.
4. Write a program to detect edges in a image.
5. Implement Canny edge detection algorithm.

Lab No 4: **Date:**

Implementation of Image Segmentation Methods

Objectives:

In this lab, student will be able to

1. Implement linear and nonlinear filtering operations
2. Understand the importance of filters in preprocessing and feature detection

I. Image segmentation

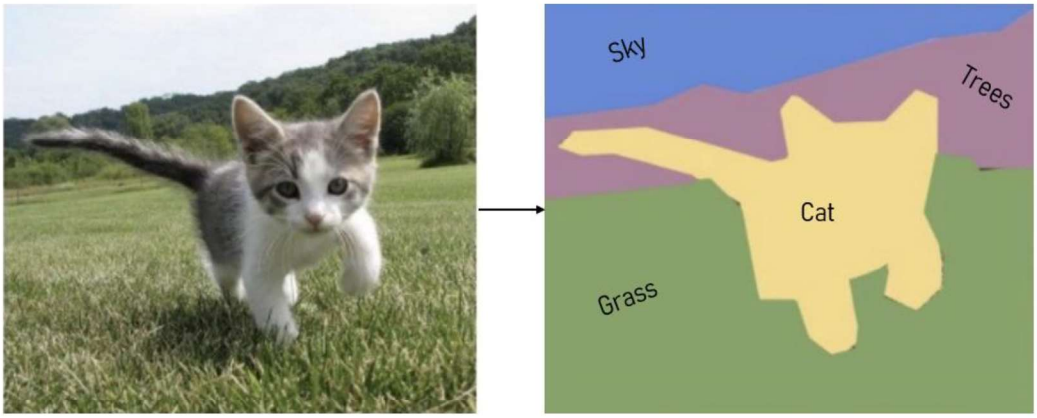
Image segmentation is the process of partitioning an image into multiple regions or segments, with each segment representing a distinct object or region of interest. The goal is to group pixels that have similar properties, such as color, intensity, texture, or spatial proximity. There are various techniques for image segmentation, including:

Thresholding: Separating objects based on pixel intensity values using global or adaptive thresholding.

Edge-based segmentation: Detecting and tracing object boundaries using edge detection algorithms like Canny, Sobel, or Laplacian.

Region-based segmentation: Assigning pixels to different regions based on properties such as color, texture, or motion. Popular methods include region growing, watershed segmentation, and graph cuts.

Clustering: Grouping pixels into clusters based on their feature similarity using techniques like k-means, mean-shift, or spectral clustering.



Solved Exercise:

```
import cv2
import numpy as np

# path to input image is specified and
# image is loaded with imread command
image1 = cv2.imread('input1.jpg')

# cv2.cvtColor is applied over the
# image input with applied parameters
# to convert the image in grayscale
img = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY)

# applying different thresholding
# techniques on the input image
# all pixels value above 120 will
# be set to 255
ret, thresh1 = cv2.threshold(img, 120, 255, cv2.THRESH_BINARY)
ret, thresh2 = cv2.threshold(img, 120, 255, cv2.THRESH_BINARY_INV)
ret, thresh3 = cv2.threshold(img, 120, 255, cv2.THRESH_TRUNC)
ret, thresh4 = cv2.threshold(img, 120, 255, cv2.THRESH_TOZERO)
ret, thresh5 = cv2.threshold(img, 120, 255, cv2.THRESH_TOZERO_INV)

# the window showing output images
# with the corresponding thresholding
# techniques applied to the input images
cv2.imshow('Binary Threshold', thresh1)
cv2.imshow('Binary Threshold Inverted', thresh2)
cv2.imshow('Truncated Threshold', thresh3)
cv2.imshow('Set to 0', thresh4)
cv2.imshow('Set to 0 Inverted', thresh5)

# De-allocate any associated memory usage
if cv2.waitKey(0) & 0xff == 27:
    cv2.destroyAllWindows()
```

Lab Exercises:

1. Write a program to create binary images using thresholding methods.
2. Write a program to detect lines using Hough transform.
3. Write a program to segment an image based on colour.
4. Implement K means clustering algorithm.

Lab No 5: Date:

Implementation of Feature Extraction Methods

Objectives:

In this lab, student will be able to

1. Understand the importance of feature extraction for representation learning
2. Implement feature extraction methods such as HOG, SIFT, LBP and FAST
3. Use task specific features like edge detection, texture analysis, or shape descriptors

Feature extraction aims to learn meaningful and informative representations of visual data. By extracting discriminative features, the model can capture important characteristics of images or videos that are relevant to the task at hand. These representations help in distinguishing between different objects, recognizing patterns, and understanding the visual content. Some of the examples are Histogram of Oriented Gradients (HOG), Scale Invariant Feature Transform (SIFT), Local Binary Pattern (LBP).

HOG Features: HOG features have gained widespread popularity for object detection tasks. They involve breaking down an image into small square cells, calculating a histogram of oriented gradients within each cell, normalizing the output using a block-wise pattern, and generating a descriptor for each cell.

By arranging these cells into a squared image region, they can serve as descriptors for image windows, which are useful for object detection. For instance, support vector machines (SVMs) can be employed to leverage these descriptors and facilitate the detection process.

SIFT: The SIFT combines both a feature detector and a feature descriptor. The detector is responsible for extracting a set of frames (attributed regions), from an image. These frames are selected in a manner that is robust to variations in illumination, viewpoint, and other viewing conditions.

The descriptor, on the other hand, associates each of these regions with a compact and reliable signature that characterizes their appearance. This signature enables the regions to be identified consistently and effectively, even in the presence of various transformations and changes.

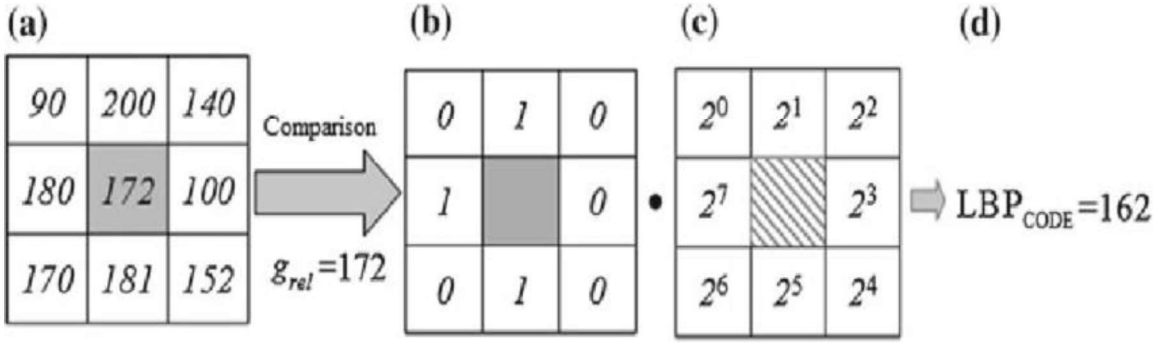
SIFT consists of following steps:

- **Scale-space Construction:** Apply Gaussian filters at multiple scales to create a scale pyramid.
- **Difference-of-Gaussian (DoG) Images:** Compute the difference between adjacent scales in the scale pyramid.
- **Keypoint Detection:** Identify local extrema in the DoG images as potential keypoints.
- **Keypoint Localization:** Refine the positions of keypoints based on contrast, curvature, and precise localization.

- **Orientation Assignment:** Assign a dominant orientation to each keypoint based on local gradient directions.
- **Keypoint Description:** Generate a descriptor for each keypoint by considering local image gradients.
- **Keypoint Matching:** Compare and match keypoints across different images using descriptors.
- **Output:** Obtained matched keypoints or further downstream processing based on the application.

LBP: It is a *texture descriptor* widely used in computer vision for analyzing and classifying images based on their local texture patterns. One of the key strengths of LBP lies in its ability to handle variations in grayscale caused by factors like changes in illumination. This robustness to monotonic gray-scale changes makes LBP particularly valuable in real-world scenarios. Additionally, LBP offers the advantage of computational simplicity, allowing for efficient image analysis even in demanding real-time environments. The steps are as follows:

- **Image Partitioning:** Divide the image into small neighborhoods or pixels of interest.
- **Pixel Comparison:** For each pixel in the neighborhood, compare its intensity value to the intensities of its surrounding neighboring pixels.
- **Binary Code Generation:** Generate a binary code based on the pixel comparisons. Assign a value of 1 if the intensity of a neighboring pixel is greater than or equal to the central pixel's intensity; otherwise, assign a value of 0.
- **Decimal Conversion:** Convert the binary code to decimal form, treating it as a binary number. This decimal value represents the texture pattern of the central pixel.
- **Histogram Calculation:** Compute histograms of the LBP values for the entire image or specific regions of interest. These histograms capture the distribution of different texture patterns within the image or region.
- **Feature Extraction:** Utilize the generated LBP histograms as texture descriptors for various computer vision tasks. They can be directly used for tasks such as texture classification or combined with other feature descriptors for more comprehensive image analysis.
- **Output:** Obtain the LBP histograms or further downstream processing based on the application.



Solved Exercise:

Given below is a program to compute SIFT descriptor using python OpenCV library.

```
#Implementation of SIFT descriptor using Opencv library
import cv2
import matplotlib.pyplot as plt
from cv2 import SIFT_create
import numpy as np
# read images
img = cv2.imread(r'Pics\tower_1.jpg')
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)


# create sift object
sift = SIFT_create()
keypoints, descriptors = sift.detectAndCompute(img, None)
keypoints_with_size = np.copy(img)


cv2.drawKeypoints(img, keypoints,  keypoints_with_size, color = (255, 0, 0),
flags = cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
plt.imshow(keypoints_with_size),
plt.show()
```

Lab Exercises:

1. Implement basic feature extraction algorithms given below:
   i Harris corner detection
   ii FAST corner detection.

   Apply it to different images and visualize the detected keypoints.
2. Implement your own version of the feature descriptors given below and compare them:
   i SIFT
   ii SURF

   Assess the robustness of descriptors to changes in scale, rotation, and affine transformations. Also, compare your implementation with the descriptors available in the opencv library. Use the earlier version of opencv (prior to OpenCV 3.4.3) for the SIFT.
3. Implement the HoG descriptor and apply it to detect humans in images. You may follow the steps given below:

   a) Obtain a dataset containing images with humans and non-human objects. Compute HoG for these images as references.

   b) Extract HoG features using a sliding window approach.

c) Calculate a similarity score or distance metric between the HoG descriptor of the window and a reference HoG descriptor.
c) Identify and collect windows that exceed the similarity score threshold. Choose the best window among the overlapping windows.

Additional Exercise:

1. Explore techniques to improve human detection results, such as multi-scale analysis, or gradient normalization with HoG descriptor.
2. Apply feature descriptors in specific computer vision tasks. For example, use feature descriptors for image retrieval.
3. Design experiments to assess their computational efficiency, and robustness to noise or occlusions.
4. Explore the invariance properties of feature descriptors. Assess the robustness of descriptors to changes in scale, rotation, and affine transformations.
5. Implement LBP based face recognition.
6. Implement Fourier shape descriptor for a shape-retrieval task. Asses the performance using precision, recall and F-measure.

Lab No 6: Date:

Implementation of Feature Matching Methods

Objectives :

In this lab, student will be able to

1. Implement different algorithms and approaches for matching features between images or frames.
2. Explore robust estimation methods like Random Sample Consensus (RANSAC) to handle outliers and improve the accuracy of feature matching.
3. Apply feature matching for various applications such as image stitching, object rccognition, and visual tracking.
4. Comprehend geometric constraints used in feature matching, including affine transformations, epipolar constraint, and their applications in tasks like image registration and 3D reconstruction.

Feature matching is a fundamental task in computer vision that aims to find correspondences between similar features in different images or frames of a video. It involves identifying and matching distinctive visual features, such as corners, edges, or descriptors, between multiple images or frames. In feature matching, the first task is to pick a matching. The second task is to develop efficient data structures and algorithms that enable fast and effective matching.

Brute-force approach: Brute-force feature matching is a straightforward approach where each feature in one set is compared with every feature in another set to find the best matches. It involves computing a distance or similarity measure between feature descriptors and selecting the pairs with the lowest distance or highest similarity. While this method guarantees accurate correspondence, it can be computationally expensive for large datasets. Techniques like approximate nearest neighbor search or distance threshold filtering can improve efficiency. Brute-force matching is a reliable method, but it may not be suitable for real-time or large-scale applications.

Fast Approximate Nearest Neighbour Search (FLANN): FLANN is a library that helps find approximate matches to a given query point in a large dataset. It uses efficient data structures and algorithms to speed up the search process. FLANN is commonly used in tasks like feature matching and image retrieval, where finding the closest matches between descriptors is important. It offers faster search times compared to exhaustive methods by utilizing techniques like hierarchical clustering and randomized sampling. FLANN is a valuable tool for matching and searching large datasets while maintaining reasonably accurate results.

K-Nearest Neighbors (KNN) Matching: KNN feature matching is a technique in computer vision that finds the closest neighbors of a given query feature in a feature space. It relies on the assumption that similar features have similar representations in this space. The process involves constructing a feature database, representing features as high-dimensional vectors, and then searching for the K nearest neighbors to a query feature. The similarity between features is

measured using distance metrics like Euclidean distance or cosine similarity. KNN feature matching is valuable for tasks like image retrieval, object recognition, and image stitching, enabling efficient comparison of visual content in large datasets.

RANSAC: RANSAC (Random Sample Consensus) is an algorithm used for feature matching in computer vision. It helps find corresponding features between images even when there are mismatches or incorrect matches. The algorithm randomly selects a subset of feature correspondences, estimates a model based on them, and determines the inliers that agree well with the estimated model. This process is repeated multiple times to find the model with the largest consensus set. RANSAC is robust to outliers and noise in the data, making it useful in applications like image registration and 3D reconstruction.

Sample Exercise:

Given below is a program to compute SIFT descriptor and find the correspondence in two images using a brute force matching.

```
import cv2
import matplotlib.pyplot as plt
from cv2 import SIFT_create

# read images
img1 = cv2.imread(r'Pics\tower_1.jpg')
img2 = cv2.imread(r'Pics\tower_2.jpeg')
img2 = cv2.resize(img2, (750, 1200))
img1 = cv2.cvtColor(img1, cv2.COLOR_BGR2RGB)
img2 = cv2.cvtColor(img2, cv2.COLOR_BGR2RGB)

# sift
sift = SIFT_create()

keypoints_1, descriptors_1 = sift.detectAndCompute(img1, None)
keypoints_2, descriptors_2 = sift.detectAndCompute(img2, None)

# feature matching
bf = cv2.BFMatcher(cv2.NORM_L1, crossCheck=True)

matches = bf.match(descriptors_1, descriptors_2)
matches = sorted(matches, key=lambda x: x.distance)

line_color = (255, 0, 0)
img3 = cv2.drawMatches(img1, keypoints_1, img2, keypoints_2, matches[:150],
         img2, matchColor=line_color, singlePointColor=line_color, flags=2)
plt.imshow(img3),
plt.show()
```

Lab Exercises:

1. Implement the nearest neighbor matching algorithm for feature descriptors. You may use any of the descriptor implemented earlier. Given two sets of descriptors, calculate the pairwise distances and find the best match for each descriptor.
2. Implement the ratio test for feature matching. Given a set of descriptors and their nearest neighbors, apply the ratio test to filter out unreliable matches and keep only the robust matches.
3. Implement the homography estimation algorithm using feature matches. Given a set of matched keypoints, use techniques like RANSAC to estimate the homography matrix that represents the geometric transformation between two images.
4. Implement an image stitching algorithm using feature matching. Extract features from overlapping images, match the descriptors, estimate the transformation, and stitch the images together to create a seamless panoramic image.

Additional Exercise:

1. Implement an image retrieval system using feature matching. Build a database of object images, extract features, and match them with the query image.
2. Implement a real-time feature matching pipeline using efficient feature descriptors such as ORB or BRISK. Introduce various image transformations such as rotation, scaling, and perspective distortion to an image. Extract feature descriptors from the transformed image and compare them with the descriptors from the original image.
   Measure the matching performance under different transformations to assess the robustness of the descriptors.
3. Implement an object tracking system that can track a specific object across a video sequence. Extract feature descriptors from the initial frame containing the target object. Match the descriptors between subsequent frames to track the object's movement. Evaluate the tracking accuracy by comparing the tracked object's position with ground truth annotations.

Lab No 7: Date:

Implementation of Camera Calibration

Objectives:

In this lab, student will be able to

1. Gain a conceptual understanding of camera calibration, including intrinsic and extrinsic camera parameters and explain mathematical models used to represent camera geometry and distortion.
2. Understand the challenges and considerations involved in calibrating cameras in real-world scenarios.
3. Determine the extrinsic parameters of the camera, including the rotation and translation matrix that define its position and orientation.

The pinhole camera model provides a fundamental basis for understanding camera geometry and image formation. While it simplifies the complexities of real-world cameras, it serves as a starting point for more advanced camera models and calibration techniques. The pinhole camera model is a simplified mathematical model used to describe the geometric principles of a camera's imaging process. It assumes that light enters the camera through a single small aperture or "pinhole" and forms an inverted image on the camera's image plane.

Extrinsic and intrinsic parameters are two types of camera parameters that describe the geometry and properties of a camera.

Intrinsic parameters: Intrinsic parameters are specific to the camera itself and do not change regardless of the scene or objects being captured. They relate to the internal characteristics of the camera's imaging system and optics. Example: focal length, Principal Point, Lens Distortion, Pixel Size etc.

Extrinsic Parameters: Extrinsic parameters describe the camera's position and orientation in the 3D world with respect to a specific coordinate system. They define the camera's external geometry and vary as the camera changes position or orientation. Example: Rotation Matrix (R), Translation Vector (T), Camera Pose:

Camera calibration: The calibration algorithm assumes a pinhole camera model. The equation given below provides the transformation that relates a world coordinate [X Y Z] and the corresponding image point [x y]:

$$w\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = K[R \quad t]\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

w: arbitrary scale factor

K: camera intrinsic matrix

$$\mathbf{K} = \begin{bmatrix} f_x & \gamma & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

R: matrix representing the 3-D rotation of the camera

t: translation of the camera relative to the world coordinate system

Camera calibration involves determining the intrinsic parameters, extrinsic parameters, and distortion coefficients. The process can be divided into two steps:

- In the first step, the intrinsic and extrinsic parameters are calculated assuming zero lens distortion, using a closed-form solution.
- In the second step, all the parameters, including distortion coefficients, are estimated simultaneously using nonlinear least-squares minimization. The initial estimates for the intrinsics and extrinsics are based on the closed-form solution obtained in the previous step, while the initial estimate for the distortion coefficients is set to zero.

Solved Exercise:

Given below is a program to estimate the projection matrix P, given a sequence of check board images taken from different views.

```
import cv2
import numpy as np
import os
import glob
# Defining the dimensions of checkerboard
CHECKERBOARD = (12, 12)
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 50, 0.0001)
# Creating vector to store vectors of 3D points for each checkerboard image
objpoints = []
# Creating vector to store vectors of 2D points for each checkerboard image
imgpoints = []
# Defining the world coordinates for 3D points
objp = np.zeros((1, CHECKERBOARD[0] * CHECKERBOARD[1], 3), np.float32)
objp[0, :, :2] = np.mgrid[0:CHECKERBOARD[0], 0:CHECKERBOARD[1]].T.reshape(-1,
         2)
prev_img_shape = None

# Extracting path of individual image stored in a given directory
images = glob.glob('./calib_example/*.tif')
```

```
    for fname in images:
        img = cv2.imread(fname)
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        # Find the chess board corners
        # If desired number of corners are found in the image then ret = true
        ret, corners = cv2.findChessboardCorners(gray, CHECKERBOARD,
                                                 cv2.CALIB_CB_ADAPTIVE_THRESH +
             cv2.CALIB_CB_FAST_CHECK + cv2.CALIB_CB_NORMALIZE_IMAGE)
        #If successful display on the image
         if ret == True:
            objpoints.append(objp)
            # refining pixel coordinates for given 2d points.
            corners2 = cv2.cornerSubPix(gray, corners, (11, 11), (-1, -1), criteria)

            imgpoints.append(corners2)

            # Draw and display the corners
            img = cv2.drawChessboardCorners(img, CHECKERBOARD, corners2, ret)

        cv2.imshow('img', img)
        cv2.waitKey(0)

    cv2.destroyAllWindows()

    h, w = img.shape[:2]

    """
    Performing camera calibration by
    passing the value of known 3D points (objpoints)
    and corresponding pixel coordinates of the
    detected corners (imgpoints)
    """
    ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints,
             gray.shape[::-1], None, None)

    print("Camera matrix : \n")
    print(mtx)
    print("dist : \n")
    print(dist)
    print("rvecs : \n")
    print(rvecs)
    print("tvecs : \n")
    print(tvecs)
```

Lab Exercises:

1. Extend the solved exercise to estimate the intrinsic parameter and the extrinsic parameter.

2. Verify the calibration results by reprojecting the feature points onto the original images using the estimated parameters.

Additional Exercise:

1. Assume that you are using a fisheye lens camera for capturing an image. Develop an application to eliminate the distortion in the captured image. You may make suitable assumptions to develop the working model.

Lab No 8: Date:

Implementation of Tracking Methods

Objectives :

In this lab, student will be able to

1. Understand and implement different types of features and descriptors commonly used in object tracking algorithms.
2. Identify the factors contributing to tracking failures and assess the tracker's ability to recover from failures.
3. Explain evaluation metrics used for assessing the performance of object trackers, such as precision, recall, intersection over union (IoU),

Object tracking refers to the process of automatically following and monitoring the movement of specific objects of interest in a video sequence. It involves locating and keeping track of the object's position and/or its bounding box across consecutive frames. Object tracking is a fundamental task in computer vision and has numerous applications, including surveillance, activity recognition, augmented reality, autonomous vehicles, and robotics.

Vision-based object trackers rely on visual information extracted from video frames to track objects. These trackers utilize various techniques, algorithms, and features to accomplish the task. Here are some common approaches used in vision-based object trackers:

Template Matching: Template matching involves comparing a template (representing the object appearance) with image regions in subsequent frames. It measures the similarity between the template and the search area using correlation or similarity metrics. The template is updated over time to adapt to appearance changes.

Feature-Based Trackers: Feature-based trackers detect and track specific features or keypoints on the object. Feature descriptors, such as SIFT, SURF, or ORB, are used to describe the object's appearance and distinguish it from the background. Matching techniques, such as nearest neighbor search or geometric verification, are employed to find correspondences between feature points across frames.

Optical Flow: Optical flow algorithms estimate the apparent motion of objects by analyzing the movement of pixels or feature points between frames. They rely on the assumption that the intensity pattern of objects remains constant over time. Methods like Lucas-Kanade or Horn-Schunck can estimate the dense optical flow, which can be used for object tracking.

Correlation Filters: Correlation filter-based trackers learn a correlation filter that can estimate the position of the object based on its appearance. They often use feature descriptors, such as HOG or deep convolutional features, to represent the object appearance. The correlation filter is updated online to adapt to appearance changes.

These are some of the common vision-based object tracking approaches. Each approach has its own advantages, trade-offs, and suitability for different tracking scenarios. Choosing the right tracker depends on factors such as the nature of the objects being tracked, computational

requirements, real-time performance, robustness to appearance changes, and occlusions.

Solved Exercise:

Given below is a program to track an object using CamShift tracker.

```
import cv2
# Read the video
cap = cv2.VideoCapture(' slow_traffic_small.mp4')

# Read the first frame
ret, frame = cap.read()
# Set the ROI (Region of Interest)
x, y, w, h = cv2.selectROI(frame)
# Initialize the tracker
roi = frame[y:y+h, x:x+w]
roi_hist = cv2.calcHist([roi], [0], None, [256], [0,256])
roi_hist = cv2.normalize(roi_hist, roi_hist, 0, 255, cv2.NORM_MINMAX)
term_crit = (cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 10, 1)

while True:
    ret, frame = cap.read()
    if not ret:
        break

    # Convert the frame to HSV color space
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)

    # Calculate the back projection of the histogram
    dst = cv2.calcBackProject([hsv], [0], roi_hist, [0,256], 1)

    # Apply the CamShift algorithm
    ret, track_window = cv2.CamShift(dst, (x,y,w,h), term_crit)

    # Draw the track window on the frame
    pts = cv2.boxPoints(ret)
    pts = np.int0(pts)
    img2 = cv2.polylines(frame, [pts], True, (0,255,0), 2)

    # Display the resulting frame
    cv2.imshow('frame',img2)

    # Exit if the user presses 'q'
    if cv2.waitKey(1) == ord('q'):
        break

cap.release()
cv2.destroyAllWindows()
```

Lab Exercises:

1. Implement Lucas-Kanade (LK) Tracker. Apply the Lucas-Kanade algorithm for optical flow estimation to track feature points between frames. Utilize feature descriptors, such as SIFT or SURF, to compute the similarity between feature points across frames. Estimate the motion model, such as affine or homography, based on the matched feature points.
2. Implement Kanade-Lucas-Tomasi (KLT) Tracker. Extract feature points, such as corners or blobs, from the first frame using a feature detector like Harris corner detector or FAST. Track the feature points in subsequent frames using the KLT algorithm. Utilize feature descriptors, such as local binary patterns (LBP) or intensity gradients, to match the tracked points across frames. Estimate the motion between frames based on the matched feature points. Identify and list the cases where the tracking algorithm fails. Develop a strategy to address this issue.
3. Implement an object tracking algorithm using the Kalman filter. Use a video sequence with a single moving object. Initialize the Kalman filter with the object's initial position and velocity. Predict the object's position in subsequent frames and update the filter based on the observed object position. Evaluate the tracking method in terms of precision, recall and Intersection over Union(IoU).

Additional Exercises:

1. Implement a simple object tracking algorithm using the OpenCV library. Choose a video sequence and manually select the object of interest in the first frame. Use mean-shift algorithm to track the object in subsequent frames. Visualize the tracking results by overlaying the object's bounding box on each frame of the video.
2. Implement a multi-object tracking system capable of tracking multiple objects simultaneously. Use datasets with multiple moving objects and develop algorithms to track and maintain identity associations among the objects ex: people tracker. Develop an object tracker that can handle occlusions, where the tracked object is temporarily obscured by other objects.

References:

1. Rafael C. Gonzalez, Richard E. Woods, "Digital Image Processing", Third Edition, Pearson Education, 2012.
2. Richard Szeliski, Computer Vision: Algorithms and Applications, Springer 2011.
3. Milan Sonka, Vaclav Hlavac, Roger Boyle, "Digital Image Processing and Computer Vision", India Edition, Cengage Learning 2009.
4. Joseph House, Prateek Joshi, Michael Beyeler, *Open CV: Computer Vision Projects with Python,* Packt Publishing, 2016
5. Daniel Lélis Baggio, Shervin Emami, David Millán Escrivá, Khvedchenia Ievgen, Naureen Mahmood, Jason Saragih, Roy Shilkrot, *Mastering Open CV3*, Second Edition, Packt Publisher, 2017
6. Robert Laganiere. *OpenCV Computer Vision Application Programming Cookbook*, 2nd. ed. Packt Publishing, 2014.