Assignment-3
Advanced Computer Vision

-By Ruchir Namjoshi
002803974

(NOTE : For each question I have explained the code and how to execute the file while writing the solution for that question, therefore a separate readme file is not created for this purpose)

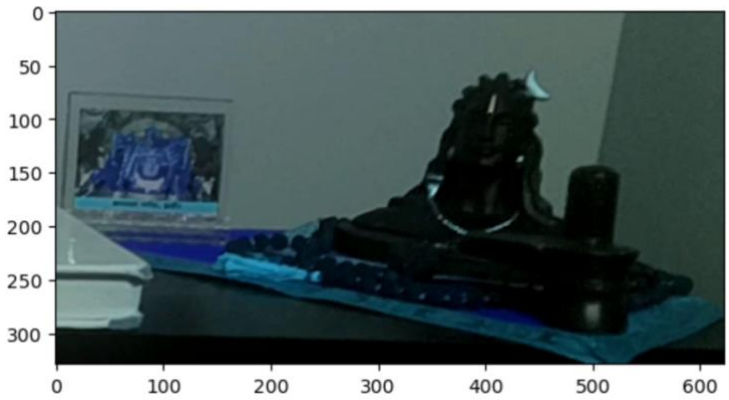
A1. Compare cropped region with randomly picked 10 images in the dataset of 10 sec video

(Execute the IPYNB file for question 1 and keep the 10 second video in the same directory as the IPYNB file)

I have used series of steps to process a video and compare selected regions of interest (ROI) with other frames. Here's a detailed step-by-step explanation of my code:

1. **Selecting and Cropping a Frame from the Video**
The function `select_frame_and_crop` loads a video file, randomly selects a frame, and allows the user to interactively select a region of interest (ROI) in that frame, which is then cropped and returned.
(This process and the whole code is shown in a video)

-**Video Capture:** Initialize video capture from the provided video path.
- **Frame Selection**: Randomly choose one frame from the video based on its total frame count.
- **ROI Selection**: Display this frame to the user and allow them to select an ROI using OpenCV's built-in `selectROI` function.
- **Cropping**: Crop the selected ROI from the frame and return this cropped region.



THIS IS THE CROPPED IMAGE

2. **Extracting Random Frames from the Video**
The function `extract_random_frames` extracts a specified number of frames(10 here) randomly from the same video and saves them to the disk.

- **Video Capture and Frame Selection:** Similar to the previous function, but this time we select multiple frames randomly without replacement.
- **Saving Frames:** Each selected frame is saved locally with a unique filename.

3. **Comparing the Cropped Region with Other Frames**
The function `compare_with_frames_fast` uses OpenCV's template matching to compare the cropped region against each frame saved in step 2.

- **Template Matching:** For each frame, apply the template matching method `cv2.matchTemplate` using normalized squared differences (`TM_SQDIFF_NORMED`), which is good for matching with minimized error.
- **Finding the Best Match:** Locate the point with the minimum squared difference, which represents the best match for the template in the frame.
- **Storing Results:** Save and sort the results based on the matching score to determine the closest matches.

4. **Displaying and Saving Results**
The function `display_results` visualizes the best matches by drawing rectangles around the matched regions in the top-matching frames.

- **Reading and Annotating:** For each top result, read the frame, draw a rectangle around the matched region, and display it.
- **Saving Annotated Frames:** Save these annotated frames to disk for further reference.
- **User Interaction:** Allow the user to view the results with drawn rectangles and wait for a key press to proceed.

5. **Main Execution Block**
This part of your script initiates the process by calling the functions defined above:

- **Cropping Region:** First, a frame is selected and a region is cropped as per user input.
- **Random Frame Extraction:** Then, a set of random frames is extracted from the video.
- **Comparison and Results:** The cropped region is compared against these frames using the faster comparison method.
- **Result Display:** Finally, the results are displayed and saved.

```
Frame: frame_98.png, SSD: 3.798207330873993e-07, Position: (1244, 225)
Match 1 saved to Match 1.png
Frame: frame_95.png, SSD: 0.0026936146896332502, Position: (1245, 217)
Match 2 saved to Match 2.png
Frame: frame_82.png, SSD: 0.016417915001511574, Position: (1248, 194)
Match 3 saved to Match 3.png
Frame: frame_113.png, SSD: 0.01679794304072857, Position: (1198, 245)
Match 4 saved to Match 4.png
Frame: frame_124.png, SSD: 0.021952185779809952, Position: (1180, 239)
Match 5 saved to Match 5.png
Frame: frame_68.png, SSD: 0.029342766851186752, Position: (1280, 190)
Match 6 saved to Match 6.png
Frame: frame_132.png, SSD: 0.03784714266657829, Position: (1159, 228)
Match 7 saved to Match 7.png
Frame: frame_44.png, SSD: 0.20459090173244476, Position: (1297, 159)
Match 8 saved to Match 8.png
Frame: frame_33.png, SSD: 0.28122052550315857, Position: (1297, 152)
Match 9 saved to Match 9.png
Frame: frame_24.png, SSD: 0.3395932614803314, Position: (1297, 150)
Match 10 saved to Match 10.png
```

THIS IS THE SSD FROM EACH ONE OF THE RANDOM FRAME

A2.
2a) Derivation of Motion Tracking Equation-

Suppose we have an image at time t and an image at time $t + \delta t$.

We assume a single point in the image 1 – (x,y)
Location of this point in the second image would be $(x + \delta x\ , y + \delta y)$

Diplacement of this point will be –
Displacement = $(\delta x\ , \delta y)$

Let us assume speed of the point in x and y direction is (u,v)

Optical flow - (u,v) = $\left(\frac{\delta x}{\delta t}\ , \frac{\delta y}{\delta t}\right)$

Assumption 1 – Brightness of image point remains constant over time

$I(x + \delta x\ , y + \delta y, t + \delta t) = I(x\ , y, t)$ ……Equation 1.

Assumption 2 – Displacement and time step are small.

We know by taylor series –

$f(x + \delta x) = f(x) + \ \frac{\partial f}{\partial x}\ \delta x + \frac{\partial^2 f}{\partial x^2}\frac{\delta x^2}{2!} + \cdots \ldots\ldots + \frac{\partial^n f}{\partial x^n}\frac{\delta x^n}{n!}$.

If δx is very small :

$f(x + \delta x) = f(x) + \ \frac{\partial f}{\partial x}\ \delta x + \ 0(\delta x).$

For a function of 3 variables with δx ,δy and δt

$f(x + \delta x, y + \delta y, t + \delta t) = f(x, y, t) + \ \frac{\partial f}{\partial x}\ \delta x + \frac{\partial f}{\partial y}\ \delta y + \frac{\partial f}{\partial t}\ \delta t$.

Thus we can write
$I(x + \delta x, y + \delta y, t + \delta t) = I(x, y, t) + \ \frac{\partial I}{\partial x}\ \delta x + \frac{\partial I}{\partial y}\ \delta y + \frac{\partial I}{\partial t}\ \delta t.$

We can use these expression instead of the derivative term

$I(x + \delta x, y + \delta y, t + \delta t) = I(x, y, t) + \ \ I_x\ \delta x + \ I_y \delta y + \ I_t \delta t$……Equation 2

Subtract Equation 1 from Equation 2 we get

$I_x\ \delta x + \ I_y \delta y + \ I_t \delta t = 0.$

Now we divide this equation with δt andtake limit as δt -> 0 we would get.

$$I_x \frac{\partial x}{\partial t} + I_y \frac{\partial y}{\partial t} + I_t = 0$$

By our assumption of (u,v) we get

$$I_x u + I_y v + I_t = 0$$

Now using this and finding motion estimate of 2 consecutive frame I have taken a 3*3 patch in each image.

To illustrate manual calculation of optical flow using a small example, let's assume a simplified scenario with two consecutive frames of an image, each containing only a few pixels. This example will provide a hands-on demonstration of the optical flow concept.

Representing image intensity levels at times $t \text{ and } t + \delta t$
:

Frame 1 (Time t):
[124, 124, 124
124, 150, 124
124, 124, 124]

Frame 2 (Time $t + \delta t$):
[124, 124, 124
124, 124, 150
124, 124, 124]

Notice the intensity at the center moves right by one pixel.

Step 1: Calculate Spatial and Temporal Derivatives

For a pixel at position (1,1) (the center pixel in a zero-indexed matrix), calculate $I_x, I_y \; and \; I_t$using forward differences:

-Spatial derivatives:
$I_x = I(x+1, y) - I(x, y)$ = 124 - 150 = -26
$I_y = I(x, y+1) - I(x, y)$ = 1124 - 150 = -26

- Temporal derivative:
$I_t = I(x, y, t+\delta t) - I(x, y, t)$ = 124 - 150 = -26

Step 2: Set Up the Optical Flow Equation

The equation $I_x u + I_y v + I_t = 0$
becomes:

-26u - 26v - 26 = 0

Step 3: Solve for u and v

To solve this equation, we notice that multiple solutions will satisfy this single equation. However, knowing that the change is primarily in the horizontal direction, we can set v = 0 and solve for u :
-26u - 26 = 0 implies u = -1

This solution implies a motion of 1 pixel to the right, which matches our observation of the pixel intensity moving from the middle to the right between the two frames.

2b) Lucas-Kanade algorithm

Assumption – For each pixel, assume motion field and optical flow to be constant within a small neighbourhood.

Adapting the Lucas-Kanade method to handle affine motion models,

The affine motion model can be expressed as:
$u(x, y) = a_1 x + b_1 y + c_1$
$v(x, y) = a_2 x + b_2 y + c_2$

Given two images, $I_1(x, y)$ and $I_2(x, y)$captured at times t and (t + dt), respectively, the Lucas-Kanade method aims to minimize the difference between the two images under the motion model. This is done within a window around each point being considered to provide enough data to solve for the parameters.

Assuming brightness constancy and using Taylor series expansion for $I_2(x + u, y + v)$ we have:
$I_1(x, y) \approx I_2(x + u, y + v).$
$I_1(x, y) \approx I_2(x, y) + I_x u + I_y v.$

Where $I_x$and $I_y$are the gradients of I_2at (x, y) in the x and y directions, respectively.

Substituting the affine model into the Taylor expansion, we get:
$I_x(a_1x + b_1y + c_1) + I_y(a_2x + b_2y + c_2) + I_2(x,y) - I_1(x,y) = 0$.

Expanding this and arranging terms, we aim to solve:

$$I_x\, a_1\, x + I_x\, b_1\, y + I_x\, c_1 + I_y\, a_2\, x + I_y\, b_2\, y + I_y\, c_2 = I_1(x,y) - I_2(x,y)\,.$$

Across all pixels (x_i, y_i) in the window, this equation can be rewritten in matrix form as (Ap = b), where:

$$A = [I_x x\,, I_x y\,, I_x, I_y x\,, I_y y\,, I_y].$$

$$p = \begin{bmatrix} a_1 \\ b_1 \\ c_1 \\ a_2 \\ b_2 \\ c_2 \end{bmatrix}$$

$$b = [I_1(x,y) - I_2(x,y)\,].$$

Solve Using Least Squares

$$p = (A^T A)^{-1} A^T b\,.$$

This results in the estimated parameters $a_1, b_1, c_1, a_2, b_2, c_2$a for the affine motion model within that window.

$$A = \begin{bmatrix} I_x x_1\,, I_x y_1\,, I_x\,, I_y x_1\,, I_y y_1\,, I_y. \\ I_x x_2\,, I_x y_2\,, I_x\,, I_y\ x_2\,, I_y y_2\,, I_y\,. \\ \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots. \\ I_x x_n\,, I_x y_n\ , I_x\,, I_y x_n,\ I_y\ y_n\,, I_y] \end{bmatrix}.$$

$$p = \begin{bmatrix} a_1 \\ b_1 \\ c_1 \\ a_2 \\ b_2 \\ c_2 \end{bmatrix}$$

$$B = \begin{bmatrix} I_1(x_1,y_1) - I_2(x_1,y_1) \\ I_1(x_2,y_2) - I_2(x_2,y_2)\,. \\ \vdots \quad \vdots. \\ I_1(x_n\,,y_n) - I_2(x_n,y_n) \end{bmatrix}$$

A3. Disparity based depth estimation in stereo-vision theory

(Execute the IPYNB file for question 3 and keep the images “image1.jpg” , “image2.jpg” in the same directory as the ipynb file)

1. Load Images

The function **load_images** loads two grayscale images from specified file paths. These images represent two views of a scene captured with a translation between the camera positions (along a horizontal axis). Grayscale is used as it simplifies the computation for feature detection and matching.

- **Image Loading**: Utilizes OpenCV's **imread** with the grayscale flag to ensure the images are loaded in the correct format.
- **Error Handling**: Checks if either image failed to load and raises an exception if so, ensuring that the program doesn't proceed with missing data.

2. Detect and Match Features

The function **detect_and_match_features** detects and matches keypoints between the two images using the ORB (Oriented FAST and Rotated BRIEF) detector. ORB is chosen for its efficiency and effectiveness in handling rotation and scale.

- **ORB Initialization**: ORB is initialized to detect and compute features.
- **Keypoint Detection**: Detects keypoints and computes descriptors in each image independently.
- **Descriptor Matching**: Uses the Brute Force matcher with Hamming distance (suitable for binary descriptors like those from ORB) and cross-check enabled, ensuring mutual best matches.
- **Sorting Matches**: The matches are sorted by their distance, indicating the quality of the match, with lower distances being better. The top 30 matches are selected to ensure that only the best matches are used for depth calculation.

3. Calculate Disparity

The function **calculate_disparity** computes the disparity between corresponding keypoints in the two images. Disparity here refers to the horizontal distance between matched keypoints across the two images.

- **Disparity Calculation**: For each matched pair, the horizontal positions (x-coordinates) are extracted, and their difference (disparity) is calculated.
- **Average Disparity**: Computes the average of these disparities, providing a single measure of how much the scene has shifted horizontally between the two images.

4. Compute Depth

The function **compute_depth** calculates the depth (distance to the scene) using the stereo vision formula that relates focal length, baseline (distance between camera centers), and disparity.

- **Depth Formula**: Uses the formula Depth=Focal Length×Baseline|Disparity|Depth=|Disparity|Focal Length×Baseline, where focal length is in pixels, baseline in the same units as the desired depth (mm in this case), and disparity in pixels.
- **Handling Zero Disparity**: If disparity is zero (unlikely but possible if there's no apparent horizontal shift), the function returns infinity to indicate an undefined or very far distance.

5. Execution and Visualization

In the main script, you initialize parameters, perform the above functions in sequence, and compute the estimated depth. The results are printed, and keypoint matches are visualized.

- **Parameters Initialization**: Defines necessary parameters like focal length and baseline.
- **Function Calls**: Sequentially calls functions to process and analyze the images.
- **Result Display**: Prints calculated disparities and estimated depth, and shows the matches between images using **cv2.drawMatches**.

6. Parameters

D was taken as 220mm and the distance between the 2 camera position was 100 mm
The achieved depth by this method is shown in the snapshot below

```
Average Disparity: 678.68 pixels
Estimated Depth: 220.28 mm
```

A4. Optical flow vectors

(Matlab script is added after each part as pdf)

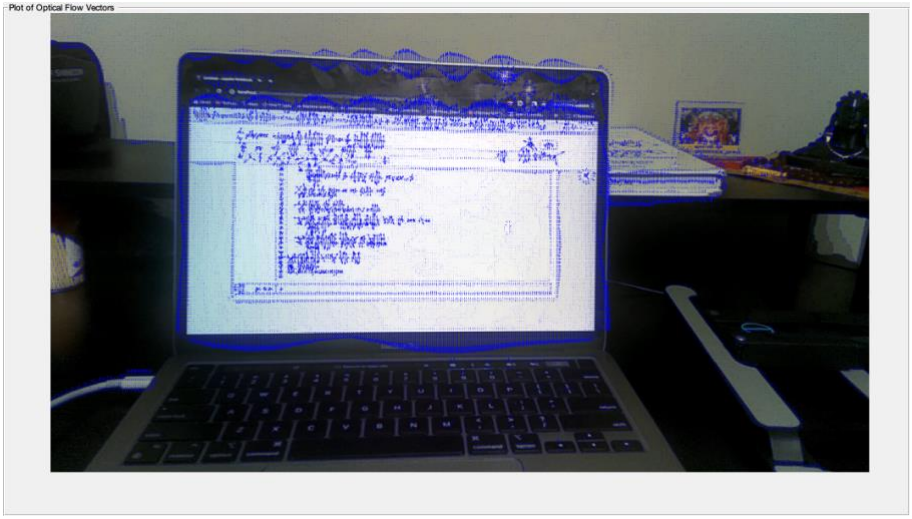
4.1) Every previous frame as a reference frame

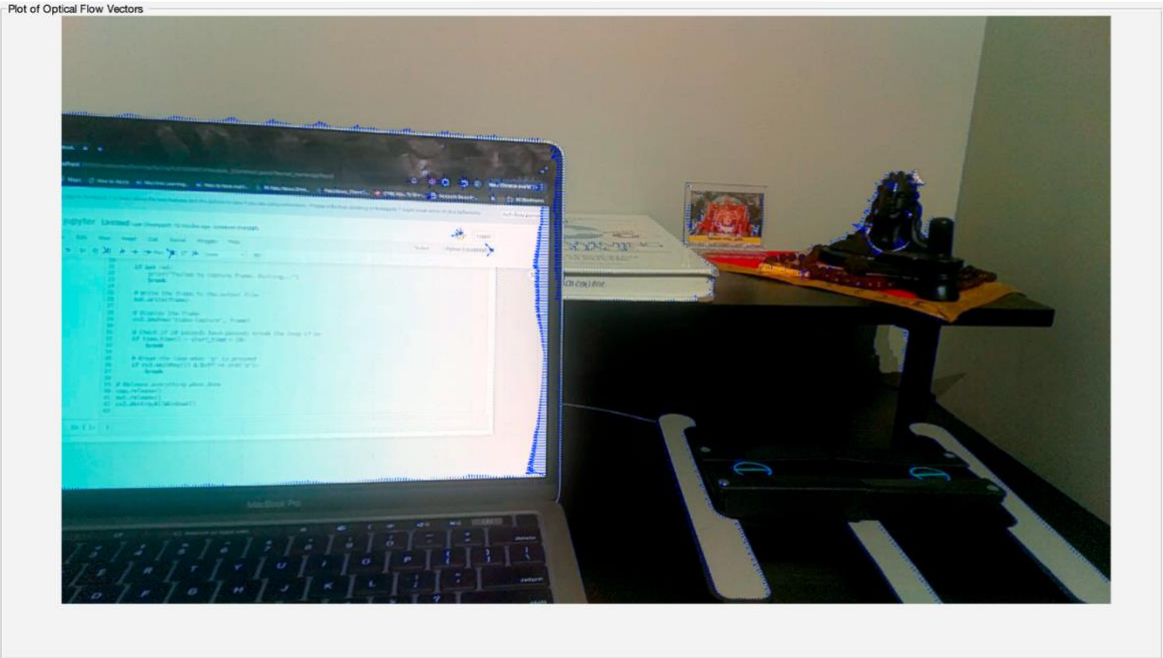
(Execute the live matlab script and keep the 10 second video in the same directory as the live script, Execution of all the 3 parts is shown in one of the videos)

1. **Initialization of Video Reader**: It sets up a video reader for a specified video file. This reader is configured to start processing the video from one second into the clip, skipping the initial frames.
2. **Optical Flow Setup Using Horn-Schunck Method**: An object to compute optical flow using the Horn-Schunck algorithm is created. This method estimates the motion by assuming that the brightness of the object remains constant between consecutive frames and that there is smoothness in the motion across the frame.
3. **Graphical User Interface Setup**: A graphical window is configured to display the video and the optical flow vectors. This includes creating a figure window and a panel within that window which takes up the whole area. The panel is titled to indicate it will display optical flow vectors.
4. **Frame-by-Frame Processing**:
    - The code enters a loop that continues as long as there are more frames to process in the video.
    - Each frame is read from the video file, and it's converted from color to grayscale because the optical flow algorithm requires intensity values to calculate motion.
    - Optical flow for the current frame is estimated using the grayscale version of the frame. This involves calculating the motion vectors that describe the apparent velocities of movement of brightness patterns in the image.
    - The original color frame and the calculated optical flow vectors are displayed. The vectors are superimposed on the color frame to visually depict how different parts of the image are moving from one frame to the next. The vectors are thinned out for clarity and scaled up to be more visible.
    - A very brief pause is included to slow down the looping so that the human eye can follow the motion being depicted.

Overall, this script visually illustrates the motion in the video by displaying flow vectors that represent the direction and speed of movement across frames, helping to analyze dynamic scenes.

```
vidReader = VideoReader('output.mp4','CurrentTime',1);
opticFlow = opticalFlowHS;

h = figure;
movegui(h);
hViewPanel = uipanel(h,'Position',[0 0 1 1],'Title','Plot of Optical Flow
Vectors');
hPlot = axes(hViewPanel);


while hasFrame(vidReader)
    frameRGB = readFrame(vidReader);
    frameGray = im2gray(frameRGB);
    flow = estimateFlow(opticFlow,frameGray);
    imshow(frameRGB)

    hold on
    plot(flow,'DecimationFactor',[5 5],'ScaleFactor',60,'Parent',hPlot);
    hold off
    pause(10^-3)
end
```



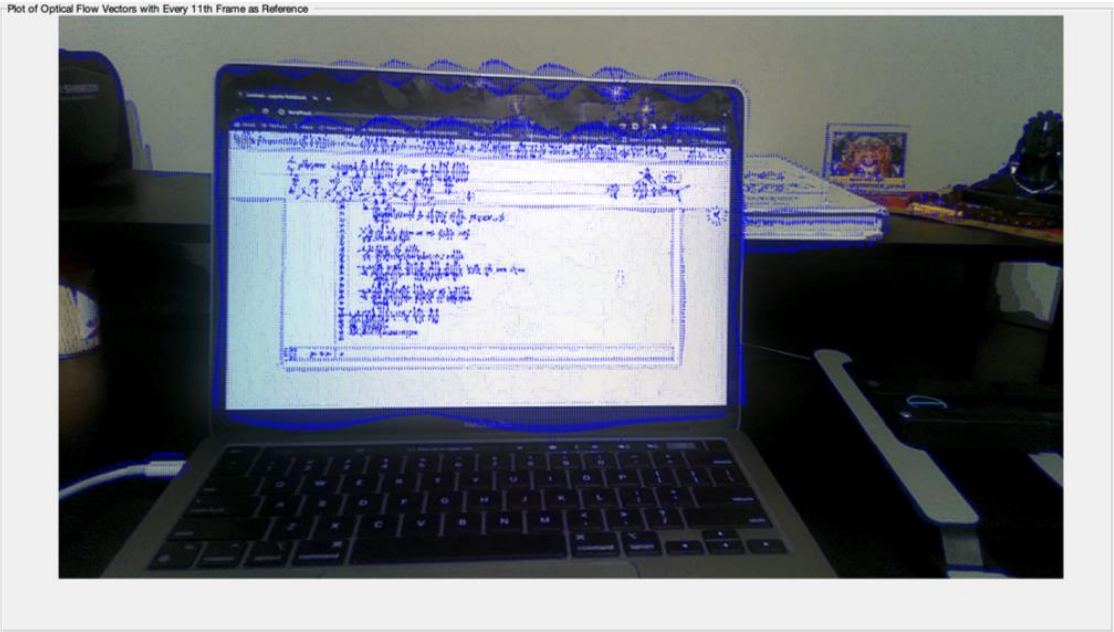


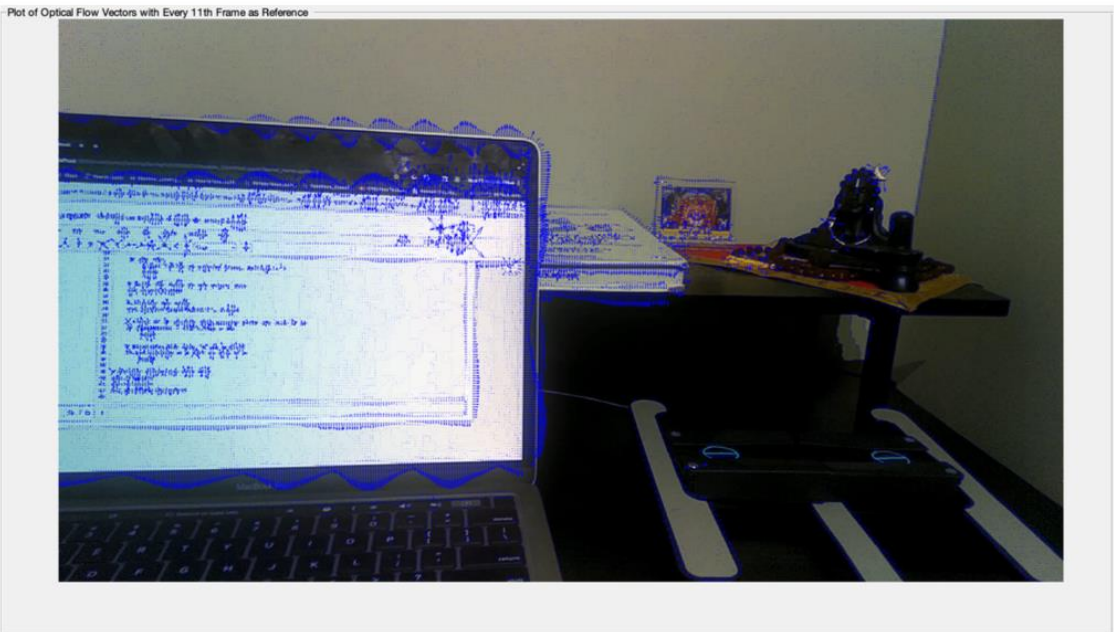
4.2) Every 11th frame as reference frame

Resetting Optical Flow Estimation: In this version, the optical flow estimation is reset every 11th frame. This means that instead of continuously updating the optical flow estimation from the start to the end of the video, the optical flow object (**opticFlow**) is re-initialized every 11th frame.
Frame Counting: A frame counter (**frameCount**) is introduced to keep track of the number of frames processed. This counter is incremented with each frame read from the video.

These changes affect how motion is visualized in the video. By resetting the flow estimation regularly, the visualization might show clear starting points for flow every 11th frame, potentially making it easier to observe changes or anomalies in motion patterns.

```
vidReader = VideoReader('output.mp4', 'CurrentTime', 1);
opticFlow = opticalFlowHS;

h = figure;
movegui(h);
hViewPanel = uipanel(h, 'Position', [0 0 1 1], 'Title', 'Plot of Optical
Flow Vectors with Every 11th Frame as Reference');
hPlot = axes(hViewPanel);

frameCount = 0; % Initialize frame counter

while hasFrame(vidReader)
    frameRGB = readFrame(vidReader);
    frameGray = im2gray(frameRGB);
    frameCount = frameCount + 1;

    % Reset optical flow estimation every 11th frame
    if mod(frameCount, 11) == 0
        opticFlow = opticalFlowHS; % Re-initialize the optical flow object
    else
        flow = estimateFlow(opticFlow, frameGray);
    end

    imshow(frameRGB);

    hold on;
    plot(flow, 'DecimationFactor', [5 5], 'ScaleFactor', 60, 'Parent',
hPlot);
    hold off;
    pause(10^-1);
end
```





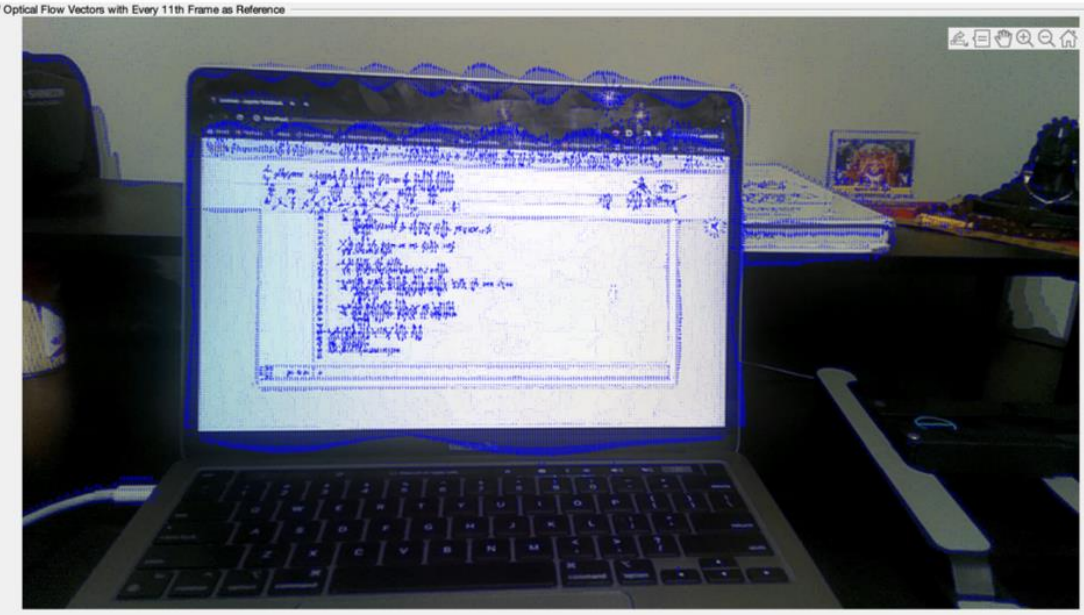
4.3) Every 31st frame as reference frame

Resetting Optical Flow Estimation: In this version, the optical flow estimation is reset every 12th frame.
Frame Counting: A frame counter (**frameCount**) is introduced to keep track of the number of frames processed. This counter is incremented with each frame read from the video.

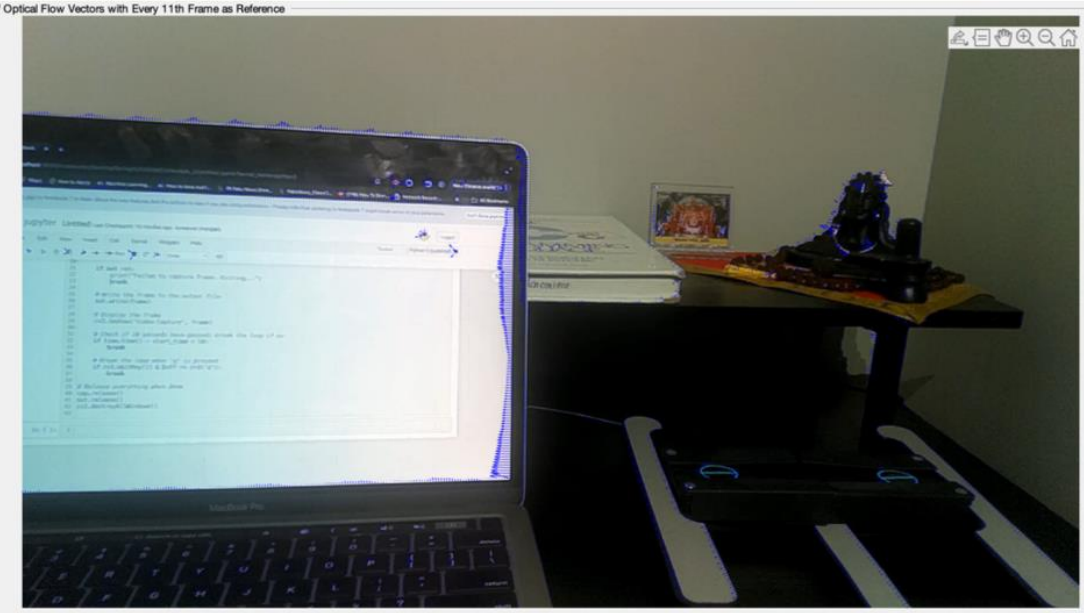
These changes affect how motion is visualized in the video. By resetting the flow estimation regularly, the visualization might show clear starting points for flow every 11th frame, potentially making it easier to observe changes or anomalies in motion patterns.

```
vidReader = VideoReader('output.mp4', 'CurrentTime', 1);
opticFlow = opticalFlowHS;

h = figure;
movegui(h);
hViewPanel = uipanel(h, 'Position', [0 0 1 1], 'Title', 'Plot of Optical
Flow Vectors with Every 11th Frame as Reference');
hPlot = axes(hViewPanel);

frameCount = 0; % Initialize frame counter

while hasFrame(vidReader)
    frameRGB = readFrame(vidReader);
    frameGray = im2gray(frameRGB);
    frameCount = frameCount + 1;

    % Reset optical flow estimation every 31th frame
    if mod(frameCount, 31) == 0
        opticFlow = opticalFlowHS; % Re-initialize the optical flow object
    else
        flow = estimateFlow(opticFlow, frameGray);
    end

    imshow(frameRGB);

    hold on;
    plot(flow, 'DecimationFactor', [5 5], 'ScaleFactor', 60, 'Parent',
hPlot);
    hold off;
    pause(10^-1);
end
```

Plot of Optical Flow Vectors with Every 11th Frame as Reference



Plot of Optical Flow Vectors with Every 11th Frame as Reference



A5. Feature-based matching object detection

(for the ease of execution the code for this question is in the same IPYNB file as the code for question 1, Code from question 1 will save the 10 random files and matched file with cropped area in the same directory just run the code to detect features and visualize them)

Below is my process of matching a template image (a cropped region from a previous frame) against a series of target images (frames).

1. Iterating Through the Results

The loop iterates over the top 10 results from a previously computed comparison (based on sum of squared differences, SSD), which are stored in the **results** list. Each entry in **results** contains a path to an image, the SSD value, and the position of the best match within that image.

2. Loading Images

- **Target Image**: Each target frame's path is used to load the corresponding image in grayscale mode. This is the image where the algorithm will look for the cropped region (the template).
- **Template Image**: This is the previously cropped region from another frame, which will be used as the reference to find similar regions in the target image.

3. Feature Detection and Description

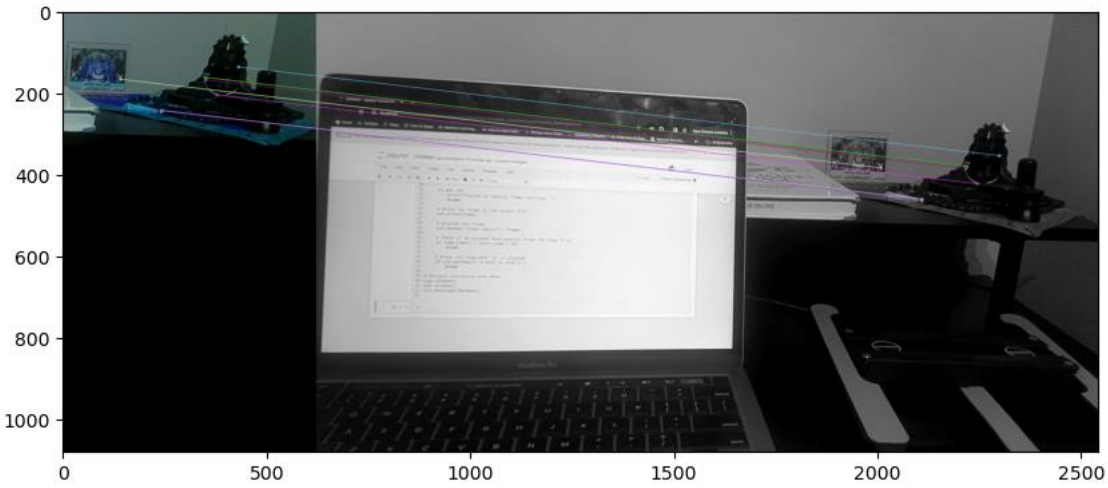
- **ORB Initialization**: ORB (Oriented FAST and Rotated BRIEF) is used to create feature detectors. ORB is a fast robust feature detector that is often used in real-time applications.
- **Keypoint and Descriptor Computation**:
    - For both the template image and the target image, ORB detects keypoints and computes their descriptors. Keypoints are points of interest where the surrounding area shows rich features such as edges, corners, etc.
    - Descriptors are small vectors that uniquely describe the information around keypoints, which makes it possible to match keypoints between different images.

4. Matching Descriptors

- **Matcher Initialization**: A Brute Force matcher using Hamming distance (suitable for binary descriptors like those from ORB) with cross-checking is created. Cross-check ensures that each match is the best match mutually between two images.
- **Matching Process**: The descriptors from the template image are matched against those from the target image. The matches are sorted based on their 'distance' which represents how similar the descriptors are; a lower distance indicates a better match.

5. Visualization of Matches

- **Match Selection**: Only the top 10 matches (sorted by distance) are selected for visualization to keep the display clear and focused on the best matches.
- **Drawing Matches**: **cv2.drawMatches** is used to create a visualization of these matches. It displays lines connecting the matched keypoints between the template image and the target image.
- **Displaying Matched Image**: The resulting matched image is displayed using Matplotlib. A larger figure size is set to better view the details.



This is one example out of 10 randomly chosen frames from the video on how the common features are matched with the cropped region, It is clearly visible through the lines that which features of the frame matches with the cropped region.

A6. Bag of Features

(for the purpose of this question I have taken a data set of cutlery items, containing a folder for each category **spoon, fork, butter knife, cutting knife and ladle** and stored all the folder in a directory named dataset, A live Matlab script was used for this, keep the dataset and the live matlab script in the same directory and execute the script)
(Matlab script is added as pdf)

1. Data Setup

- **Root Folder Specification**: The **rootFolder** variable defines the location of the dataset. All images are stored in this directory, categorized by subfolders (e.g., spoons, forks).
- **Categories Listing**: **categories** is an array of strings representing the names of subfolders within the root folder. Each subfolder corresponds to a category of the images (e.g., 'spoon', 'fork').
- **Image Datastore Creation**: An **imageDatastore** object (**imds**) is created to manage the image data. It automatically labels the images based on their subfolder names and includes all subfolders within the specified categories.

2. Data Splitting

- **Splitting Datastore**: The **splitEachLabel** function is used to divide the images into a training set and a validation set, with 75% of images used for training and 25% for validation. This split is done randomly to ensure the model is trained and validated on different subsets of data.

3. Bag of Features Creation

- **Training a Bag of Features**: A "bag of features" (**bag**) is created using the training set. This involves detecting keypoints in each image, extracting features from these keypoints (using an extractor like SIFT, SURF, etc.), and then clustering these features across all images to create a "visual vocabulary". The visual vocabulary represents common patterns or features in the dataset.

4. Feature Encoding and Visualization

- **Feature Vector Encoding**: The **encode** function transforms an image into a histogram of visual word occurrences based on the previously trained bag of features. This histogram serves as the feature vector for the image.
- **Histogram Plotting**: The feature vector for the first image in the datastore is plotted as a histogram. This visualizes the frequency of each visual word's occurrence in the image.

5. Classifier Training and Evaluation

- **Training the Classifier**: An image category classifier (**categoryClassifier**) is trained using the **trainImageCategoryClassifier** function. This function takes the labeled training images and the bag of features to learn how different visual words are associated with each category.
- **Evaluating on Training Set**: The classifier is evaluated on the training set using the **evaluate** function to compute the confusion matrix (**confMatrixTrain**). The diagonal of the confusion matrix represents correct classifications. Training accuracy is calculated as the average of these correct predictions.

- **Evaluating on Validation Set**: Similarly, the classifier is evaluated on the validation set to compute the validation accuracy (**validationAccuracy**). This measures how well the classifier performs on unseen data.

```
                                        PREDICTED
KNOWN           | butter_knife   cutting_knife   fork   ladle   spoon
-------------------------------------------------------------------------
butter_knife    | 0.25           0.00            0.25   0.50    0.00
cutting_knife   | 0.25           0.25            0.00   0.25    0.25
fork            | 0.00           0.00            0.75   0.25    0.00
ladle           | 0.00           0.00            0.00   1.00    0.00
spoon           | 0.00           0.00            0.00   0.00    1.00

* Average Accuracy is 0.65.

Training Accuracy: 81.82%
Validation Accuracy: 65.00%
```

The above screenshot shows the final confusion matrix and accuracy of this model.

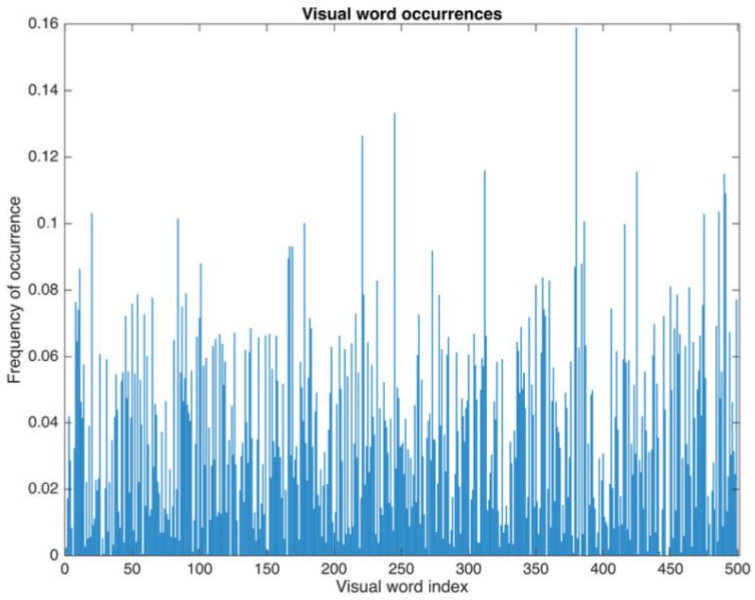
```
% Specify the root folder where your dataset is stored
rootFolder = 'dataset';  % Update this to your dataset path

% List of categories as subfolder names within the root folder
categories = {'spoon', 'fork', 'butter_knife', 'cutting_knife', 'ladle'};

% Create an imageDatastore and automatically label the images based on
folder names
imds = imageDatastore(fullfile(rootFolder, categories), 'LabelSource',
'foldernames', 'IncludeSubfolders', true);

% Split the datastore into training and validation sets
[trainingSet, validationSet] = splitEachLabel(imds, 0.75, 'randomized');

% Create a bag of features (visual vocabulary) from the training set
bag = bagOfFeatures(trainingSet);
```

```
Creating Bag-Of-Features.
-------------------------
* Image category 1: butter_knife
* Image category 2: cutting_knife
* Image category 3: fork
* Image category 4: ladle
* Image category 5: spoon
* Selecting feature point locations using the Grid method.
* Extracting SURF features from the selected feature point locations.
** The GridStep is [8 8] and the BlockWidth is [32 64 96 128].

* Extracting features from 55 images...done. Extracted 11903544 features.

* Keeping 80 percent of the strongest features from each category.

* Balancing the number of features across all image categories to improve clustering.
** Image category 1 has the least number of strongest features: 1637549.
** Using the strongest 1637549 features from each of the other image categories.

* Creating a 500 word visual vocabulary.
* Number of levels: 1
* Branching factor: 500
* Number of clustering steps: 1

* [Step 1/1] Clustering vocabulary level 1.
* Number of features          : 8187745
* Number of clusters          : 500
* Initializing cluster centers...100.00%.
* Clustering...completed 57/100 iterations (~16.40 seconds/iteration)...converged in 57 iterations.

* Finished creating Bag-Of-Features
```

```
img = readimage(imds, 1);
featureVector = encode(bag, img);
```

```
Encoding images using Bag-Of-Features.
--------------------------------------
* Encoding an image...done.
```

```
% Plot the histogram of visual word occurrences
figure
bar(featureVector)
title('Visual word occurrences')
xlabel('Visual word index')
ylabel('Frequency of occurrence')
```




```
% Train an image category classifier using the bag of features
categoryClassifier = trainImageCategoryClassifier(trainingSet, bag);
```

```
Training an image category classifier for 5 categories.
--------------------------------------------------------
* Category 1: butter_knife
* Category 2: cutting_knife
* Category 3: fork
* Category 4: ladle
* Category 5: spoon

* Encoding features for 55 images...done.

* Finished training the category classifier. Use evaluate to test the classifier on a test set.
```

```
% Evaluate the classifier using the training set (as a sanity check)
confMatrixTrain = evaluate(categoryClassifier, trainingSet);
```

```
Evaluating image category classifier for 5 categories.
-------------------------------------------------------
```

```
* Category 1: butter_knife
* Category 2: cutting_knife
* Category 3: fork
* Category 4: ladle
* Category 5: spoon

* Evaluating 55 images...done.

* Finished evaluating all the test sets.

* The confusion matrix for this test set is:


                              PREDICTED
KNOWN            | butter_knife   cutting_knife   fork   ladle   spoon
-----------------------------------------------------------------------
butter_knife     | 0.64           0.18            0.09   0.09    0.00
cutting_knife    | 0.00           0.82            0.00   0.18    0.00
fork             | 0.00           0.00            0.82   0.00    0.18
ladle            | 0.00           0.00            0.00   1.00    0.00
spoon            | 0.00           0.00            0.00   0.18    0.82

* Average Accuracy is 0.82.
```

```
trainAccuracy = mean(diag(confMatrixTrain));

% Evaluate the classifier using the validation set
confMatrixValidation = evaluate(categoryClassifier, validationSet);
```

```
Evaluating image category classifier for 5 categories.
-------------------------------------------------------

* Category 1: butter_knife
* Category 2: cutting_knife
* Category 3: fork
* Category 4: ladle
* Category 5: spoon

* Evaluating 19 images...done.

* Finished evaluating all the test sets.

* The confusion matrix for this test set is:


                              PREDICTED
KNOWN            | butter_knife   cutting_knife   fork   ladle   spoon
-----------------------------------------------------------------------
butter_knife     | 0.25           0.00            0.25   0.50    0.00
cutting_knife    | 0.25           0.25            0.00   0.25    0.25
fork             | 0.00           0.00            0.75   0.25    0.00
ladle            | 0.00           0.00            0.00   1.00    0.00
spoon            | 0.00           0.00            0.00   0.00    1.00

* Average Accuracy is 0.65.
```

```
validationAccuracy = mean(diag(confMatrixValidation));

% Display the accuracies
fprintf('Training Accuracy: %.2f%%\n', trainAccuracy * 100);
```

```
Training Accuracy: 81.82%
```

```
fprintf('Validation Accuracy: %.2f%%\n', validationAccuracy * 100);
```

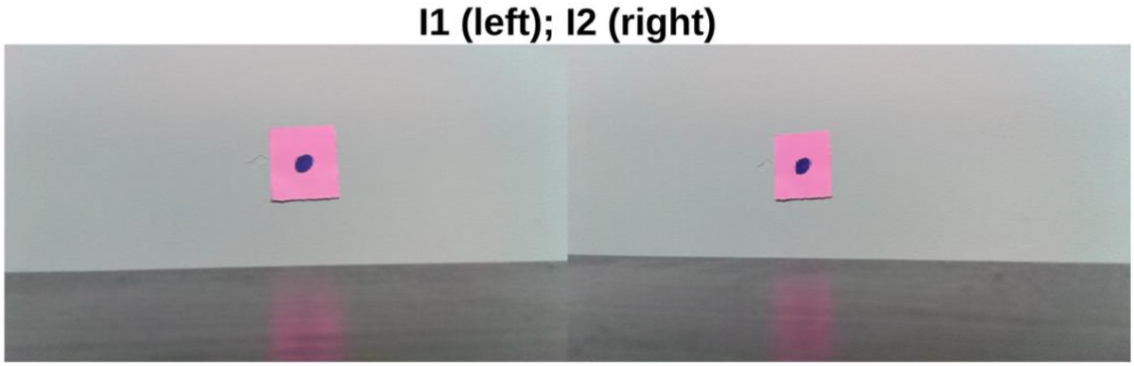
```
Validation Accuracy: 65.00%
```

A7. Disparity based depth estimation with rotation using MATLAB tutorial

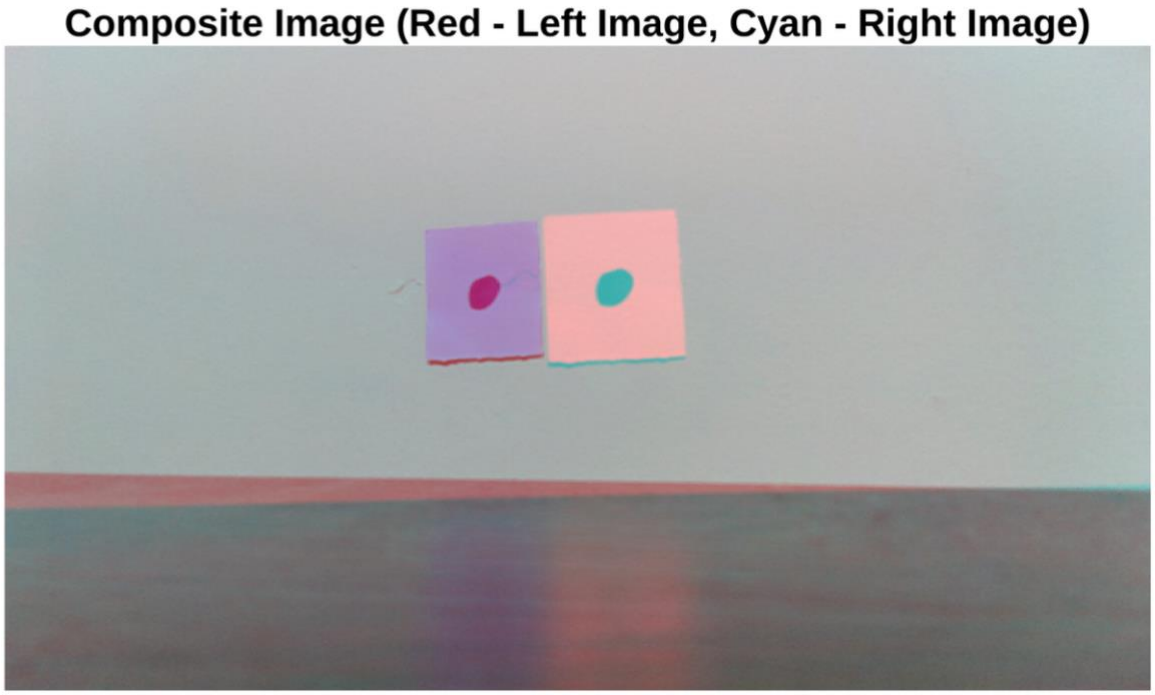
(For this question I have used the tutorial provided with some changes in it to calculate the depth from the wall to the camera. A live matlab script was used for this question, the script should be in the same directory as “image1.jpg” and “image3.jpg”)
(Matlab live script is attached at the end)

Initial Image Processing

- **Image Loading**: Two images, **I1** and **I2**, are loaded. These are assumed to be stereo images taken from slightly different viewpoints.

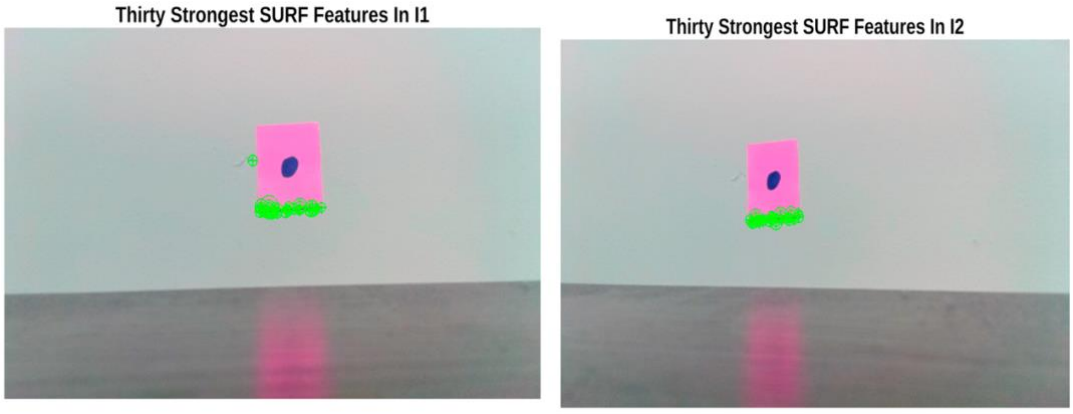


-
- **Grayscale Conversion**: The images are converted to grayscale because many feature detection algorithms perform better on single-channel data.
- **Montage and Anaglyph Creation**: The script displays the two images side-by-side in a montage and as an anaglyph (where one image is shown in red and the other in cyan) to visualize the differences and alignments in the stereo pair.

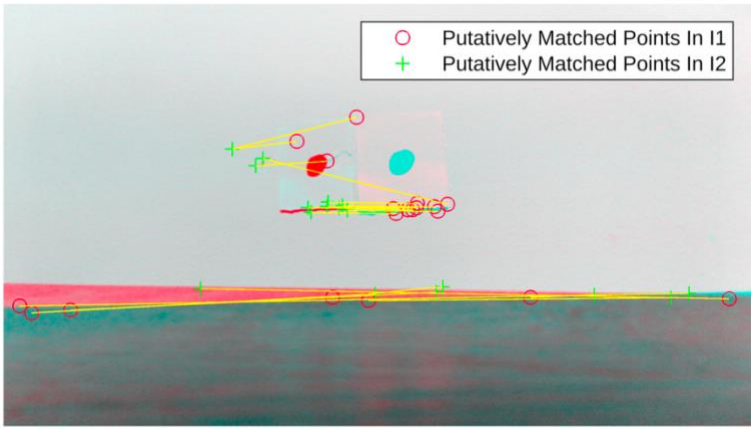


Feature Detection and Matching

- **SURF Feature Detection**: Speeded-Up Robust Features (SURF) are used to detect features in both images. SURF is effective for this purpose because it is invariant to scale and rotation changes, making it suitable for stereo images that might have perspective differences.

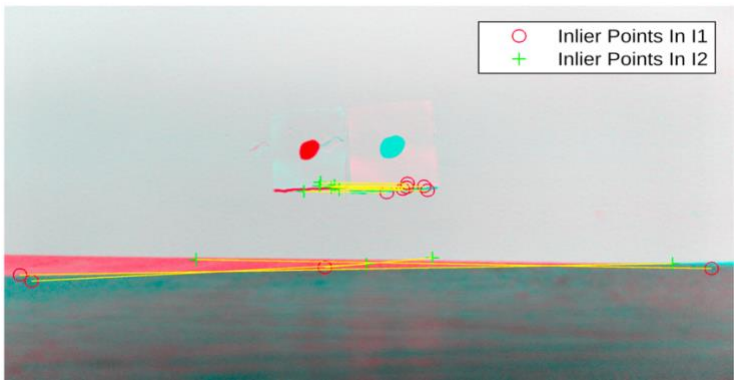


- **Feature Matching**: Features between the two images are matched using a descriptor matcher that minimizes the Sum of Absolute Differences (SAD). Matches are filtered by a threshold to retain only the best matches.



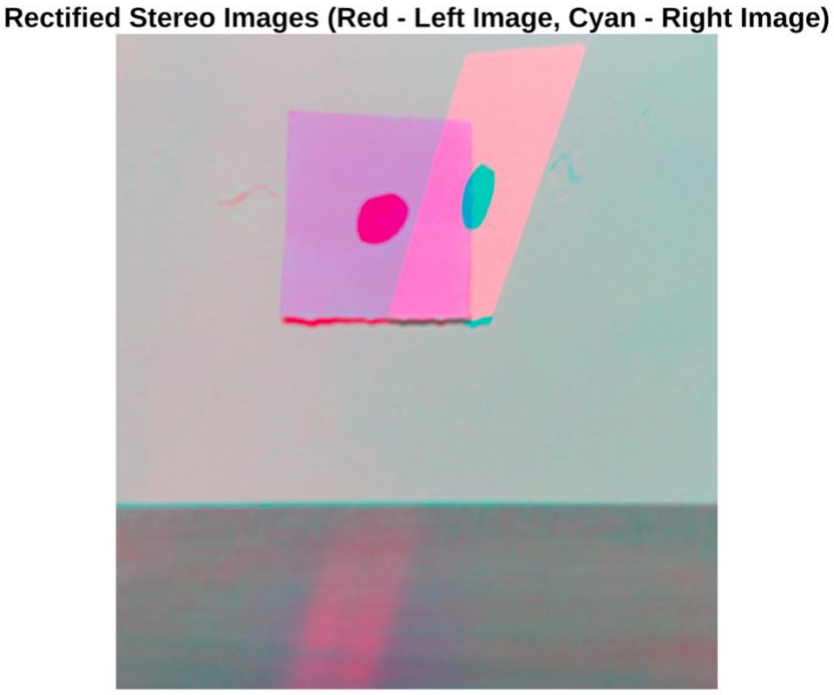
Fundamental Matrix Estimation

- **Estimate Fundamental Matrix**: The fundamental matrix, which encapsulates the epipolar geometry of the stereo pair, is estimated using the RANSAC algorithm. This step also identifies inliers (matches that fit the epipolar constraints well).
- **Error Handling**: The script checks for potential errors in fundamental matrix estimation, such as insufficient matches or the presence of epipoles within the image frame, which can complicate further calculations.



Stereo Rectification

- **Stereo Rectification**: The stereo images are rectified to align them such that corresponding points appear on the same row in both images. This simplifies the calculation of disparity.
- **Rectified Image Display**: The rectified images are displayed as an anaglyph to visualize the alignment.



Disparity Map Calculation

- **Disparity Map Calculation**: Using the rectified grayscale images, a disparity map is calculated using the Semi-Global Matching (SGM) algorithm. This map indicates the pixel displacement (disparity) between corresponding points in the two images.
- **Disparity to Depth Conversion**: The disparity values are converted to depth using the formula **(focal length * baseline) / disparity**, where the focal length is given in pixels, and the baseline is the distance between the camera centers in meters.

Depth Calculation

- **Depth at Image Center**: The depth at the center of the image is calculated using the depth map.

- **Average Depth in ROI**: The average depth within a defined region of interest (ROI) around the center of the image is also calculated.

Result Output

- **Depth Values Display**: Finally, the depth at the center and the average depth in the ROI are printed to the MATLAB console.

```
% Output the calculated depths
fprintf('Depth at the center of the image: %.2f mm\n', depthAtCenter);
```

```
Depth at the center of the image: 220.23 mm
```

```
fprintf('Average depth in the defined ROI: %.2f mm\n', averageDepthROI);
```

```
Average depth in the defined ROI: 220.36 mm
```

Uncalibrated Stereo Image Rectification

This example shows how to use the `estimateFundamentalMatrix`, `estimateStereoRectification`, and `detectSURFFeatures` functions to compute the rectification of two uncalibrated images, where the camera intrinsics are unknown.

Stereo image rectification projects images onto a common image plane in such a way that the corresponding points have the same row coordinates. This process is useful for stereo vision, because the 2-D stereo correspondence problem is reduced to a 1-D problem. As an example, stereo image rectification is often used as a preprocessing step for computing or creating anaglyph images. For more details, see the Uncalibrated Stereo Image Rectifcation example.

Step 1: Read Stereo Image Pair

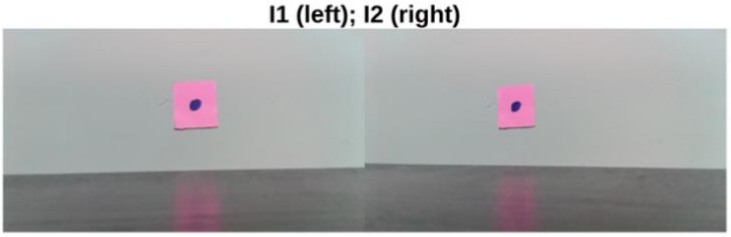
Read in two color images of the same scene, which were taken from different positions. Then, convert them to grayscale. Colors are not required for the matching process.

```
I1 = imread("image1.jpg");
I2 = imread("image3.jpg");

% Convert to grayscale.
I1gray = im2gray(I1);
I2gray = im2gray(I2);
```

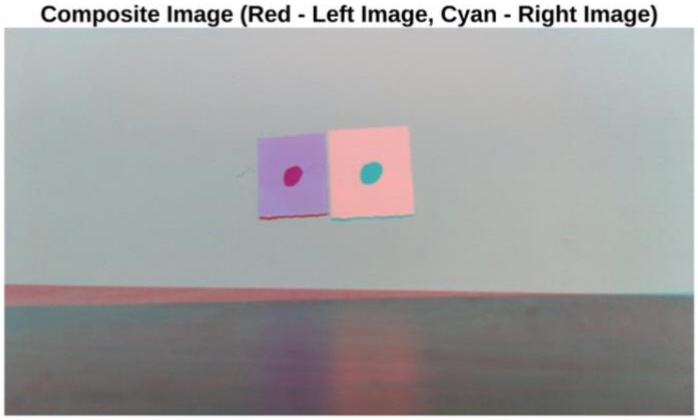
Display both images side by side. Then, display a color composite demonstrating the pixel-wise differences between the images.

```
figure
imshowpair(I1,I2,"montage")
title("I1 (left); I2 (right)")
```

I1 (left); I2 (right)



```
figure
imshow(stereoAnaglyph(I1,I2))
title("Composite Image (Red - Left Image, Cyan - Right Image)")
```



There is an obvious offset between the images in orientation and position. The goal of rectification is to transform the images, aligning them such that corresponding points will appear on the same rows in both images.

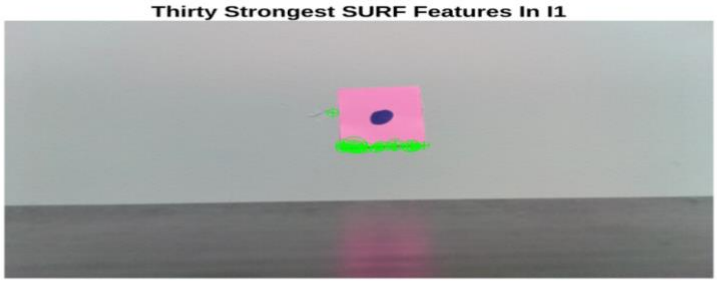
Step 2: Collect Interest Points from Each Image

The rectification process requires a set of point correspondences between the two images. To generate these correspondences, you will collect points of interest from both images, and then choose potential matches between them. Use `detectSURFFeatures` to find blob-like features in both images.

```
blobs1 = detectSURFFeatures(I1gray,MetricThreshold=100);
blobs2 = detectSURFFeatures(I2gray,MetricThreshold=100);
```

Visualize the location and scale of the thirty strongest SURF features in I1 and I2. Notice that not all of the detected features can be matched because they were either not detected in both images or because some of them were not present in one of the images due to camera motion.

```
figure
imshow(I1)
hold on
plot(selectStrongest(blobs1,30))
title("Thirty Strongest SURF Features In I1")
```




```
figure
imshow(I2)
hold on
plot(selectStrongest(blobs2,30))
title("Thirty Strongest SURF Features In I2")
```

Step 3: Find Putative Point Correspondences

Use the `extractFeatures` and `matchFeatures` functions to find putative point correspondences. For each blob, compute the SURF feature vectors (descriptors).

```
[features1,validBlobs1] = extractFeatures(I1gray,blobs1);
[features2,validBlobs2] = extractFeatures(I2gray,blobs2);
```

Use the sum of absolute differences (SAD) metric to determine indices of matching features.

```
indexPairs = matchFeatures(features1,features2,Metric="SSD", ...
  MatchThreshold=5);
```

Retrieve locations of matched points for each image.

```
matchedPoints1 = validBlobs1(indexPairs(:,1),:);
matchedPoints2 = validBlobs2(indexPairs(:,2),:);
```

```
matchedPoints1
```

```
matchedPoints1 =
  19x1 SURFPoints array with properties:

              Scale: [19x1 single]
```

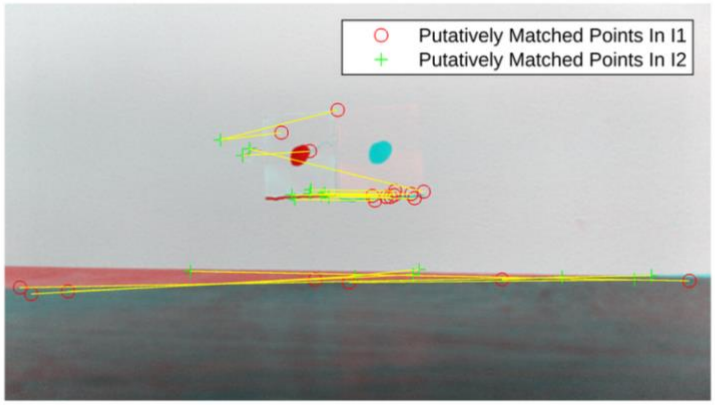
```
    SignOfLaplacian: [19x1 int8]
        Orientation: [19x1 single]
           Location: [19x2 single]
             Metric: [19x1 single]
              Count: 19
```

```
matchedPoints2
```

```
matchedPoints2 =
  19x1 SURFPoints array with properties:

              Scale: [19x1 single]
    SignOfLaplacian: [19x1 int8]
        Orientation: [19x1 single]
           Location: [19x2 single]
             Metric: [19x1 single]
              Count: 19
```

Show matching points on top of the composite image, which combines stereo images. Notice that most of the matches are correct, but there are still some outliers.

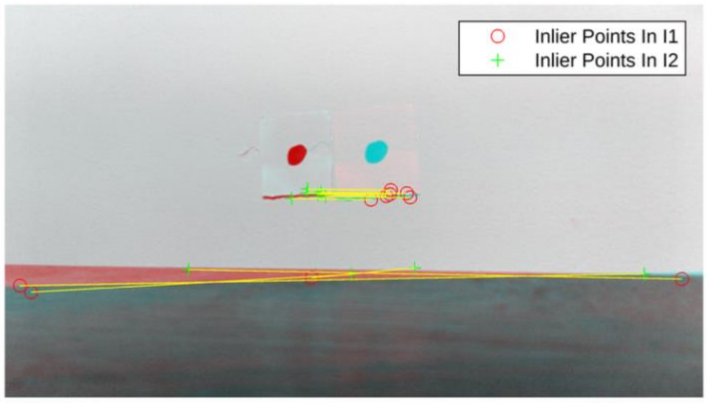
```
figure
showMatchedFeatures(I1, I2, matchedPoints1, matchedPoints2)
legend("Putatively Matched Points In I1","Putatively Matched Points In I2")
```



Step 4: Remove Outliers Using Epipolar Constraint

The correctly matched points must satisfy epipolar constraints. This means that a point must lie on the epipolar line determined by its corresponding point. You will use the `estimateFundamentalMatrix` function to compute the fundamental matrix and find the inliers that meet the epipolar constraint.

```
[fMatrix, epipolarInliers, status] = estimateFundamentalMatrix(...
  matchedPoints1,matchedPoints2,Method="RANSAC", ...
  NumTrials=200,DistanceThreshold=0.1,Confidence=99.99);



inlierPoints1 = matchedPoints1(epipolarInliers, :);
inlierPoints2 = matchedPoints2(epipolarInliers, :);

figure
showMatchedFeatures(I1, I2, inlierPoints1, inlierPoints2)
legend("Inlier Points In I1","Inlier Points In I2")
```



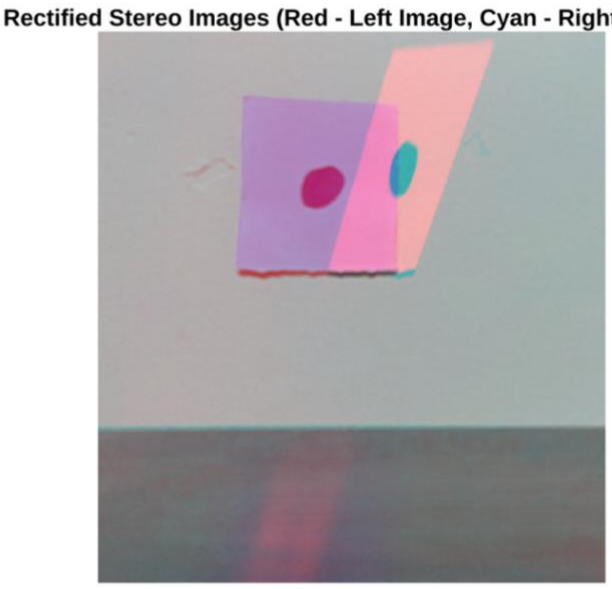
Step 5: Rectify Images

Use the `estimateStereoRectification` function to compute the rectification transformations. These can be used to transform the images, such that the corresponding points will appear on the same rows.

```
[tform1, tform2] = estimateStereoRectification(fMatrix, ...
  inlierPoints1.Location,inlierPoints2.Location,size(I2));
```

Rectify the stereo images, and display them as a stereo anaglyph. You can use red-cyan stereo glasses to see the 3D effect.

```
[I1Rect, I2Rect] = rectifyStereoImages(I1,I2,tform1,tform2);
figure
imshow(stereoAnaglyph(I1Rect,I2Rect))
title("Rectified Stereo Images (Red - Left Image, Cyan - Right Image)")
```



Step 6: Detecting depth

This code furhter calculates the depth of the wall from the camera

```
% Convert rectified images to grayscale if they are not already
I1Gray = rgb2gray(I1Rect);
I2Gray = rgb2gray(I2Rect);

% Calculate the disparity map using the Semi-Global Matching algorithm
disparityRange = [350, 358];  % This should be adjusted based on the
specific setup
disparityMap = disparitySGM(I1Gray, I2Gray, 'DisparityRange',
disparityRange, 'UniquenessThreshold', 15);

% Known camera parameters
baselineMeters = 0.052;  % The distance between the two camera centers in
meters
```

```
focalLengthPixels = 1495;  % The focal length of the cameras in pixels

% Convert disparity to depth in millimeters
disparityMap(disparityMap == 0) = 0.1;  % Small value to avoid division by
zero
depthMap = (focalLengthPixels * baselineMeters) ./ disparityMap * 1000;  %
Convert meters to millimeters

% Calculate depth at a specific point, e.g., the center of the image
centerX = size(I1Gray, 1) / 2;
centerY = size(I1Gray, 1) / 2;
depthAtCenter = depthMap(round(centerY), round(centerX));

% Optionally calculate average depth over a defined ROI, e.g., central square
roiSize = 100;  % Define the size of the ROI square
xRange = round(centerX - roiSize/2) : round(centerX + roiSize/2);
yRange = round(centerY - roiSize/2) : round(centerY + roiSize/2);
averageDepthROI = mean(depthMap(yRange, xRange), 'all', 'omitnan');

% Output the calculated depths
fprintf('Depth at the center of the image: %.2f mm\n', depthAtCenter);
```

```
Depth at the center of the image: 220.23 mm
```

```
fprintf('Average depth in the defined ROI: %.2f mm\n', averageDepthROI);
```

```
Average depth in the defined ROI: 220.36 mm
```

References

[1] Trucco, E; Verri, A. "Introductory Techniques for 3-D Computer Vision." Prentice Hall, 1998.

[2] Hartley, R; Zisserman, A. "Multiple View Geometry in Computer Vision." Cambridge University Press, 2003.

[3] Hartley, R. "In Defense of the Eight-Point Algorithm." IEEE® Transactions on Pattern Analysis and Machine Intelligence, v.19 n.6, June 1997.

[4] Fischler, MA; Bolles, RC. "Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography." Comm. Of the ACM 24, June 1981.

A8. Real-time object tracker
(Both the parts can be executed in a single IPYNB file)

8.1) Uses a marker (QR code)

1. Video Capture Initialization

- **Video Capture Object**: The script starts by creating a **VideoCapture** object **cap** with the argument **0**, which specifies that the video should be captured from the default camera.
- **Camera Check**: It checks if the camera was successfully opened. If not, it prints an error message and exits the function.

2. Processing Video Frames

- **Reading Frames**: The script enters a while-loop that continues as long as the camera is open. Within the loop, it attempts to read a frame from the video stream.
- **Frame Check**: If a frame is successfully read (**ret** is True), the script proceeds to process the frame.

3. QR Code Detection and Decoding

- **QR Code Detector**: A **QRCodeDetector** object is instantiated.
- **Detection and Decoding**: The script calls **detectAndDecode** on the frame to identify any QR codes. This method returns three values:
    - **data**: The decoded text from the QR code.
    - **bbox**: The coordinates of the bounding box around the QR code.
    - **_**: A placeholder for additional data (unused).

4. Displaying Results

- **Bounding Box Drawing**: If a QR code is detected (**bbox** is not None), the bounding box coordinates are converted to integers, and lines are drawn between the corners of the bounding box using **cv2.line**. The box is drawn in blue (**color=(255, 0, 0)**).
- **Text Display**: If **data** is available (i.e., the QR code was successfully decoded), the script displays this data as green text on the frame, positioned just above the top-left corner of the QR code.

5. Frame Display and Exit Control

- **Display Frame**: The **cv2.imshow** function displays the current frame in a window titled 'QR Code Tracker'.
- **Quit Option**: The script waits for 25 milliseconds for a key press and will exit the loop if the 'q' key is pressed (**cv2.waitKey(25) & 0xFF == ord('q')**).

6. Cleanup

- **Release and Destroy**: After exiting the loop (either through a frame read error or a 'q' key press), the video capture object is released with **cap.release()**, and all OpenCV windows are closed with **cv2.destroyAllWindows()**.

8.2)Does not uses a marker

This is implemented in detail from scratch as described below-

Step 1: Initialize Video Capture

- **Start video capture**: The script initiates capturing video from the default camera (typically the first connected webcam) by creating a **VideoCapture** object using **cv2.VideoCapture(0)**.

Step 2: Select ROI from the Video

- **Read Frames**: In a loop, the script continuously reads frames from the video capture object. This is done using **cap.read()**, which returns a Boolean indicating success (**ret**) and the captured frame (**frame**).
- **Check Frame Validity**: If a frame is successfully captured (**ret** is True), it proceeds to the next step; otherwise, it breaks the loop, indicating an issue or end of the video stream.
- **Display Frame for ROI Selection**: The captured frame is displayed, and the user is prompted to select an ROI using **cv2.selectROI("Select ROI", frame, False, False)**. This function allows the user to draw a rectangular region on the frame, which defines the ROI.

Step 3: Extract and Process the Selected ROI

- **Extract the ROI**: Using the coordinates returned from **selectROI**, the specific region of the frame is extracted. This is achieved by slicing the array representing the frame (**frame[y:y+h, x:x+w]** where **x, y, w, h** are the coordinates and dimensions of the ROI).
- **Initialize ORB Detector**: An ORB detector is created to identify and describe keypoints within the ROI. ORB (Oriented FAST and Rotated BRIEF) is chosen for its efficiency in real-time applications.
- **Detect Keypoints and Compute Descriptors**: The ORB detector processes the extracted ROI to detect keypoints and compute their descriptors, which are compact representations of the local features around each keypoint.

Step 4: Matching Features in Subsequent Frames

- **Reinitialize Video Capture**: The video capture is restarted to begin tracking the ROI in subsequent frames.
- **Continuous Frame Processing**: The script enters another loop to process each new frame captured from the video stream:
    - **Detect and Compute Descriptors**: Just like with the ROI, ORB is used to detect keypoints and compute descriptors in the entire frame.
    - **Match Descriptors**: A Brute Force matcher with Hamming distance is used to find matches between the descriptors in the ROI and those in the current frame. Matches are sorted by their distances, with lower distances indicating better matches.

Step 5: Estimating Object Motion Using Homography

- **Compute Homography**: If enough good matches (more than 10) are found, the script computes a homography matrix using **cv2.findHomography()**. This matrix describes the projective transformation required to align the ROI with its corresponding points in the new frame.
- **Transform ROI Coordinates**: Using the computed homography matrix, the corners of the ROI are transformed to their new positions in the current frame.

- **Draw Transformed ROI**: A polygon is drawn on the current frame to visually represent the tracked ROI. This polygon connects the transformed corners, showing the new position and orientation of the ROI.

Step 6: User Interface for Real-Time Tracking

- **Display Tracking**: Each transformed frame, showing the tracked ROI, is displayed in real-time using **cv2.imshow("Tracking", frame)**.
- **Exit Option**: The user can terminate the tracking by pressing 'q'. This is checked after each frame is processed (**cv2.waitKey(1) & 0xFF == ord('q')**).

Step 7: Cleanup

- **Release Resources**: After exiting the loop, the script releases the video capture object and closes all OpenCV windows to clean up resources.

Github Link - https://github.com/ruchirnamjoshi/Module-3-CV