

Computer Vision assignment 2A and 2B report

Padhye Soham Satish

M22RM007

Question 1-

You are given a scene image containing logo, and a gallery containing reference logos for 10 business brands. Find out which business brand is present in the scene. Try out three different approaches and compare them.

I have solved this problem using three different methods 1.SIFT, 2.ORB, 3.BRISK

I have uploaded 3 different code files for 3 different methods named logoMatch1.py, logoMatch2.py and logoMatch3.py. Please go through these files for question number 1.

Method 1- Using SIFT to detect the keypoints and matching the scene with given logos

Colab Link for method 1-

https://colab.research.google.com/drive/1s9VLT6OJ78LKTS4oApzrUoh4l5hv0LmW#scrollTo=3Ic3NPeuO_y1

SIFT (Scale-Invariant Feature Transform) is a computer vision algorithm used for feature extraction, matching and object recognition. The SIFT algorithm extracts key points or features from an image that are invariant to scale, orientation, and affine distortion. These key points are represented by descriptors, which are vectors of numerical values that describe the local image features around each key point. The SIFT algorithm is robust to changes in lighting conditions and is able to accurately match features between different images even when there are significant changes in viewpoint, scale, and rotation. One of the main advantages of SIFT is its robustness to variations in image appearance, making it suitable for a wide range of applications, including object recognition, image registration, and 3D reconstruction.

I'm using the SIFT to match the logos because it gives the keypoints which are invariant to scale, orientation, brightness, rotation etc. As the scene image has logo in one orientation and reference logo has different orientation, it is beneficial to use SIFT.

I have used FLANN matcher for matching the descriptors of the images. Brief algorithm for FLANN is-

1. Extract features from image 1 and image 2 using a feature detector, such as SIFT,ORB,BRISK.
2. Compute descriptors for the extracted features.
3. Build an index using the FLANN library to efficiently search for nearest neighbors in the descriptor space.
4. For each query feature descriptor, search for its nearest neighbors in the reference image using the FLANN index.
5. Apply a ratio test to filter out unreliable matches. This test compares the distance of the two nearest neighbors found in step 4, and keeps only those matches where the ratio of distances is below a certain threshold.
6. We can also apply additional geometric constraints to further filter out mismatches, such as using RANSAC to estimate a homography matrix between the query and reference images.

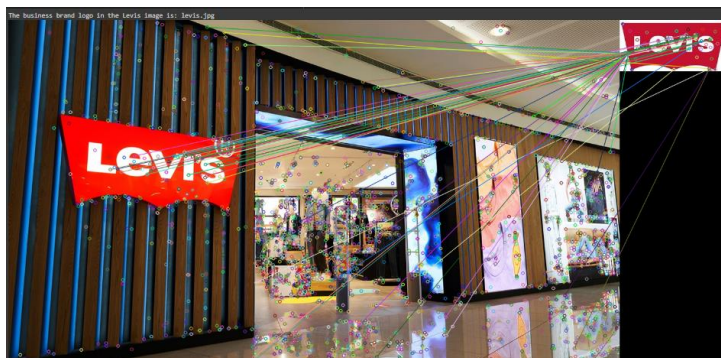
Algorithm for matching the logos with given scene-

1. Create SIFT class
 - 1.1. Define the constructor method for the LogoDetector class, which takes in two arguments: image and logo array
 - 1.2. Create SIFT object and assign to sift variable
 - 1.3. Initializes an empty list for storing the SIFT features of each reference logo
 - 1.4. Use for loop to iterate through logos
 - 1.5. Read one logo and extract the SIFT keypoints and descriptors.
 - 1.6. Append a tuple of the current reference logo file path and its SIFT descriptors to the reference_features list
 - 1.7. Use FLANN to match the descriptors, Initialize a dictionary with the algorithm key set to FLANN_INDEX_KDTREE
 - 1.8. Creates a FLANN-based matcher object using the FlannBasedMatcher function
 - 1.9. Define best match function to get the keypoints and descriptors
 - 1.9.1. Detect keypoints (kp) and extract descriptors (des) from the scene image using SIFT algorithm
 - 1.9.2. Use a FLANN (Fast Library for Approximate Nearest Neighbors) algorithm to find matching descriptors between the scene image and the logo image. The k=2 parameter specifies that we want to find the two nearest neighbors for each descriptor in the scene image
 - 1.9.3. For each pair of matching descriptors, if the distance ratio between the closest and second-closest match is less than 0.7, then add the closest match to a list of "good match."
 - 1.9.4. Compute a match score by dividing the number of good matches by the total number of keypoints in the scene image
 - 1.9.5. If the match score for the current reference logo is higher than the current best match score, update the best match score and best match logo
 - 1.9.6. Return the best match logo.
 - 1.10. Define new function draw best matches from best match and scene image

- 1.10.1. Call the `find_best_match()` function to get the filename of the best match logo image.
- 1.10.2. Load the reference image
- 1.10.3. Detect keypoints (`reference_kp`) and extract descriptors (`reference_des`) from the logo image using SIFT algorithm.
- 1.10.4. Detect keypoints (`kp`) and extract descriptors (`des`) from the scene image using SIFT algorithm.
- 1.10.5. Use FLANN to find matching descriptors between the scene image and the reference image.
- 1.10.6. For each pair of matching descriptors, if the distance ratio between the closest and second-closest match is less than 0.6, then add the closest match to a list of "good matches". Adjust the threshold to get the desired output.
- 1.10.7. Draw the good matches between the scene image and the reference image using `drawMatches()` function.
- 1.10.8. Show the result .

By using this we can match both the scenes with thrie corresponding logos

Output-



Checking the code on Custom image taken by mobile camera.

I have captured Starbucks scene image at airport and checked the algorithm for custom scene image

Output -

- It is perfectly matching the scene image even if scene image is having low lights



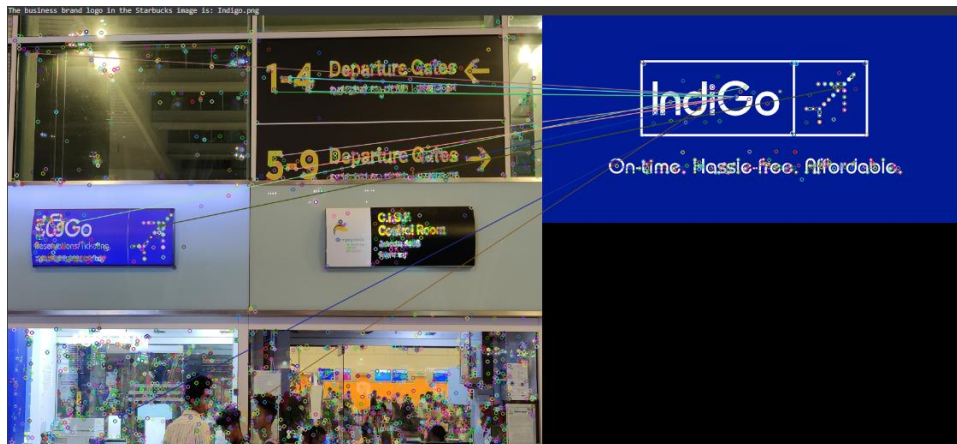
Checking one more custom scene image

Output-It is perfectly matching the logo in the scene.

Scene image-



Logo matching-



Observations-

1. First, I used the Brute Force matcher to match the descriptors. But it was not giving the good results. That may be due to
 - 1.1.Brute force matching compares each descriptor in the image 1 to every descriptor in the image 2, which can be computationally expensive and may result in many false matches.

- 1.2.Brute force matching is sensitive to changes in illumination, rotation, and scale, which can make it difficult to find good matches between images with different appearances. In our case the illumination, rotation and scale are different that's why it is not showing good results.
2. If we change the threshold value for matching the logo the result will change.
 - 2.1.If the threshold value is too high then we may get wrong result(higher likelihood of false positives) We are allowing the outliers also in this case.
 - 2.2.If the threshold value is too small then we will not get enough matches to match the logo, and logo will not be matched.But it will tend to produce fewer but more accurate matches.
 - 2.3.Changing the value of threshold will have a direct impact on the quality and number of matches found by the algorithm.
3. Changing the parameters of FLANN-
 - 3.1.The trees parameter denotes the number of trees to use in the KDTree. If we increase this value it will generally improve the accuracy of the matching, but increases memory usage and computation time.
 - 3.2.Similarly, the checks parameter in the search_params dictionary determines the number of times the algorithm will search the tree structure during the matching process. A higher number of checks will result in better accuracy but slower performance.
4. In flann.knnMatch() the value of nearest neighbours can be changed.
 - 4.1.higher value of K will increase the number of potential matches returned, but may also increase the number of false matches.
 - 4.2.lower value will decrease the number of matches returned, but may also exclude some correct matches.
 - 4.3.Changing the value of k can affect the trade-off between matching accuracy and computational efficiency.

Method 2-

Colab Link for method 2-

https://colab.research.google.com/drive/1eqaVU7wNUBIJNzXHK4gvomEudISt0bVo#scrollTo=xdS9g_M1j4rL

In this method I'm using the ORB feature detector. For matching the descriptors of the images I'm using Brute-force matcher.

The ORB (Oriented FAST and Rotated BRIEF) algorithm is a computer vision feature detection method that identifies keypoints.

The basic algorithm can be described in the following steps:

1. Detect keypoints using a variant of the FAST corner detector. The corners are then refined to increase accuracy.

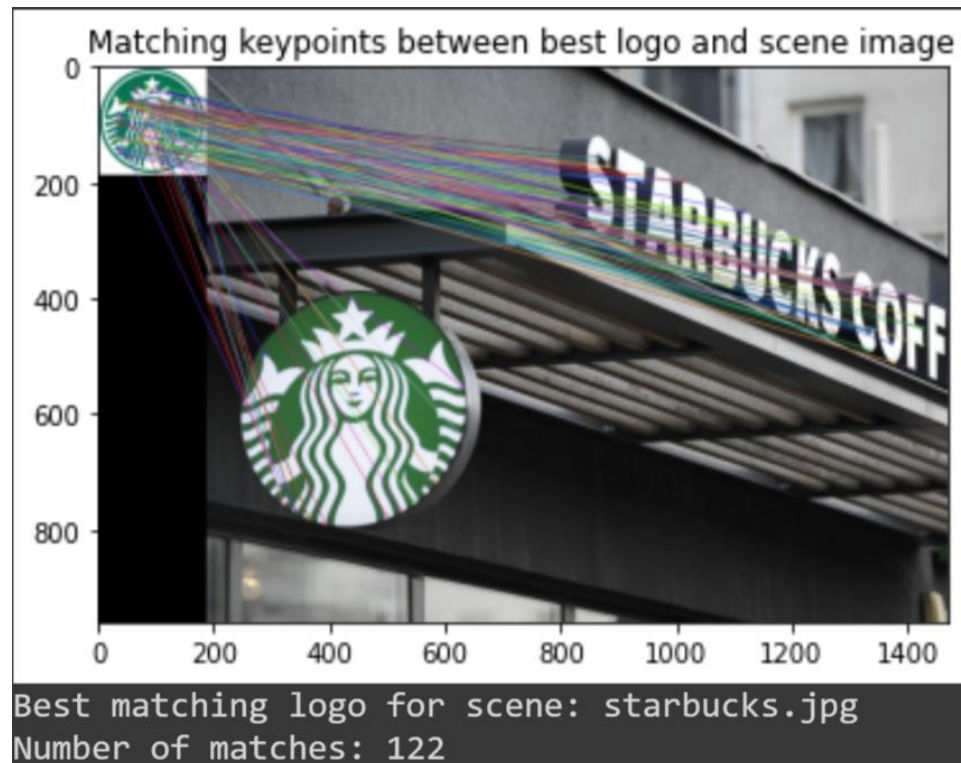
2. Computation of the intensity gradient of each keypoint's neighborhood using the modified center-symmetric algorithm.
3. Assignment of an orientation to each keypoint based on the direction of the dominant gradient.
4. Description of the neighborhood of each keypoint using the BRIEF (Binary Robust Independent Elementary Features) descriptor.
5. Creation of a binary feature vector representing each keypoint's appearance, using the BRIEF descriptor and the assigned orientation.
6. Matching of keypoints across images using the Hamming distance to compare binary feature vectors.

Ref for ORB- https://docs.opencv.org/3.4/d1/d89/tutorial_py_orb.html

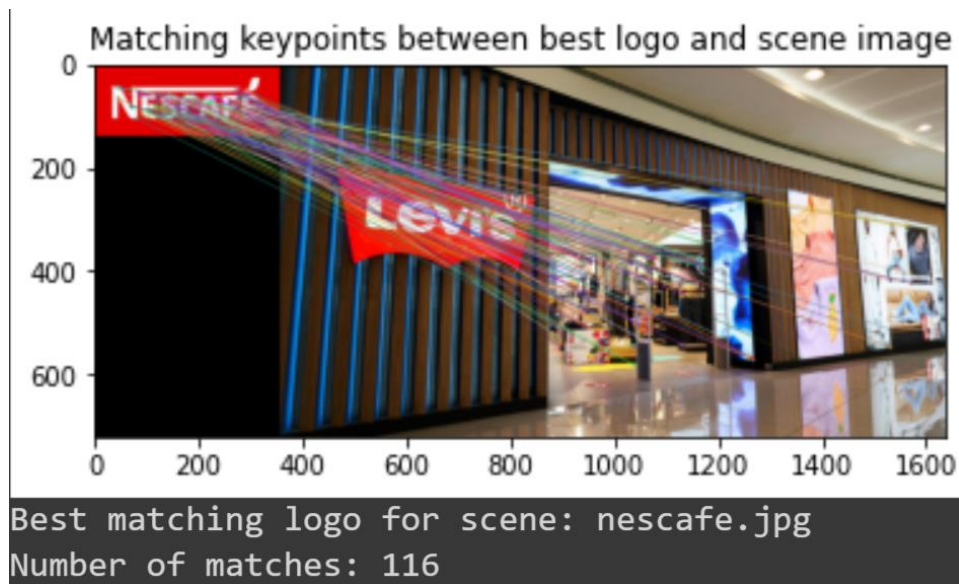
Algorithm for matching the images using ORB feature detector and Bruteforce matcher-

1. Basic algorithm for importing the images and processing the images is same as the method one, only difference is here I'm using ORB. So create ORB object using inbuilt function
2. Use BFmatcher to match descriptors. cv2.NORM_HAMMING specifies the distance measurement method used for matching and crossCheck=True enables cross-checking of matches.
3. Rest procedure is same as method 1

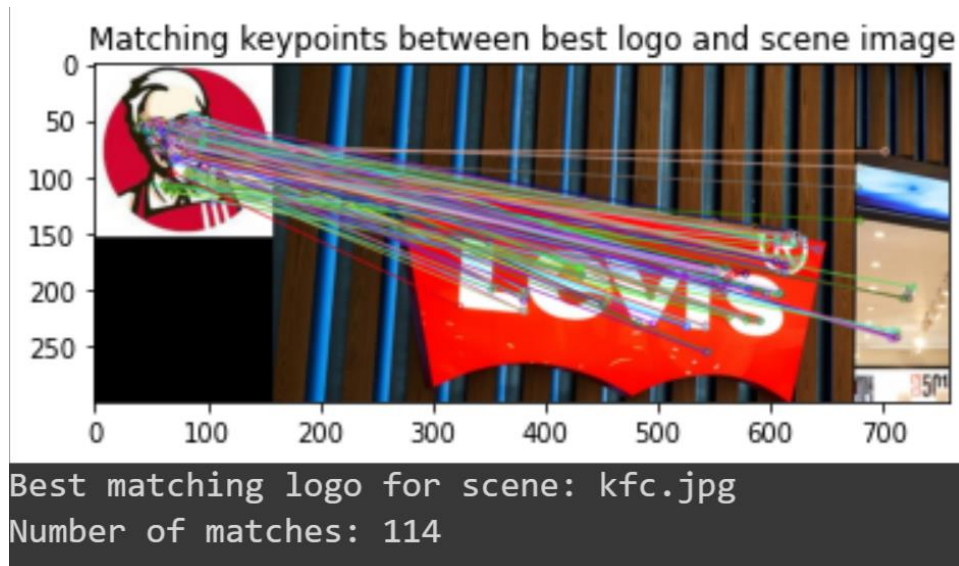
Output for ORB detector-



Without considering the Region of interest the output is -

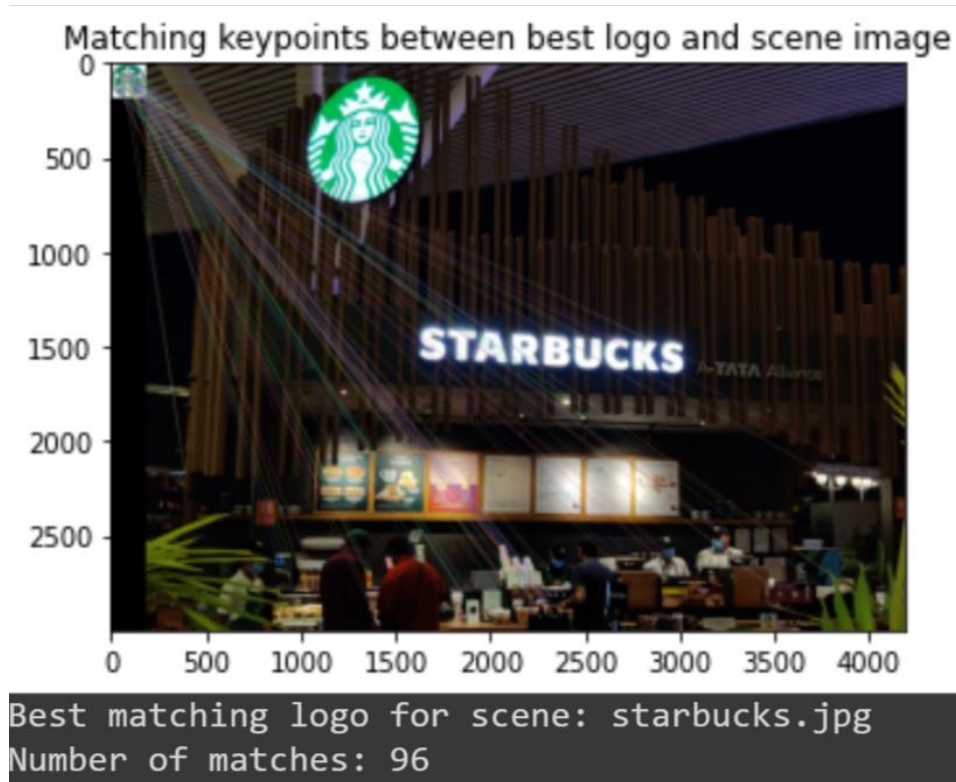


By considering the Region of interest the output is -



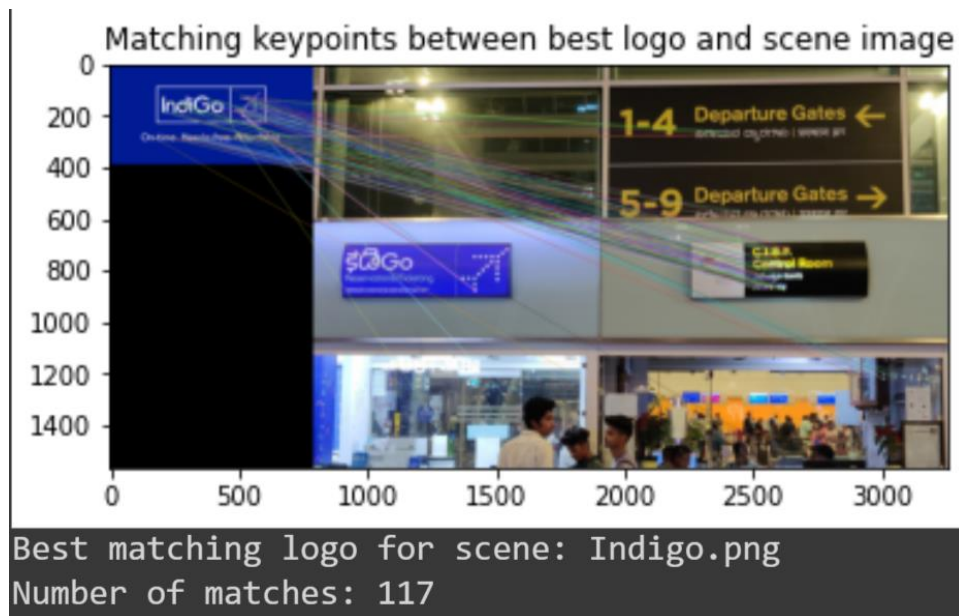
I have taken the custom image to check the algorithm accuracy.

- Output- It is perfectly detecting the logo in the given scene



Checking one more custom scene to match the logo

Output-It is perfectly matching the logo in the scene



Observations-

1. ORB can detect the starbucks logo in the scene but it is unable to detect Levi's logo in the scene.
2. I tried cropping the image by considering the region of interest (logo in the given scene) also, but it is not able to match the logo image with scene image. Reasons may be:
 - 2.1. Lighting conditions: Lighting conditions in the scene are different from the lighting conditions in the logos in our case.
 - 2.2. Scale and rotation differences: The logo in the sequence has different scales and rotations than the scene in our case.

So because of these reasons it is not able to match the levis image with scene.

Method 3-

In this method I have used BRISK algo for finding the keypoints and descriptors.

Colab code link-

https://colab.research.google.com/drive/1eqaVU7wNUBIJNzXHK4gvomEudISt0bVo#scrollTo=xdS9g_M1j4rL

Algorithm for BRISK-

1. Detect keypoints: Use the BRISK feature detector to detect keypoints in both images.
2. Compute descriptors: Compute the BRISK descriptors for each keypoint in both images.
3. Match descriptors: Match the descriptors of the keypoints between the two images using a nearest neighbor search. For each descriptor in the first image, find the closest descriptor in the second image.
4. Apply distance ratio test: Apply a distance ratio test to filter out ambiguous matches. For each match, compare the distance to the nearest neighbor descriptor with the distance to the second nearest neighbor descriptor. If the ratio between the two distances is below a threshold, consider the match valid.
5. Apply geometric verification: Apply a geometric verification step to further filter out false matches. Use the RANSAC algorithm to estimate the homography between the two images based on the valid matches. Then, use the homography to warp one image to align with the other and count the number of inliers. If the number of inliers is above a threshold, consider the match valid.
6. Return matched keypoints: Return the keypoints and their matches between the two images.

Reference - <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6126542>

Algorithm for BRISK feature points and descriptor matching-

1. Procedure is same as first two methods except here we are using the BRISK to detect keypoints
2. Use inbuilt command to detect the keypoints and descriptors
3. Rest procedure is same as method one and two
4. First I have used Brute-Force matcher and matched the descriptors which was not giving good results then I used FLANN with Region of interest cropped image which gave very good results.

Output for BFMatcher and BRISK-

I tried the concept of Region of Interest and cropped the image.

But still, it is not able to match the scene with the logo image.

It does not match the logo perfectly.





Logo is not matching

- Then I used BRISK with FLANN for descriptor matching. Still it is not able to give good results.

Output-



- Then I used Region of interest to crop the image. Finally it is giving **very good results**

Output-



Observations-

1. BF matcher is not giving satisfactory results. Reasons for this are already mentioned in the report.
2. When I didn't consider the region of interest then the result was not good, but when I crop the image to ROI then it is giving very good matching results. Reasons may be-
 - 2.1. Eliminating Irrelevant Information: When we crop the image to ROI, we remove all the irrelevant information that is present outside of the ROI. We reduce the amount of noise and increase the signal-to-noise ratio, which can result in more accurate feature matching.
 - 2.2. Reducing Computational Complexity: When we crop the image to ROI, we reduce the amount of data that needs to be processed. This can result in faster feature extraction and matching algorithms.

- 2.3.Improving Robustness to Image Deformations: When the ROI is cropped, it becomes easier to handle image deformations such as rotation, translation, or scale changes. Hence it will produce good results.

Question 2-

Implement Hough Transform for line detection from scratch. Compare the result of openCV implementation vs your implementation (both speed and performance wise) on picture of your choice.

Colab link-

<https://colab.research.google.com/drive/1QzI47xlZ8EJJkAXG23kMXI35hahZ9rMW#scrollTo=-oxReq1UVCYZ>

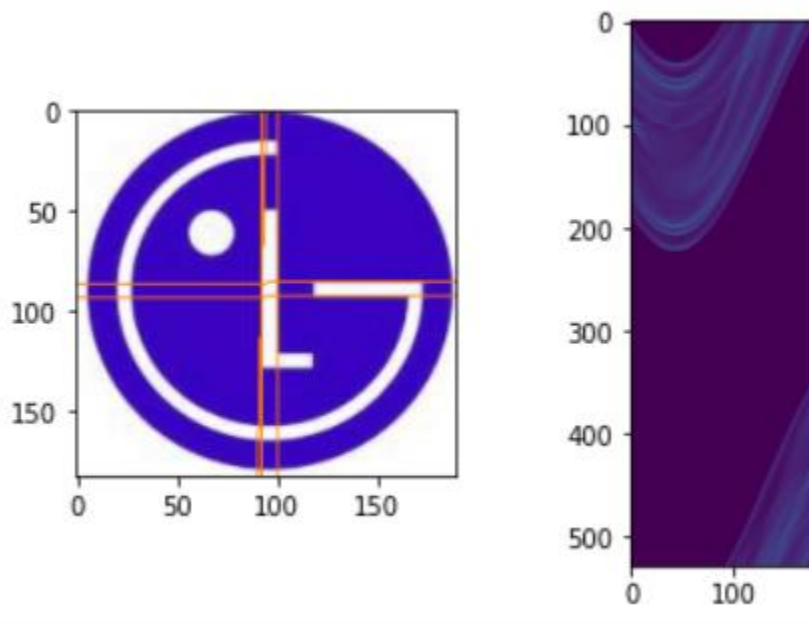
Algorithm for creating the custom class for HoughLine detection-

1. Create the class and import image
 - 1.1. Define new function to find hough line.
 - 1.2. Create the array of values from 0-180 degrees
 - 1.3. Calculate the cos theta and sine theta in advance to reduce the time
 - 1.4. Generate the accumulator matrix to store the values that are required in HoughLines Voting. Line which will get the highest vote will be considered as a line. We can adjust the threshold to get as many lines we want
 - 1.5. Rho is distance of the line from the origin and theta is the inclination of the line. By using the polar coordinate systems, we can handle the case where the line is vertical.
 - 1.6. Calculate the maximum possible value of rho and create an accumulator array to store the votes for each pair of (rho, theta) values. The rho_range variable is computed using the Pythagorean theorem with the height and width of the input image.
 - 1.7. Find the locations of edge pixels in the input image and store them as tuples of (row, column) coordinates.
 - 1.8. Find the white pixels in the image that might be edge. np.where function is used that returns a tuple of arrays representing the row and column indices of the nonzero (white) pixels in the binary image. Then create a list of tuples by combining the two arrays into pairs of (row, column) coordinates.
 - 1.9. For each pixel and angle pair, the corresponding value of rho is calculated using the equation $\rho = x \cdot \cos(\theta) + y \cdot \sin(\theta)$. The round function is used to round the value of rho to the nearest integer. The accumulator array is then updated by adding 2 votes to the corresponding (rho, theta) pair. The value of 2 is used instead of 1 for better visualization of the results.

- 1.10. Finally return the accumulator array.
2. Define the function for drawing the detected hough lines on the image
 - 2.1. Load the image, convert to gray scale and apply canny edge detector by setting some threshold.
 - 2.2. Call the hough_line method and pass the canny edge detected image to it.
 - 2.3. Store the value of accumulator in variable.
 - 2.4. Now the main step is to apply some threshold to the accumulator to get the coordinates of the line. If the threshold is too high then the number of lines detected will be less and if the threshold is less then many lines will be detected.
 - 2.5. Collect the coordinates of all point which are above the threshold value.
 - 2.6. Now using simple for loop fetch the coordinates from the coordinates array and draw the line on the image using Line method in opencv. Use the polar coordinate form of line equation to calculate the x_1, y_1 and x_2, y_2 coordinates of the line.
 - 2.7. Finally show the hough line detected image using matplotlib.
3. Start the timer and call the functions to get the hough lines.
4. Stop the timer and calculate the time required to run the code.
5. Now use the inbuilt HoughLines method to detect and draw the houghLines on the image
6. Calculate the time required for running inbuilt function also.
7. Finally compare the result images and calculate the time difference between the implementation of both the methods.

Output-

- Custom HoughLine class



- Custom HoughLine class

```
total_time1=end_time-start_time
print(f"Time required to run the program is {total_time1} seconds")
```

Time required to run the program is 6.591020822525024 seconds

- Inbuilt HoughLines method



- Inbuilt HoughLines method

```
total_time2 = end_time - start_time
print(f"Time required to run the program: {total_time2} seconds")
```

Time required to run the program: 0.012407541275024414 seconds

Observations-

1. Time required for computing and drawing the houghLines is high in case of custom HoughLines class as compared with the inbuilt function. This might be due to
 - 1.1.Algorithm complexity: The custom implementation that is written by me may have a higher algorithmic complexity than the inbuilt method.
 - 1.2.Optimizations: Inbuilt methods may have been optimized and fine-tuned for performance over many iterations.
 - 1.3.Data structures: The data structures used for the implementation can also have an impact on the execution time.
 - 1.4.Language performance.

2. In case of my implementation of the houghLines the threshold value is 90 whereas in case of inbuilt function threshold value is 50 for detecting and drawing the same lines for same image. This might be due to-
 - 2.1. Differences in the voting scheme: The custom implementation and the inbuilt function may be using different voting schemes.
 - 2.2. Differences in the image pre-processing: The custom implementation and the inbuilt function may use different image pre-processing techniques.
 - 2.3. Differences in the line drawing: The custom implementation and the inbuilt function may use different algorithms for drawing the detected lines, which may affect the number of valid lines detected.
3. In case of my custom class for Hough lines flexibility is more. My class can be modified and tailored to specific needs or use cases, which gives more flexibility in the voting process or parameter tuning. Pre-built functions may not provide as much flexibility in the algorithm's implementation.
4. Accuracy: The accuracy of line detection may vary between a custom implementation and a pre-built function depending on the implementation details, such as the choice of parameters, edge detection method, etc. For some specific cases the custom implementation may give good results while for general purpose will give more accuracy on inbuilt function.

Question 3-

A manufacturing company in Bangalore, came up with the following problem statement: They have one reference design image for a part of equipment and a probe image of either faulty or perfect. You need to identify faulty image and show the defective region.

Colab Link - https://colab.research.google.com/drive/1OnrfoA2TkQya-NORGRO9_hBd-H80sVL0#scrollTo=aaOODFwG2NJt

In this question I have used different methods to find the image whether it is faulty or not:

1. Image registration with the help of SIFT to find the initial correspondence
2. Image registration with the help of ORB to find the initial correspondence
3. Using ORB feature points extractor to find the keypoints and showing the non-matching keypoints of the image. If the image is perfect then nearly all keypoints will match and it will not show any difference on contrary if the image is faulty then there will be feature points corresponding to the faulty area will not match with the perfect image. I have shown these non matching keypoints by rectangle on faulty image
4. Using SIFT to find the keypoints and showing non matching keypoints for faulty image same as 3rd method.

Algorithm for method 1-

1. Create SIFT object.
2. Compute keypoints and descriptor.
3. Use FLANN matcher to match the descriptors.
4. Calculate the homography matrix using inbuilt function.
5. Apply the homography matrix to the input image
6. Display the registered image.

Image registration is the process of aligning two or more images of the same scene taken at different times, from different viewpoints, or using different sensors. The goal is to create a single composite image that incorporates the best features of each of the original images. The algorithm for image registration is mentioned above.

Detailed description of SIFT,FLANN is already mentioned in the report.

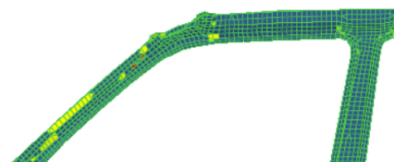
In algorithm 2 I have used the ORB feature detector and rest algorithm is same as SIFT algorithm that is mentioned above.

In algorithm 3 I have used a different approach to find the difference between faulty image and reference image that is already mentioned above. The algorithm for the code is same as that of ORB feature matching that is performed in question 1. The only difference is I'm finding good matches and sorting it in increasing order. I'm capturing the not matching feature points in the image and drawing the rectangle around it to show the difference.

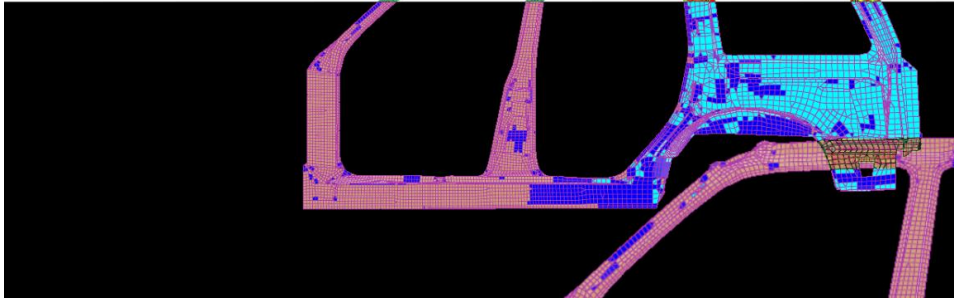
In the fourth algorithm I have used SIFT for finding the features and rest procedure is same.

Output for Image registration and SIFT for finding keypoints.

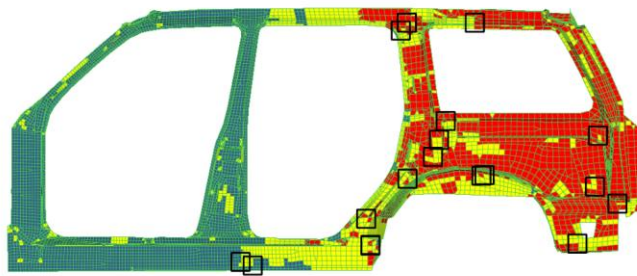
- Registered Image-



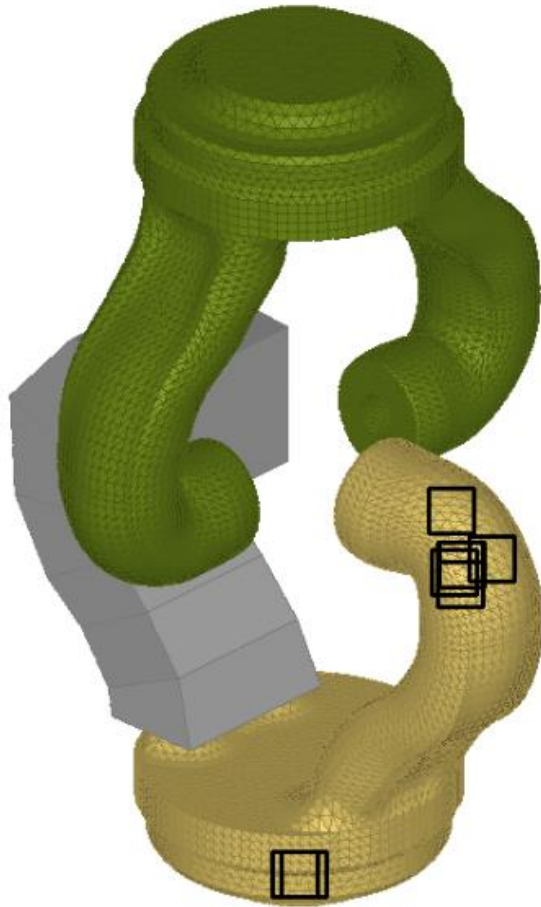
- Difference between registered image and reference image



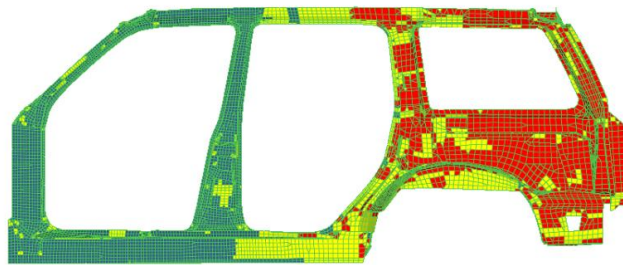
- Rectangle in the below image shows defects in the image with respect to reference image

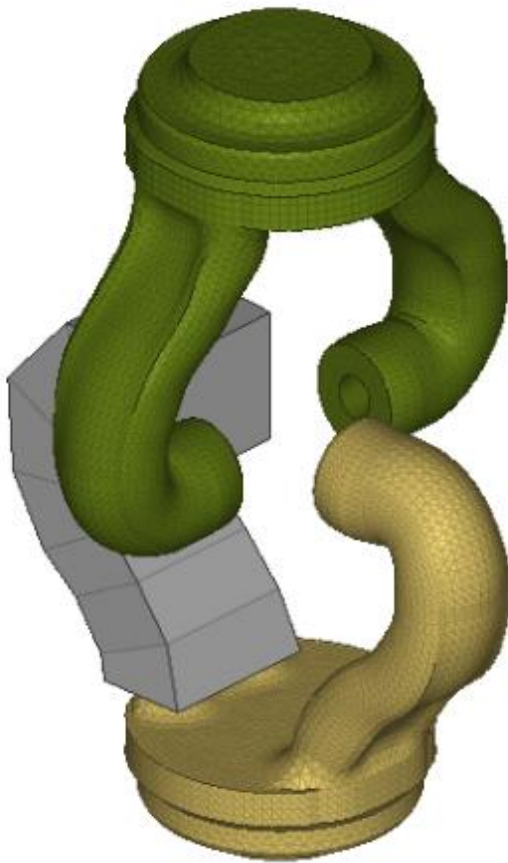


- Rectangle shows defect in the faulty image.

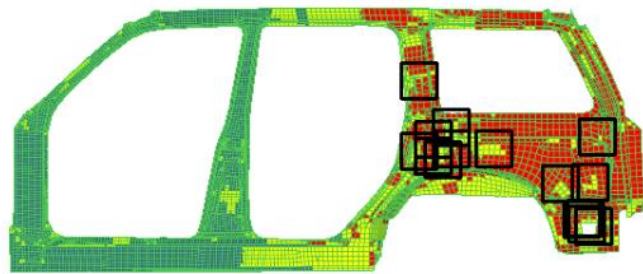


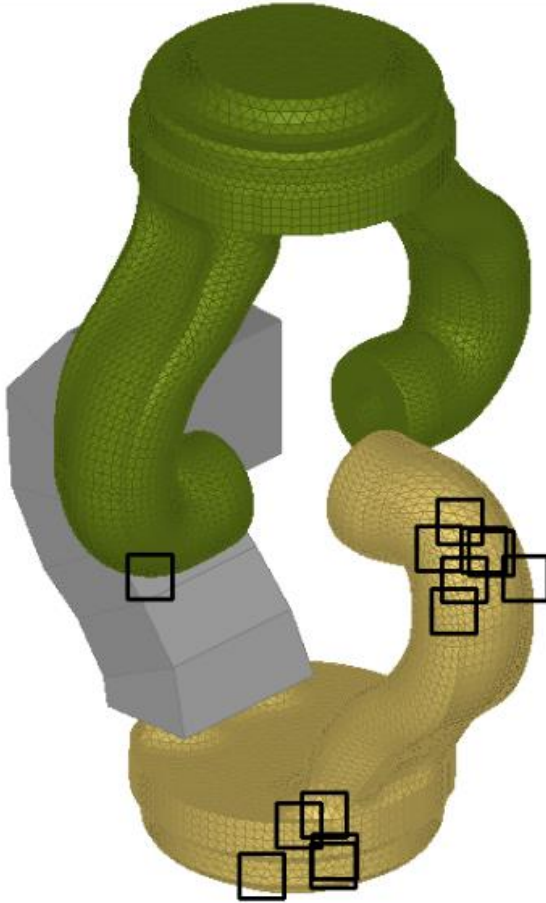
- Original images given to us are





- Perfect images given to us are also showing the difference between the image and reference





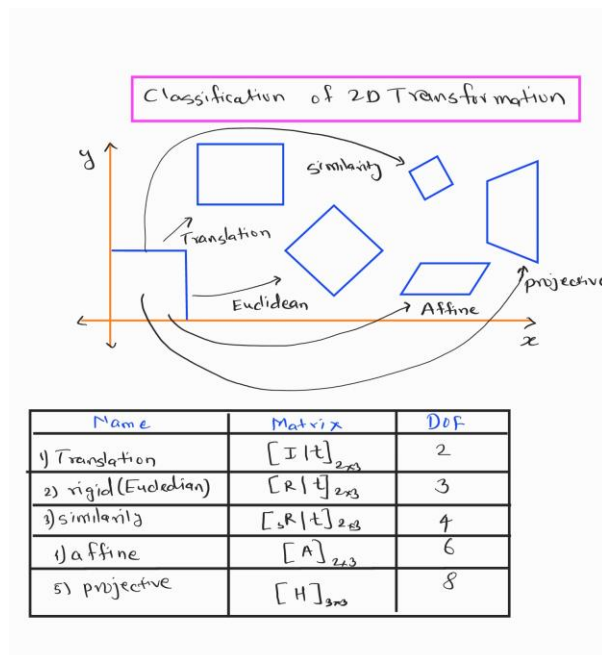
This algorithm can detect the faults in the image perfectly. But it is showing faults in the perfect image also. This might be due to-

1.Outliers- SIFT/ORB have detected the outlier keypoints and it is trying to match it with the image.

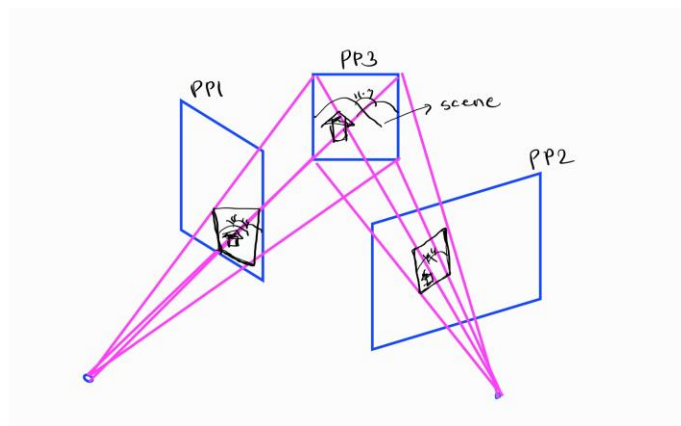
Question 4-

A Homography Matrix is a transformation matrix that maps points in one plane to corresponding points in another plane. It is also known as a perspective transformation. It is commonly used applications like object recognition, picture stitching, and camera calibration frequently employ it in computer vision and image processing.

Classification of 2D Transformation-



We can use Homography transformation to warp projective plane 1 into projective plane 2 as shown in the figure below



We can use the homography when

1. The scene is planar
2. The scene is very far or has small (relative) depth variation
3. The scene is captured under camera rotation only (no translation or pose change)

Step 1: Step 1: Select Matching points in Both Pictures

To compute the Homography Matrix, we must choose corresponding points in both the source and destination pictures. These points should be clearly visible in both images, and the

estimating procedure needs at least four matching points. Since collinearity might result in unstable solutions, the points should be chosen such that they are not close together.

Step 2: Estimate the Homography Matrix

To calculate the homography matrix, use methods like least squares or direct linear transformation (DLT). The process for DLT is

1. Create Linear Equations: We should create two linear equations of the following form for each pair of related points (x_i, y_i) in the source picture and (x'_i, y'_i) in the destination image.
2. Build a Matrix: Stack these equations for all the corresponding points to form a $2N \times 9$ matrix A .
3. Solve Linear Equations: Solve the system of linear equations $Ax = 0$ using Singular Value Decomposition (SVD) to obtain a 9×1 vector h (It is flattened)
4. Obtain Homography Matrix: Reshape h into a 3×3 matrix H .

Step 3: Normalize the Homography Matrix

There may be some numerical instability in the homography matrix that was produced in step two. By using a normalising step, we may increase the matrix's stability. The normalization step scales and translates the Homography Matrix to the origin so that the average distance from the origin is $\sqrt{2}$. Here are the steps to normalize the Homography Matrix:

1. Determine the centroid of the source image points and the destination image points.
2. Calculate the average distance of the source image points and the destination image points from their respective centroids.
3. Scale the Homography Matrix by a factor of $\sqrt{2}$ divided by the average distance computed in step 2.
4. Translate the Homography Matrix so that the centroid of the destination image points is at the origin.
5. Obtain the normalized Homography Matrix by multiplying the translation and scaling matrices with the Homography Matrix obtained in Step 2.

Step 4: Verify the Homography Matrix

We can verify the correctness of the Homography Matrix by applying it to the source image points and comparing the result with the corresponding points in the destination image. The error between the transformed points and the corresponding points in the destination image should be minimized. The steps to verify the Homography Matrix are as follows:

1. Apply the Homography Matrix to the source image points to obtain the transformed points in the destination image.
2. Compute the distance between the transformed points and the corresponding points in the destination image using a distance metric like Euclidean distance.
3. If the error is high, refine the corresponding points and repeat the process until the error is minimized.
4. Once the error is minimized, the Homography Matrix is accurate and can be used for image processing

This is the basic algorithm for calculating the homography matrix.

But in the first step we are calculating the correspondence so that we can map the points between source points and destination points in the images. For this we can use SIFT. Find the keypoints, find descriptors and then match the keypoints to get correspondence.

But all correspondences are not good. We need to filter out good correspondences to avoid bad matches. For this purpose we use RANSAC.

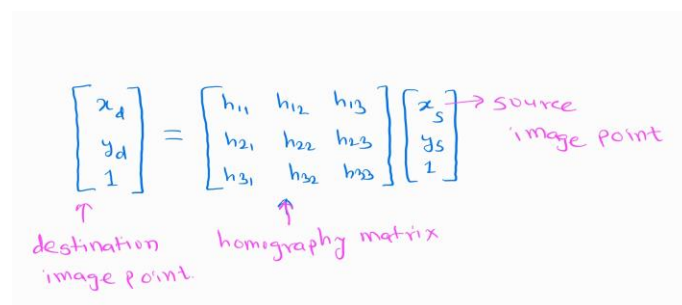
Estimating homography using RANSAC

1. Run the RANSAC loop
 - 1.1. Get four point correspondence randomly
 - 1.2. Compute Homography matrix H using selected correspondence using DLT
 - 1.3. Count inliers
 - 1.4. Keep H if largest number of inliers
2. Recompute H using all inliers

Mathematical equation formulation

Suppose X_s and Y_s are the point in the source image and X_d and Y_d is the point in the destination image, we convert them to homogeneous coordinate system to solve the equations easily.

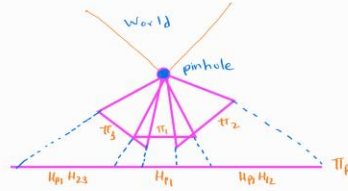
Apply the homography transformation on the source image and get the destination image. The equation is as follows


$$\begin{bmatrix} x_d \\ y_d \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x_s \\ y_s \\ 1 \end{bmatrix}$$

The image shows the equation above with handwritten annotations in pink. An arrow points from the text 'destination image point.' to the vector $\begin{bmatrix} x_d \\ y_d \\ 1 \end{bmatrix}$. Another arrow points from the text 'homography matrix' to the matrix $\begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix}$. A third arrow points from the text 'source image point' to the vector $\begin{bmatrix} x_s \\ y_s \\ 1 \end{bmatrix}$.

To get the values of $h_{11}, h_{12}, \dots, h_{33}$ use DLT technique in algebra.

Detailed mathematical expression for homography is given below-



We have 3 orientation of the cameras 1,2,3. π_1, π_2, π_3 are the planes on which image is formed.

HP_i = Homography from plane 1 to plane P.

$$\begin{bmatrix} x_d \\ y_d \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x_s \\ y_s \\ 1 \end{bmatrix}$$

$$x_d^i = \frac{h_{11} x_s^i + h_{12} y_s^i + h_{13}}{h_{31} x_s^i + h_{32} y_s^i + h_{33}} \quad \& \quad y_d^i = \frac{h_{21} x_s^i + h_{22} y_s^i + h_{23}}{h_{31} x_s^i + h_{32} y_s^i + h_{33}}$$

$$x_d^i [h_{31} x_s^i + h_{32} y_s^i + h_{33}] = h_{11} x_s^i + h_{12} y_s^i + h_{13}$$

$$y_d^i [h_{31} x_s^i + h_{32} y_s^i + h_{33}] = h_{21} x_s^i + h_{22} y_s^i + h_{23}$$

$$h_{11} x_s^i + h_{12} y_s^i + h_{13} - h_{31} x_d^i x_s^i - h_{32} x_d^i y_s^i - h_{33} x_d^i = 0$$

$$h_{21} x_s^i + h_{22} y_s^i + h_{23} - h_{31} y_d^i x_s^i - h_{32} y_d^i y_s^i - h_{33} y_d^i = 0$$

Converting the above equations to matrix form:-

$$\begin{bmatrix} x_s^i & y_s^i & 1 & 0 & 0 & 0 & -x_s^i x_d^i & -y_s^i x_d^i & -x_d^i \\ 0 & 0 & 0 & x_s^i & y_s^i & 1 & -x_s^i y_d^i & -y_s^i y_d^i & -y_d^i \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

This matrix equation is of the form $Ah=0$ s.t $\|h\|^T = 1$

Now we will define the least square problem

$$\min_h \|Ah\|^2 \quad \text{s.t. } \|h\|^2 = 1$$

$$\rightarrow \|Ah\|^2 = (Ah)^T(Ah) = h^T A^T A h \quad / \quad \boxed{h^T h = 1} \quad \text{orthonormal}$$

considering it as a loss function with penalty

$$L(h, \lambda) = h^T A^T A h - \lambda (h^T h - 1)$$

$$\frac{dL}{dh} = 0$$

$$2 A^T A h - 2 \lambda h = 0$$

$$\boxed{(A^T A) h = \lambda h}$$

Eigen vector h correspond to smallest eigen value λ of matrix $(A^T A)$ will minimize the loss $L(h, \lambda)$.

Question 5-

Stereo matching is a technique used in computer vision to obtain a 3D representation of a scene from multiple 2D images captured from different viewpoints. The main concept is to determine the disparity or difference in their positions by comparing the matching pixels in each picture. The distance between the object and the camera may be calculated using the disparity information.

Image rectification, feature extraction, correspondence matching, and disparity calculation are the four primary processes that stereo matching algorithms usually take. To make the process of matching correspondences between the two pictures easier, image rectification converts the two images into a single coordinate system. Identifying distinguishing points in the images, such edges or corners, that may be utilised to find matching points between the two images is the process of feature extraction. A few methods, like template matching, feature-based matching, and graph-based matching, are used to find the corresponding points in each image in correspondence matching. Finally, a depth map or a 3D reconstruction of the scene is produced using the disparity—or difference in position—between the corresponding points.

Depending on the complexity of the environment, the image quality, and the particular application, several stereo matching methods can be used. Some of the challenges associated with stereo matching include occlusion, where objects are partially or completely hidden from one of the cameras, and noise or inconsistencies in the images, which can lead to errors in correspondence matching. To overcome these difficulties, a number of methods have been developed, including multi-view stereo, which reconstructs a scene using more than two images, and deep learning-based approaches, which employ neural networks to train the correspondence matching algorithm.

Mathematical expressions for stereo matching is-

Computing 2D to 3D Outgoing ray
(Stereo)

3D to 2D (point) $u = f_x \frac{x}{Z} + o_x$

2D to 3D (Ray) $x = \frac{Z}{f_x} (u - o_x), y = \frac{Z}{f_y} (v - o_y), \boxed{Z > 0} \xrightarrow{\text{img}}$

we use triangulation using two cameras to get the location of 3D point.

from perspective projection:-

$$(u_l, v_l) = \left[f_x \frac{x}{Z} + o_x, f_y \frac{y}{Z} + o_y \right], (u_r, v_r) = \left[f_x \frac{(x-b)}{Z} + o_x, f_y \frac{y}{Z} + o_y \right]$$

Solving for (x, y, Z) :

$$x = \frac{b(u_l - o_r)}{(u_l - u_r)}, y = \frac{b f_x (v_l - o_y)}{f_y (u_l - u_r)}, \boxed{Z = \frac{b f_x}{(u_l - u_r)}} \text{ here } Z \text{ is the depth of the object (point)}$$

where $(u_l - u_r)$ is called Disparity.
Depth Z is inversely proportional to Disparity.
Disparity is proportional to the baseline

we can find the correspondence betⁿ left and right image point by template matching. We have shifted the image in horizontal direction. We will match the template of small window and find correspondence.

Applications of Stereo

1. **Autonomous Vehicles-** For object identification and distance estimate, autonomous vehicles use stereo image matching. The system can accurately detect obstacles and determine their distance by analysing the stereo images captured by the cameras mounted on the vehicle stereo matching is essential for reliable and efficient navigation.
 2. **3D mapping-** Stereo image matching is used in 3D mapping applications to create accurate 3D models of the environment. The technology can precisely estimate the location and depth of objects in the scene by analysing the stereo images taken by cameras from various angles, enabling the creation of comprehensive 3D maps.
 3. **Robotics-** Robotics uses stereo image matching to detect objects and grab it. The system can precisely detect objects and determine their size and form by examining the stereo pictures taken by the robot's cameras, which is crucial for effective grabbing and manipulation.
 4. **Medical imaging-** In medical imaging, stereo image matching is used to rebuild organs and tissues in three dimensions. The device can produce precise 3D models of internal organs and tissues by evaluating stereo images taken from various angles, allowing doctors to understand better and diagnose medical disorders.
 5. **Augmented reality-** In augmented reality applications, stereo image matching is utilised to build immersive and realistic virtual worlds. The system can precisely follow the user's motions and place virtual items in the environment by evaluating the stereo pictures taken by the user's device. This results in an interactive and immersive experience.
- Detailed application of stereo in autonomous vehicle is mentioned below

Object detection- Stereo vision technology is used for object detection. This includes detecting obstacles, other cars, and pedestrians. The system can precisely detect and locate things in 3D space by examining the stereo pictures that were captured by the cameras. This allows the autonomous car to move safely and avoid collisions.

Distance Estimation: Stereo vision technology is utilised to determine how far away other things in the surroundings are from the autonomous vehicle. The technology can precisely determine the depth of things in 3D space by evaluating the stereo pictures that were captured by the cameras, allowing the car to maintain a safe following distance from other vehicles and objects.

Lane Departure Warning: Stereo vision technology is used to detect lane markings on the road and alert the driver or autonomous vehicle if it deviates from its lane. By analyzing the stereo images captured by the cameras, the system can accurately detect the position of lane markings in 3D space, enabling the vehicle to maintain its lane and avoid accidents.

Traffic Sign Recognition: The technique of stereo vision is used to identify traffic signs, such as yield signs, stop signs, and speed limit signs. The technology can precisely identify and distinguish the contour and colour of traffic signs in 3D space by examining the stereo pictures that were collected by the cameras. This allows the car to change its speed and adhere to traffic regulations.

- Detailed application of stereo in Robotics is mentioned below

Navigation- Robotics use stereo vision technology for navigation and obstacle avoidance. The robot can travel through complicated surroundings and avoid collisions by using the system's ability to precisely detect and locate objects in 3D space by interpreting the stereo pictures that the cameras have acquired.

Grasping- Robotics uses stereo vision technology for object movement and gripping. The system can precisely establish the location and orientation of things in 3D space by evaluating the stereo pictures that were acquired by the cameras. This allows the robot to precisely grab and move objects.

Quality Control: Stereo vision technology is used in robotics for quality control in manufacturing processes. By analyzing the stereo images captured by the cameras, the system can detect defects in products and classify them based on their severity, enabling the robot to sort and discard defective products and improve the overall quality of the manufacturing process.

- Detailed application of stereo in Medical Imaging is mentioned below

Image-Guided Surgery: In image-guided surgery, stereo vision is used to give the surgeon feedback in real time while the procedure is going on. By analysing the stereo images that the cameras take, the system can make a 3D model of the surgical site and show it on a monitor. This helps the surgeon navigate the complex anatomy and do the procedure with accuracy.

Rehabilitation: In rehabilitation, stereo vision technology is used to track the progress of individuals with medical problems, like those caused by a stroke or a spinal cord injury. By analyzing the stereo images captured by the cameras, the system can create a 3D model of the patient's movements, enabling the therapist to assess the patient's range of motion and track the progress of their rehabilitation.

Diagnosis: Stereo vision technology is used in medical imaging for the diagnosis of medical conditions, such as tumors, fractures, and deformities. Analyzing the stereo images captured by the cameras helps the doctor see the problem more clearly and make a more accurate diagnosis.

Radiation Therapy: In radiation therapy, stereo vision technology is used to make a 3D model of the tumour and the tissues around it. This helps the doctor plan the radiation treatment more accurately. By analyzing the stereo images physicians deliver radiation therapy with precision and minimize damage to healthy tissue.

Question 6-

Write down steps to stitch images to create the panorama. Use the three

Taj Mahal Images provided with this assignment to create one panorama.

Show panorama into one.

Colab link - <https://colab.research.google.com/drive/1FNqUircZA68RttW2s9z-jptUqGy1leFT#scrollTo=hBdbhSi9et6X>

First I have tried using the inbuilt function in python to stitch the images. But it is not showing good results.



Pre-Processing the images-

I tried image resizing so that all images will be of same size also I have rotated the third image so that appearance of all the images is same.

Then I stitched first and second image and then I stitched this image with third image using inbuilt function. Still the result is not good.



Steps to stitch the image to create the panorama-

1. Preprocess images: Preprocess each image before stitching to improve the quality of the final panorama. This involves steps like resizing, converting to grayscale, applying filters, enhancing contrast, and removing distortion as described earlier.
2. Load images: Load all preprocessed images that we want to stitch together using a library OpenCV. The most important point in panorama stitching is-- make sure that the images are in the correct order so that they can be stitched together seamlessly.
3. Find feature points: Use a feature detection algorithm like SIFT, ORB, or SURF to find feature points in each preprocessed image. Feature points include corners, edges, and blobs.
4. Match feature points: Use a feature matching algorithm like brute-force matching or FLANN matching to match feature points between pairs of adjacent images.

5. Estimate homography matrix: Use the matched feature points to estimate the homography matrix that maps one preprocessed image onto another. However, since the images may be rotated and have different sizes, the homography matrix may need to be refined using an iterative approach like RANSAC. This involves randomly selecting a subset of matched feature points and estimating the homography matrix based on this subset. The homography matrix is then applied to all the feature points, and the inliers (points that are consistent with the homography matrix) are used to refine the homography matrix.
6. Warp images: Use the refined homography matrix to warp the second preprocessed image so that it aligns with the first. This involves applying a perspective transformation to the second image using a function like `cv2.warpPerspective()` in OpenCV. The output of this step will be a single image that combines the first two images.
7. Blend images: Blend the two images together to create a seamless transition between them. We can use methods like feathering or multi-band blending to achieve this
8. Repeat: Repeat steps 3-7 for each pair of adjacent images, until all the preprocessed images have been stitched together into a panorama. In each iteration, the previous output image is used as the first image, and the next image is used as the second image.
9. Crop: Finally, crop the resulting panorama to remove any unwanted black borders or areas of overlap between the original images. Create the function to crop the image or use the inbuilt function.

I created the stitcher class from scratch to improve the results of panorama-

Algorithm-

1. Load the given images
2. Create SIFT object
 - 2.1. Find keypoints and descriptor in the images using SIFT algorithm
 - 2.2. Store the keypoints in the list and descriptor in the numpy array for all three images
 - 2.3. Use Brute force matcher to match descriptors using KNN=2, method will return the two best matches for each descriptor.
 - 2.4. Calculate the ratio between the distances of the two best matches. If the distance of the first match i is less than 0.3 times the distance of the second match j , we consider this match to be good and add it to the `best_matches` list.
 - 2.5. Find pixel coordinates of the keypoints in the two images that correspond to the best matches found by the `BFMatcher`. These keypoints can be used to perform further operations estimating a homography between the two images.
3. Reshape the array.
4. Estimate a homography matrix between the two sets of keypoints. Use RANSAC to remove outliers. Give some maximum allowed distance in pixels between a keypoint in one image and its corresponding keypoint in the other image. Any match with a distance larger than this threshold is considered an outlier and is not used in the homography estimation.

- 4.1. Transform the (right) image using the homography matrix. The function applies the homography transformation to each pixel in the right image and maps it to its corresponding location in the output image.
5. Blend the transformed right image with the original left image to create a panoramic image.

Output panorama-



Panorama output for custom images-



Reason behind the black part in the panorama image are

Overlapping regions:

When stitching images together, it is important to have overlapping regions between adjacent images to ensure blend happens seamlessly. But if the overlapping regions are not aligned properly or are too small, there may be areas where there is not enough information to blend the images together. This will result in black regions where no image data is available there.

In our case first image given to us is of higher size as compared with the other two images. That's why we must reduce the size of first image so that it becomes equal to the other two images. While doing image size reduction, image border will be changed also we may lost some important information in the image. And then if we try to stitch them together we get the black region where no information is available

Same is the case for third image given to us. Third image is slightly rotated. So we have to rotate the image first before stitching otherwise we will not get good stitching result. Once we do rotation we will get the image with black border. And if we try to stitch it with other image we will not be able to perform good blending at the image border, this will result in poor stitching. So we have to remove black box that is around the image and then do the image stitching and blending.

Lens distortion:

Another factor that can contribute to black regions in a stitched panorama is lens distortion. Most lenses have some degree of distortion, which can cause the edges of the images to be warped or stretched. When stitching these images together, the software may have difficulty aligning them properly, which can result in black regions where the images do not overlap correctly.

END