# Computer vision

*CSL7360*

**Ajit Kumar**

**Instructor - Prof. Anand Mishra**

## Assignment-2A

## Question1:

### Approach1 : Encoding

- In this approach, I have made a dataframe containing 3 columns “Anchor”, “Positive” and “Negative”.
- “Anchor” columns contain all images names and “Positive” contains the same image name as “Anchor” and “Negative” contains different images names in random order.
- I have used a pre-trained model for getting the encoding of the images.

```
self.efficientnet=timm.create_model('efficientnet_b0'
, pretrained='True')
```

- For the training, I have used the triplet loss function and adam optimizer.

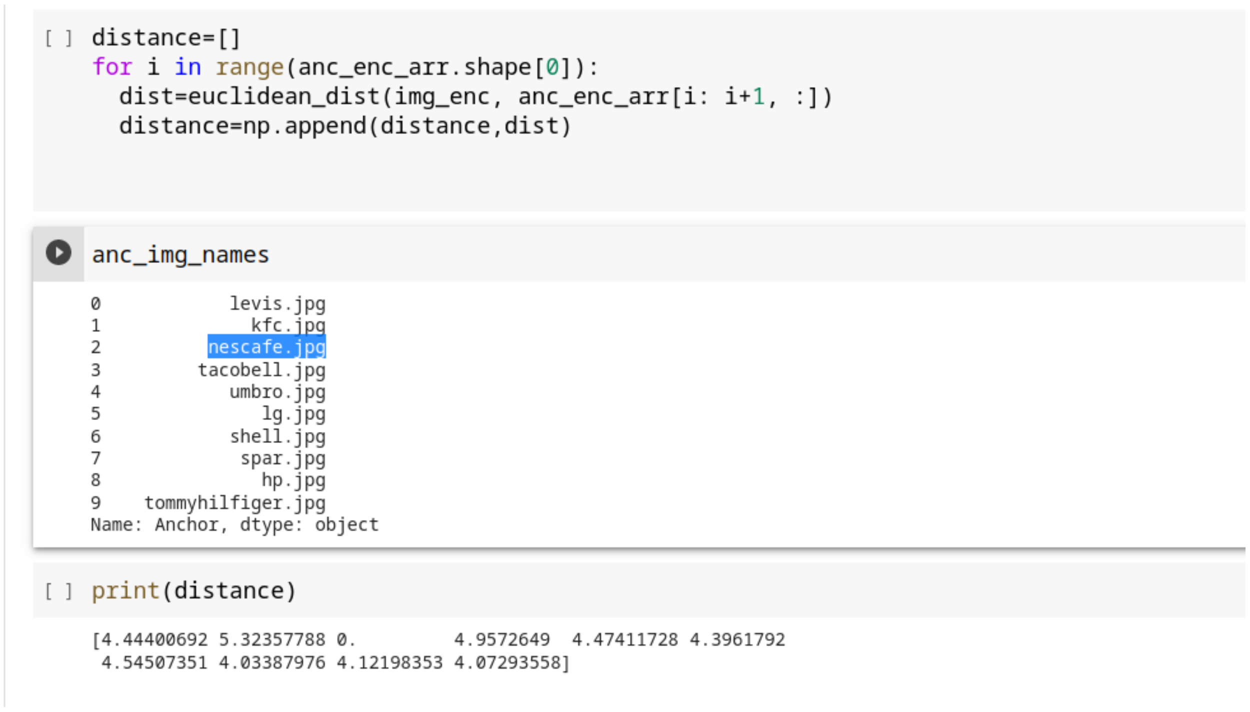
```
criterion=nn.TripletMarginLoss()

optimizer=torch.optim.Adam(model.parameters(), lr=LR)
```

- While training, I have saved the best weight for the encoding.
- Then got the encoding for each image and stored into the data frame.
- Also got the encoding for the testing logo and then

calculated the euclidean distance between the testing logo and all the “Anchor” images.

**Observation:**

- If we are comparing the distance for the images on which it trained, getting the distance as zero. Following images contain the result for the **nescafe.jpg** and get the result **0**.



```
distance=[]
for i in range(anc_enc_arr.shape[0]):
  dist=euclidean_dist(img_enc, anc_enc_arr[i: i+1, :])
  distance=np.append(distance,dist)
```

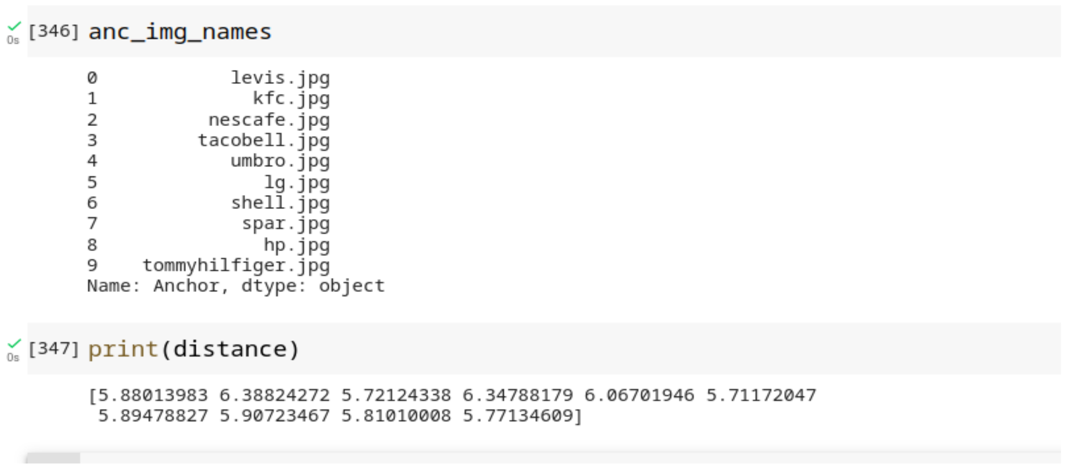
```
anc_img_names
```

```
0            levis.jpg
1              kfc.jpg
2          nescafe.jpg
3         tacobell.jpg
4            umbro.jpg
5               lg.jpg
6            shell.jpg
7             spar.jpg
8               hp.jpg
9     tommyhilfiger.jpg
Name: Anchor, dtype: object
```

```
print(distance)
```

```
[4.44400692 5.32357788 0.         4.9572649  4.47411728 4.3961792
 4.54507351 4.03387976 4.12198353 4.07293558]
```

- If we are comparing some other images(not present in the training set), which contain the specific logo then it doesn’t give a good result. It means that the images which do not contain the same logo as testing images give less distance from the image which contains the logo.



```
[346] anc_img_names

0            levis.jpg
1              kfc.jpg
2          nescafe.jpg
3         tacobell.jpg
4            umbro.jpg
5               lg.jpg
6            shell.jpg
7             spar.jpg
8               hp.jpg
9     tommyhilfiger.jpg
Name: Anchor, dtype: object

[347] print(distance)

[5.88013983 6.38824272 5.72124338 6.34788179 6.06701946 5.71172047
 5.89478827 5.90723467 5.81010008 5.77134609]
```

- Above images contains the Levis logo but it gives the distance for levis image is **5.88013983** which is greater than nescafe.jpg, lg.jpg, hp.jpg and tommyhilfiger.jpg which distance is getting as **5.72124338**, **5.71172047**, **5.81010008** and **5.77134609** respectively. I have tried with different testing images but not getting the correct result.

**Limitation:** Due to less number of images, I have inserted the same image for the **positive** image. So there is no learning happening during the training. So it is giving zero distance for testing on the training images but bad results on the new images.

## Approach2 :  Template Matching

- Read the logo image and convert it into the gray scale.
- Traverse all the images that are present in the folder and read it.
- Convert each image to the gray scale.
- Then apply the template matching algorithm.

```
result = cv2.matchTemplate(img_gray, logo_gray, cv2.TM_CCOEFF_NORMED)
```

- Take a threshold as 0.8.
- This is the following output for each image and we can see that loc[1] value gives the matching result and for all images it is less than the threshold.

## Output for the First Example:

```
hp.jpg

(-0.20690877735614777, 0.15650929510593414, (805, 216), (745, 306))

Logo not found

kfc.jpg

(-0.36642175912857056, 0.32862213253974915, (146, 232), (863, 343))

Logo not found
```

```
levis.jpg

(-0.2836189568042755, 0.35560357570648193, (505, 264), (624, 145))

Logo not found

lg.jpg

(-0.4046027660369873, 0.3764464557170868, (484, 482), (847, 350))

Logo not found

nescafe.jpg

(-0.26755261421203613, 0.43968716263771057, (134, 330), (136, 218))

Logo not found

shell.jpg

(-0.1931936889886856, 0.2516849637031555, (950, 298), (261, 7))

Logo not found

spar.jpg

(-0.37393710017204285, 0.47376421093940735, (915, 462), (830, 410))

Logo not found

tacobell.jpg

(-0.38691946864128113, 0.40521830320358276, (220, 226), (1144, 486))
```

```
Logo not found

tommyhilfiger.jpg

(-0.3722340166568756, 0.29572850465774536, (881, 367), (290, 495))

Logo not found

umbro.jpg

(-0.25178492069244385, 0.21928204596042633, (906, 421), (328, 180))

Logo not found
```

## Output for the Second Example :

```
honda.jpg

(-0.267192006111145, 0.2295844405889511, (966, 249), (119, 686))

Logo not found

hp.jpg

(-0.31191951036453247, 0.2593490481376648, (60, 428), (407, 573))

Logo not found

lg.jpg

(-0.3656882643699646, 0.40515926480293274, (1090, 90), (449, 588))

Logo not found
```

```
motorola.jpg

(-0.2869853973388672, 0.36276862025260925, (141, 416), (446, 589))

Logo not found

pepsi.jpg

(-0.3535686433315277, 0.28235334157943726, (45, 386), (401, 346))

Logo not found

puma.jpg

(-0.4353804886341095, 0.4512764811515808, (568, 285), (186, 781))

Logo not found

rolex.jpg

(-0.3359859883785248, 0.4490935802459717, (20, 426), (994, 0))

Logo not found

starbucks.jpg

(-0.24176011979579926, 0.2860548794269562, (124, 624), (454, 587))

Logo not found

toyota.jpg

(-0.2431039959192276, 0.27720773220062256, (432, 173), (999, 563))
```

**Logo not found**

**warnerbros.jpg**

**(-0.294883131980896, 0.26273503899957428, (817, 198), (267, 754))**

**Logo not found**

## Observation:

- In the first above result, we are matching to the levis.jpg. But for some images the matching result is greater than levis.jpg. For example, levis value is **0.35560357570648193** and lg.jpg value is **0.3764464557170868**.
- In the second example also, we are matching to the starbucks.jpg. But for some images the matching result is greater than starbucks.jpg. For example, the value is starbucks **0.2860548794269562** and rolex.jpg value is **0.4490935802459717**.

## Approach 3: Template Matching( By cropping the Logo image):

- In this approach, I tried to crop the logo image and image containing only the logo part of the images.

- And then apply the inbuilt matchtemplate algorithm to find the matching point.
- The function returns a floating-point array, which represents the similarity between the template and the target image at each position. The array is the same size as the target image, with each element containing a value that indicates how well the template matches the corresponding pixel in the target image.
- Then this result passes to the c2.minMaxLoc() function, and it returns minimum and maximum values in a given array.
- It is commonly used to find the position of the best match (maximum value) or worst match (minimum value) between a template image and a larger target image.
- The function returns a tuple containing four values:
    1. The minimum value in the input array
    2. The maximum value in the input array
    3. The position of the minimum value (as a tuple)
    4. The position of the maximum value (as a tuple)
- Then the 2nd element represents the maximum matching, it is compared with the threshold value, if it is greater than the **threshold=0.6** then draw the boundary box on the given coordinate.
- But, in our result i am getting less value than threshold, so no matching is detecting in the given example.

```
Result:

hp.jpg

(-0.39969775080680847, 0.3594321310520172, (235, 7), (163,
100))

Logo not found

kfc.jpg

(-0.4741867780685425, 0.4042331278324127, (120, 73), (120,
90))

Logo not found

levis.jpg

(-0.37870925664901733, 0.41346728801727295, (60, 60), (29,
47))

Logo not found

lg.jpg

(-0.5700510740280151, 0.4154691696166992, (56, 154), (141,
76))
```

```
Logo not found

nescafe.jpg

(-0.387645959854126, 0.4198983609676361, (118, 79), (127,
70))

Logo not found

shell.jpg

(-0.4593660235404968, 0.39826029539108276, (116, 222),
(163, 187))

Logo not found

spar.jpg

(-0.5624626278877258, 0.5464843511581421, (70, 90), (72,
107))

Logo not found

tacobell.jpg

(-0.44872623682022095, 0.4203816056251526, (36, 109), (88,
94))
```

**Logo not found**

**tommyhilfiger.jpg**

**(-0.4089187979698181, 0.36633849143981934, (104, 24), (183, 162))**

**Logo not found**

**umbro.jpg**

**(-0.4912477135658264, 0.5578799247741699, (267, 172), (137, 20))**

**Logo not found**

Reference: https://docs.opencv.org/4.x/d4/dc6/tutorial_py_template_matching.html

## Question2:

### Preprocessing Step:

- Read the image using openCv
- Convert it to the grayscale
- Apply the gaussian blur to remove the noise.
- Then apply the canny edge detector to detect the line, but it gives the two lines because there are two edges in the line.
- To get only one line in the image, first apply the dilation to fill the gap between both edges and to make thin apply the erosion.
- All of the above steps are the preprocessing steps.

### Algorithm:

- Set the Hough transform parameters

  rho_resolution = 1

  theta_resolution = np.pi / 180

  threshold = 250

- Calculate the width and height of the images to traverse over the images where the pixel value is greater than zero.
- Also calculate the max_rho value and rho gives the distance between origin to the pixels that lie on the line. So its

maximum value would be the diameter of the images.

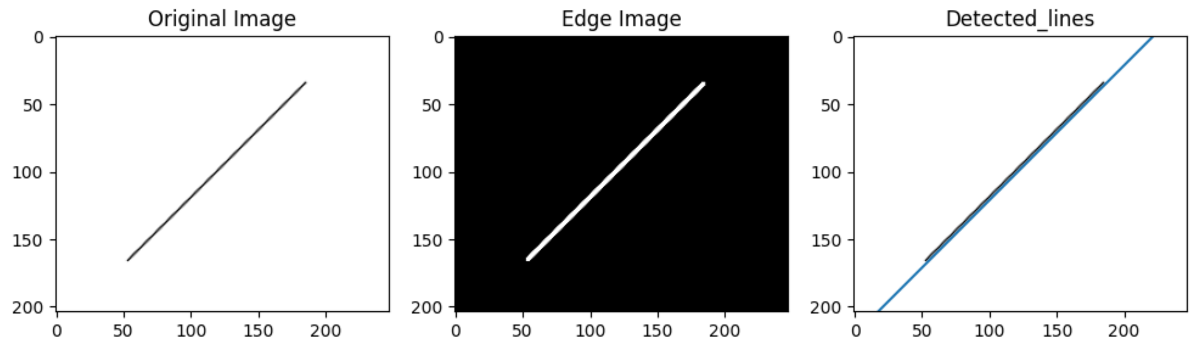
- Value of rho could be [-diagonal, diagonal]
- Initialize a accumulator array of size (2* diogonal_length/ rho_resolution, 180/ theta_resolution).
- Then traverse each coordinate(x, y) of the image, where edge is present( image[x,y] >0).
- Then calculate the theta and rho value for this coordinate, and then increase the value of the accumulator for this index.
- Find the roh_indices and theta_indices by comparing with threshold values.
- Then traverse over the container that contains the rho_indices and theta_indices and calculate the coordinate for this rho and theta value and draw the line on that coordinate.

**Observation**:

1. As we decrease the value of the threshold, the detected line is going to be thin.
2. Time taken by the own_implementation is more than the opencv implementation.

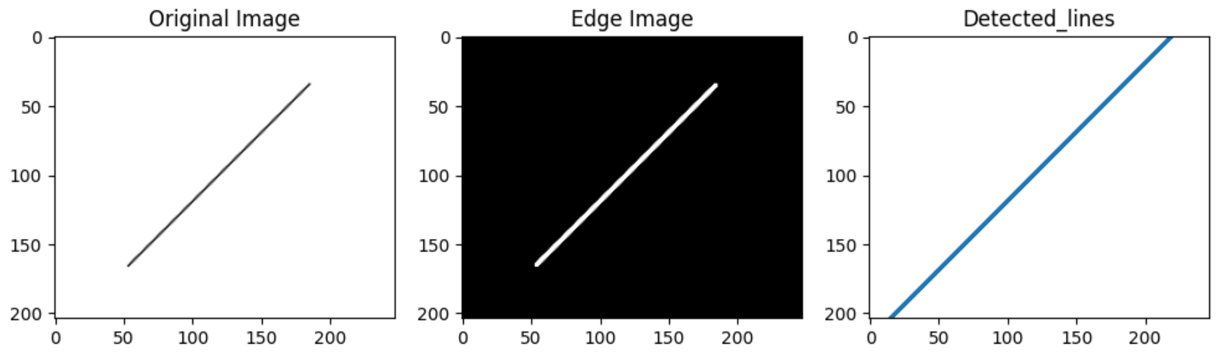
**Result from the openCv library:**

Avg time taken **: 0.278888463973999**



**Result from own implementation:**

Avg Time taken=**15.57**



References:
https://towardsdatascience.com/lines-detection-with-hough-transform-84020b3b1549

https://docs.opencv.org/3.4/d9/db0/tutorial_hough_lines.html

## Question3:

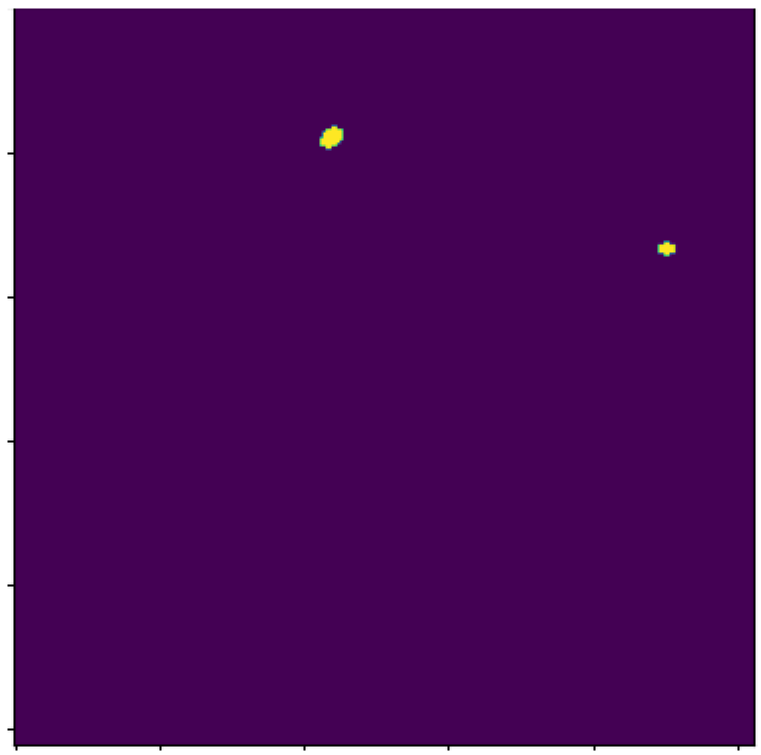
To solve this problem, I have applied three approaches.

1. Absolute difference method .
2. Pre-trained deep learning methode.
3. Image registration and then difference or xor of two images.

### Absolute Difference:

1. Read the images.
2. Resize the both images to (256,256) because to take the difference images should have the same size.
3. Then convert the images to the grayscale.
4. Then take the absolute difference.
5. Then use the adaptive thresholding method to get the binary images. In this method, it doesn't have a fixed threshold value. It calculates the threshold value depending upon the neighbor.
6. Then we apply the morphological method to fill the gap and remove the noise.

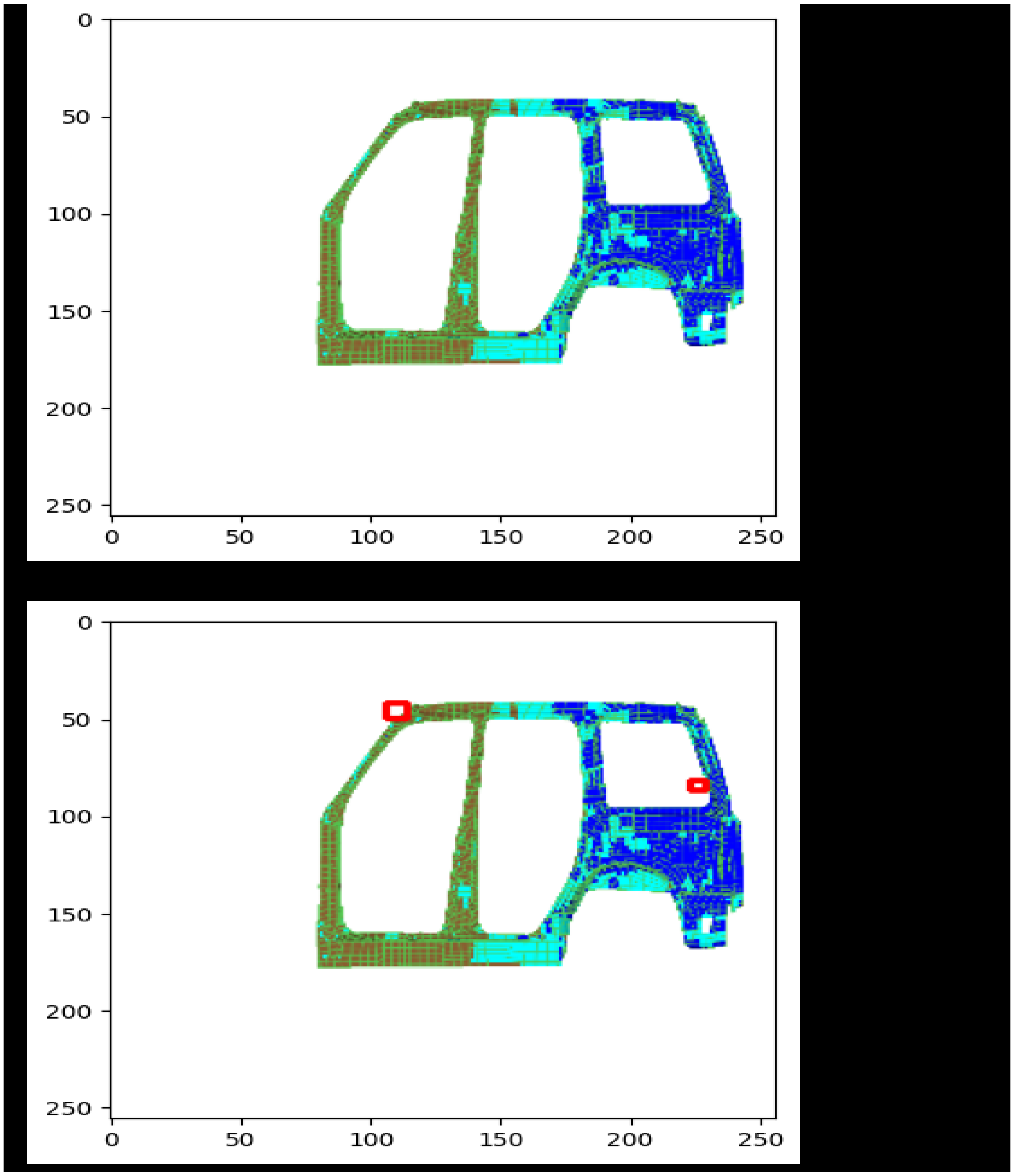
   This is the following morphological image I am getting when comparing with perfect images. As per the question, this is the perfect image, so it should not be the output.



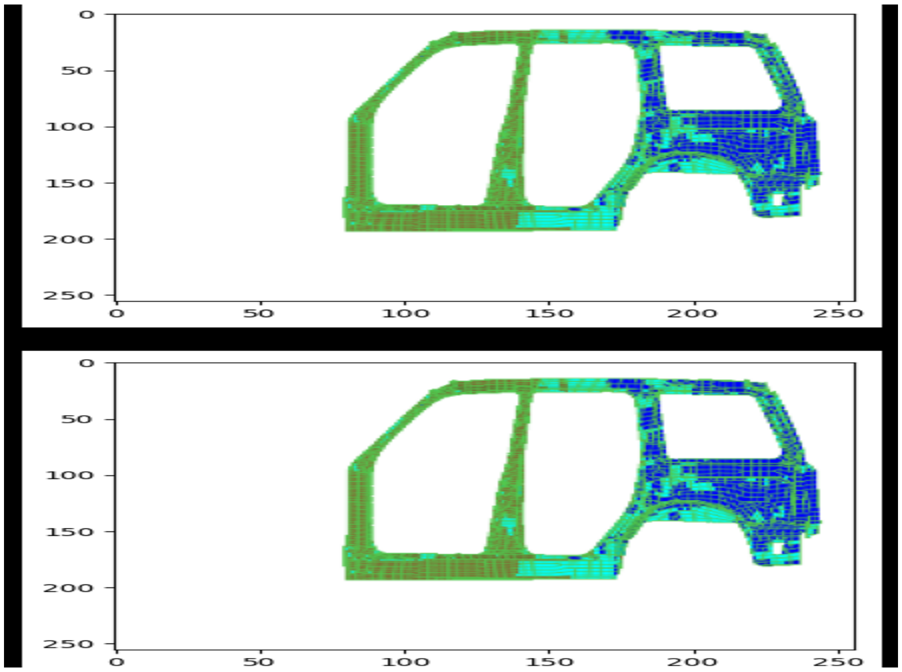
7. Then find the contour for the morphological images for each image in the folder.
8. And draw the contour on the corresponding images.

## Result:

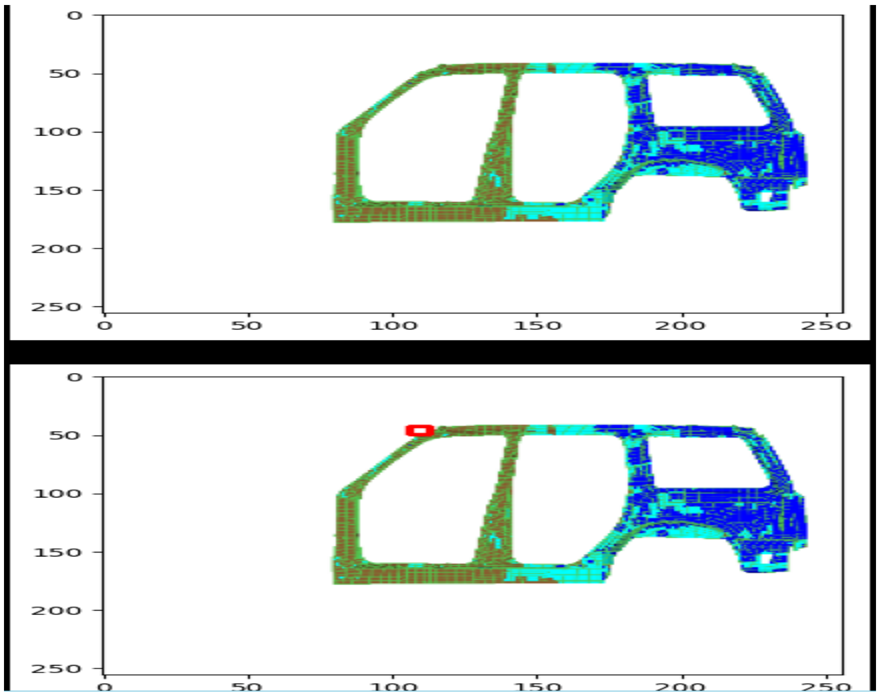
## For perfect image1:



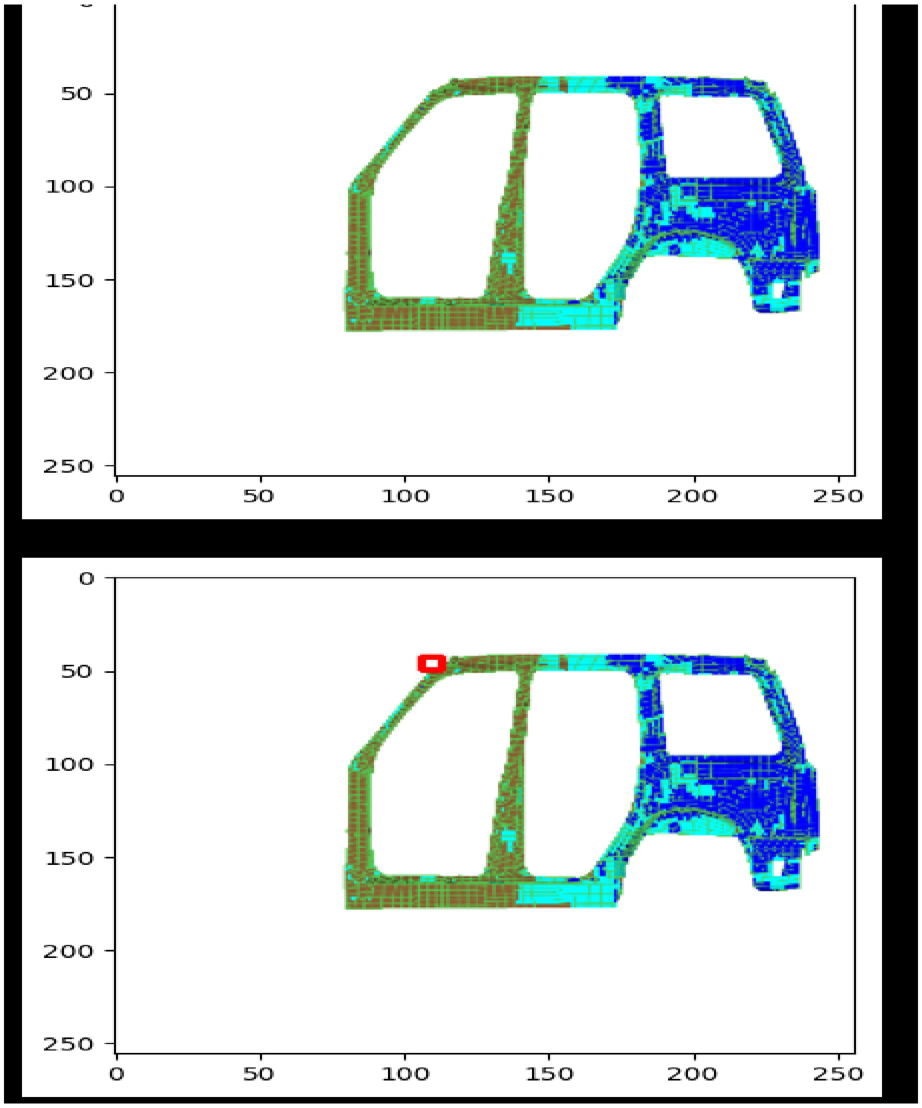
## For Perfect Image2:



## For Faulty Image1:



## For Faulty image2:

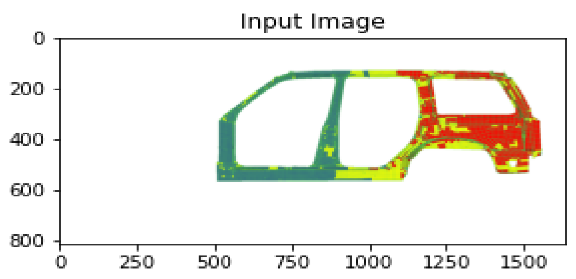


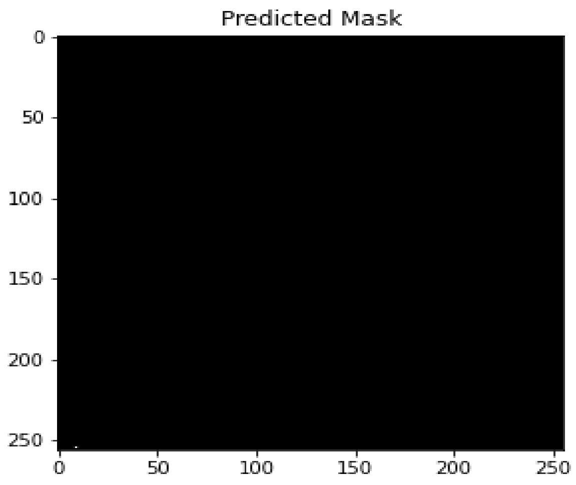
## Deep Learning Method:

In this method, there is no requirement of the reference images. Here we are using the pre-trained deep learning model to detect the fault region in the images. It is used by the segmentation method. That model is generally used to find the defective part in brained images, but I have used it for this task and just checked the result.

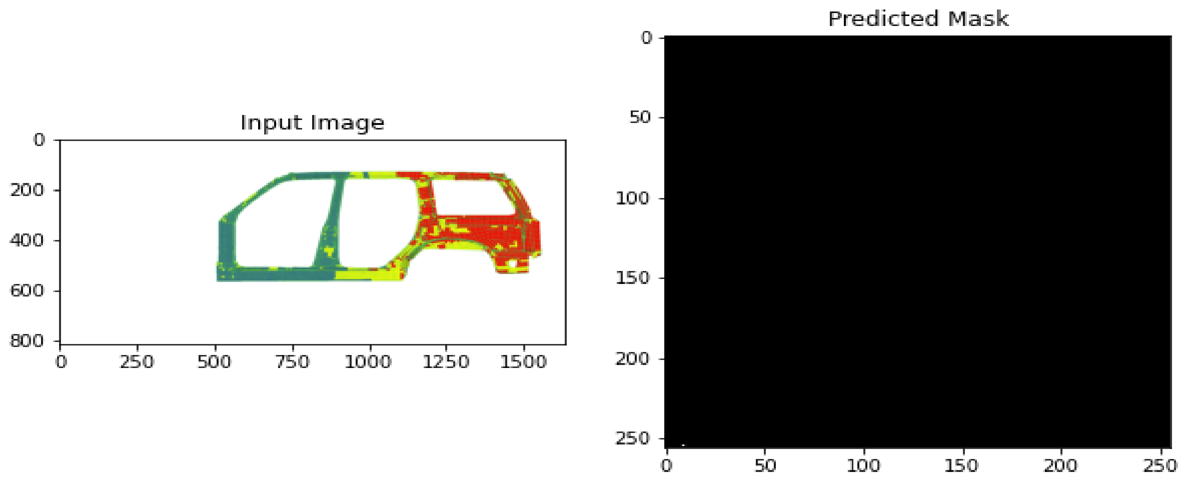
Result:

For faulty1:

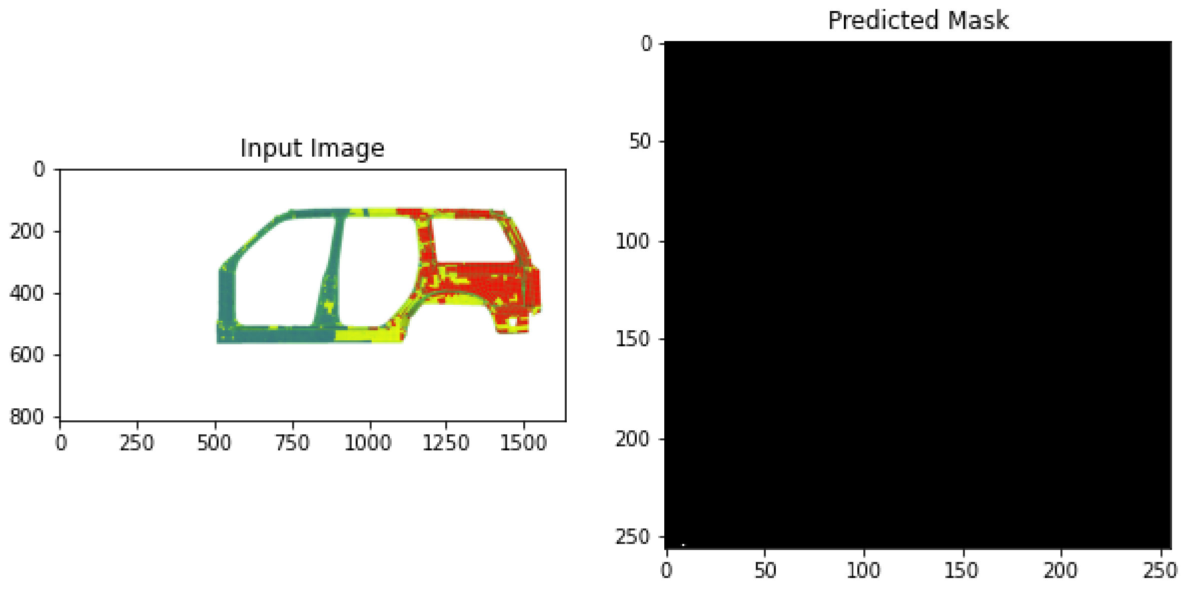




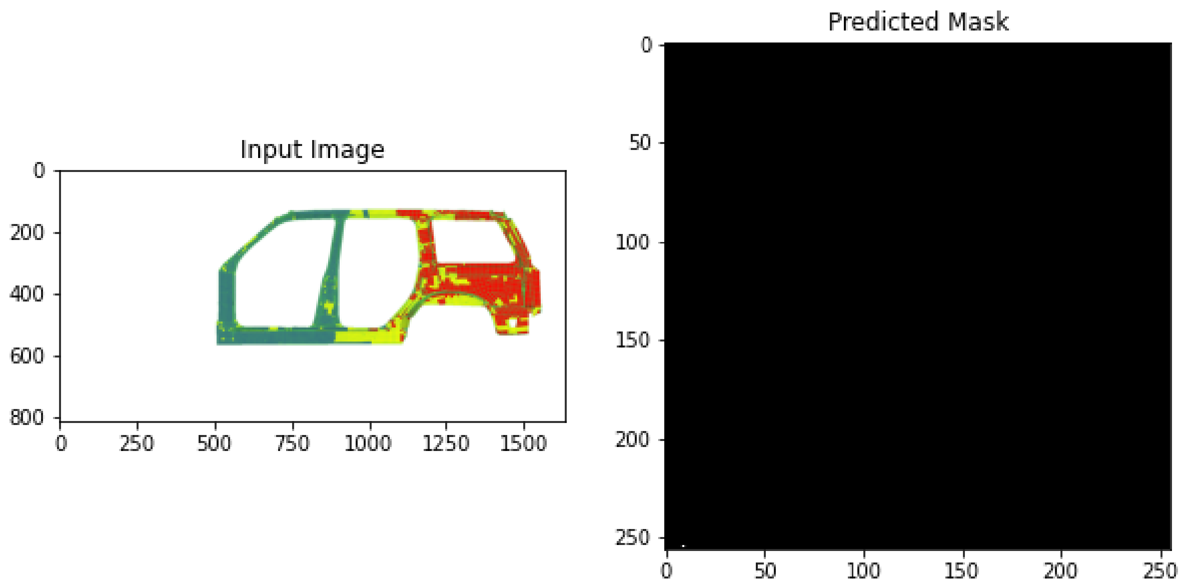
For faulty2:



For perfect1:



For perfect2:



**Conclusion:** This pre-trained model is trained on the brain images. So it is not giving results for any random images.

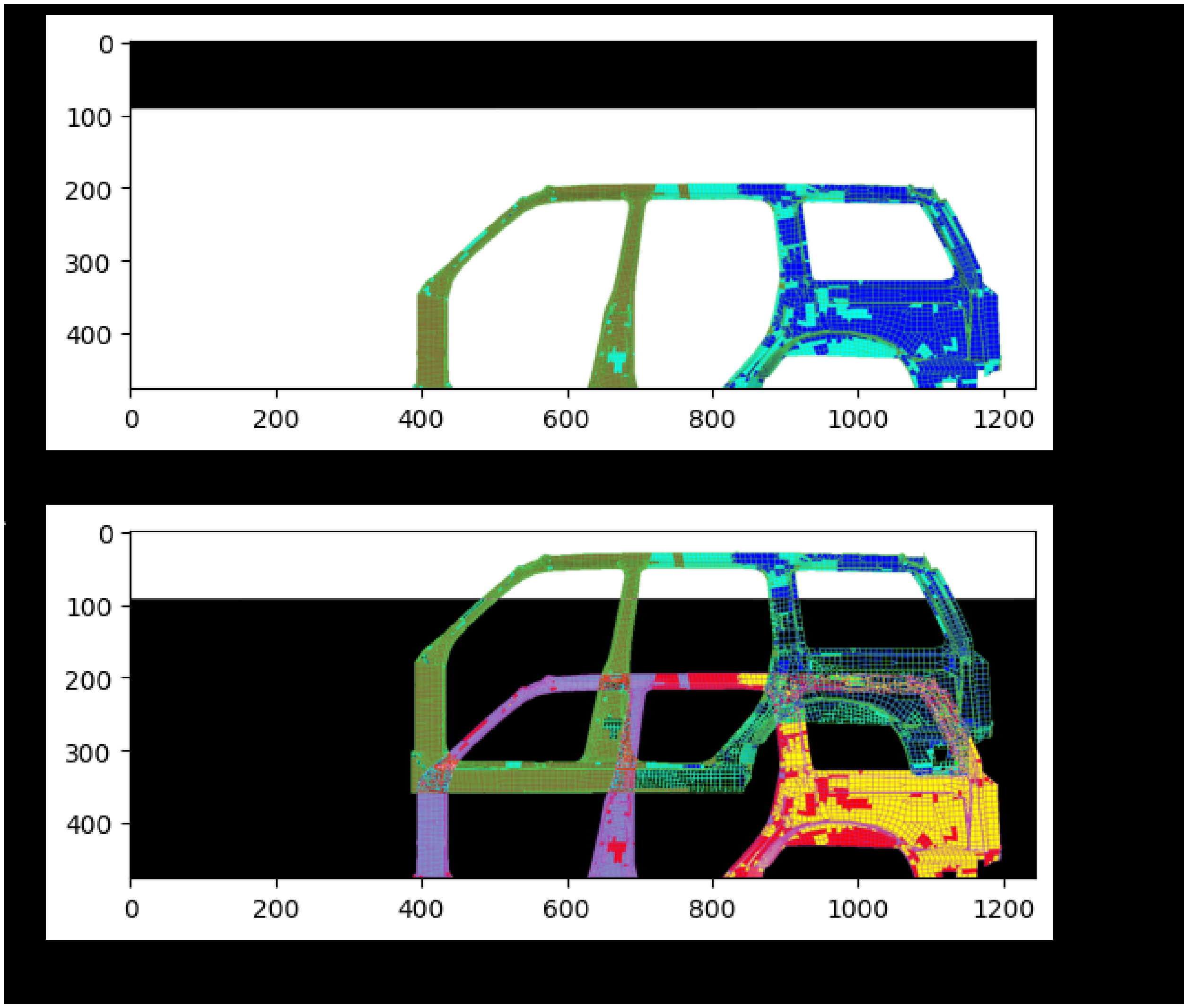
## Image Registration and difference Method:

- Read the image from the given folder.
- Convert it into grayscale.
- Use **SIFT_create()** function to find the matching feature from the reference image and the image which have to check either perfect and faulty.
- Then find the keypoint and descriptor using the function **sift.detectAndCompute().**
- Use bruteforce **BFMatcher()** to match the descriptor.
- Then calculate the homography matrix from the matched keypoints.
- Then apply the **warpPerspective()** function to align the image using a homography matrix on the images that have to check w.r.t to reference the image. And from this operation I got the registered image.
- Then find the difference or x-or to get a defective position in the image.

### Why I apply This Approach:

- Size of the reference image and the image which i have to check is faulty is different, so i have apply image registration to make the same alignment.
- And then take the difference between a reference image and a faulty/ perfect image.

## Result:



## Observation:

- Result is not perfect because alignment of image is not correct because both images are not for the same object. Image registration works if two photos are clicked by the camera for the same object.

# Assignment-2A

## Question1: What is a Homography Matrix? Write down steps to compute Homography Matrix in detail with clear illustrative figures.

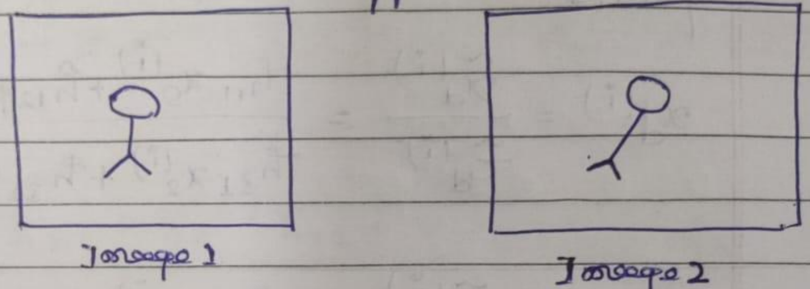
**Homography Matrix:**

- Homography matrix is a matrix that maps one plane to the another plane through a point. It comes under the projective transformation.
- Homography matrix, also known as a perspective transformation matrix or a projective transformation matrix, is a mathematical matrix that represents the transformation between two images of the same scene taken from different viewpoints or under different camera perspectives.
- Homography matrix is commonly used in computer vision, robotics, and augmented reality applications, such as image stitching, 3D reconstruction, object recognition, and image registration. It is typically represented by a 3x3 matrix, and can be computed using a variety of techniques, such as

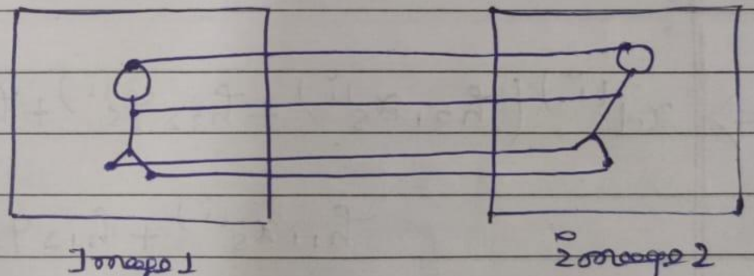
direct linear transformation (DLT) or least-squares estimation.

Following steps to calculate the homography matrix:

Let we have two image that contain same scene with different angle



1. first find the feature from both image



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

$(x_d, y_d) \in$ Cordinate of image 1

$(x_s, y_s) \in$ Co-ordinate of Image 2

for a given pair i of corresponding points

$$x_d^{(i)} = \frac{\tilde{x}_d^{(i)}}{\tilde{z}_d^{(i)}} = \frac{h_{11} x_s^{(i)} + h_{12} y_s^{(i)} + h_{13}}{h_{31} x_s^{(i)} + h_{32} y_s^{(i)} + h_{33}}$$

$$y_d^{(i)} = \frac{\tilde{y}_d^{(i)}}{\tilde{z}^{(i)}} = \frac{h_{21} x_s^{(i)} + h_{22} y_s^{(i)} + h_{23}}{h_{31} x_s^{(i)} + h_{32} y_s^{(i)} + h_{33}}$$

$$\Rightarrow x_d^{(i)} \left( h_{31} x_s^{(i)} + h_{32} y_s^{(i)} + h_{33} \right) = h_{11} x_s^{(i)} + h_{12} y_s^{(i)} + h_{13}$$

$$y_d^{(i)} \left( h_{31} x_s^{(i)} + h_{32} y_s^{(i)} + h_{33} \right) = h_{21} x_s^{(i)} + h_{22} y_s^{(i)} + h_{23}$$

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

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

Now arranging it to matrix form

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

Combining all the equations for all matching feature points

$$\begin{bmatrix} x_s^{(1)} & y_s^{(1)} & 1 & 0 & 0 & 0 & -x_d^{(1)} x_s^{(1)} & -x_d^{(1)} y_s^{(1)} & -x_d^{(1)} \\ 0 & 0 & 0 & x_s^{(1)} & y_s^{(1)} & 1 & -y_d^{(1)} x_s^{(1)} & -y_d^{(1)} y_s^{(1)} & -y_d^{(1)} \\ x_s^{(2)} & y_s^{(2)} & 1 & 0 & 0 & 0 & -x_d^{(2)} x_s^{(2)} & -x_d^{(2)} y_s^{(2)} & -x_d^{(2)} \\ 0 & 0 & 0 & x_s^{(2)} & y_s^{(2)} & 1 & -y_d^{(2)} x_s^{(2)} & -y_d^{(2)} y_s^{(2)} & -y_d^{(2)} \\ \vdots \\ \vdots \\ x_s^{(n)} & y_s^{(n)} & 1 & 0 & 0 & 0 & -x_d^{(n)} x_s^{(n)} & -x_d^{(n)} y_s^{(n)} & -x_d^{(n)} \\ 0 & 0 & 0 & x_s^{(n)} & y_s^{(n)} & 1 & -y_d^{(n)} x_s^{(n)} & -y_d^{(n)} y_s^{(n)} & -y_d^{(n)} \end{bmatrix}$$

$$\begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$$

Date
Page

Solve for $h$

This linear equation can be solve by direct linear transformation or least square estimation :

Constrained Least squares

$$Ah = 0 \quad \text{such that } ||h||^2 = 1.$$

Define least squares problem:

$$\min_{h} ||Ah||^2 \quad \text{such that } ||h||^2 = 1.$$

We know that

$$||Ah||^2 = (Ah)^T (Ah) = h^T A^T A h$$

and

$$||h^2|| = h^T h = 1.$$

So now

$$\boxed{\min (h^T A^T A h) \quad \text{such that } h^T h = 1}$$

Page

Constrained least squares

$$\min_{h} \left( h^T A^T A h \right) \quad \text{such that} \quad h^T h = 1.$$

Define loss function

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

Taking derivative of $L(h,\lambda)$ w.r.t $h$

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

$$A^T A h = \lambda h$$

Eigen vector $h$ with smallest eigen value $\lambda$ of matrix $A^T A$ minimizes the loss function $L(h)$

## Question2: What is stereo matching? Write down three applications of stereo matching.

## Answer:

## Stereo Matching:

- Stereo matching is the process of finding corresponding points or features between two or more images of the same scene taken from different viewpoints or with slightly different angles. It is a fundamental problem in computer vision and is widely used in applications such as 3D reconstruction, augmented reality, and autonomous navigation.
- The basic idea behind stereo matching is to compare the images pixel by pixel or feature by feature to find matches between them. The goal is to find pairs of pixels or features in the left and right images that represent the same point or object in the scene. This can be done by measuring the similarity between the two images at each pixel or feature location and finding the best match.
- One of the most popular approaches to stereo matching is based on the use of disparity maps. A disparity map is a representation of the differences in depth or distance between corresponding points in the left and right images.

It can be computed by finding the horizontal displacement between the corresponding pixels in the left and right images. This displacement is called the disparity, and it is inversely proportional to the depth of the scene.

## Three Application of Stereo Matching:

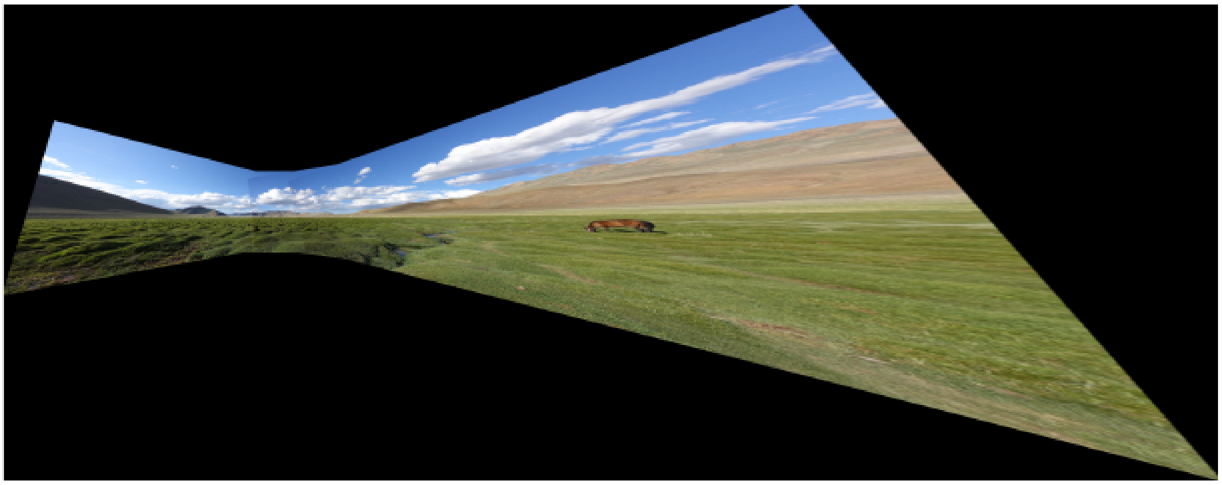
- It is used in 3-d construction.
- Augmented reality
- Autonomous navigation

## Question3:

- As we know the method of stitching of two images, one image is considered as base image and another is secondary images and panoramic is formed by stitching the secondary images. So can we apply this method for the multiple images? So the answer is No. If we apply this methode on multiple images then we will face heavy distortion in image and we can get the incorrect result.
- So to solve the multiple image to panoramic form, we first project all the images to the cylinder and then unroll them and then finally stitch them together.
- First read all the images and store them in the list name

**Images**.

- If we have more than two images, then images near to the side get distorted.



- Then project all the images to the cylinder and projection on the cylinder is done by the following formula.

$$x' = f * tan((x - xc) / f) + xc$$
$$y' = \{ (y - yc) / cos((x - xc) / f) \} + yc$$

**Algorithm for project the image on the cylinder:**

1. Define the parameters of the cylinder such as radius and height.
2. Then create a new blank image of the size of the cylinder.
3. For each pixel in the new image, calculate its corresponding location in the original image using cylindrical projection.
4. Copy the pixel value from the original image to the

corresponding location in the new image.
5. Repeat steps 3-4 for all pixels in the new image
6. Load every image and repeat step-6 for all images to be stitched.

**Algorithm for the stitching the one by one images on the base images:**

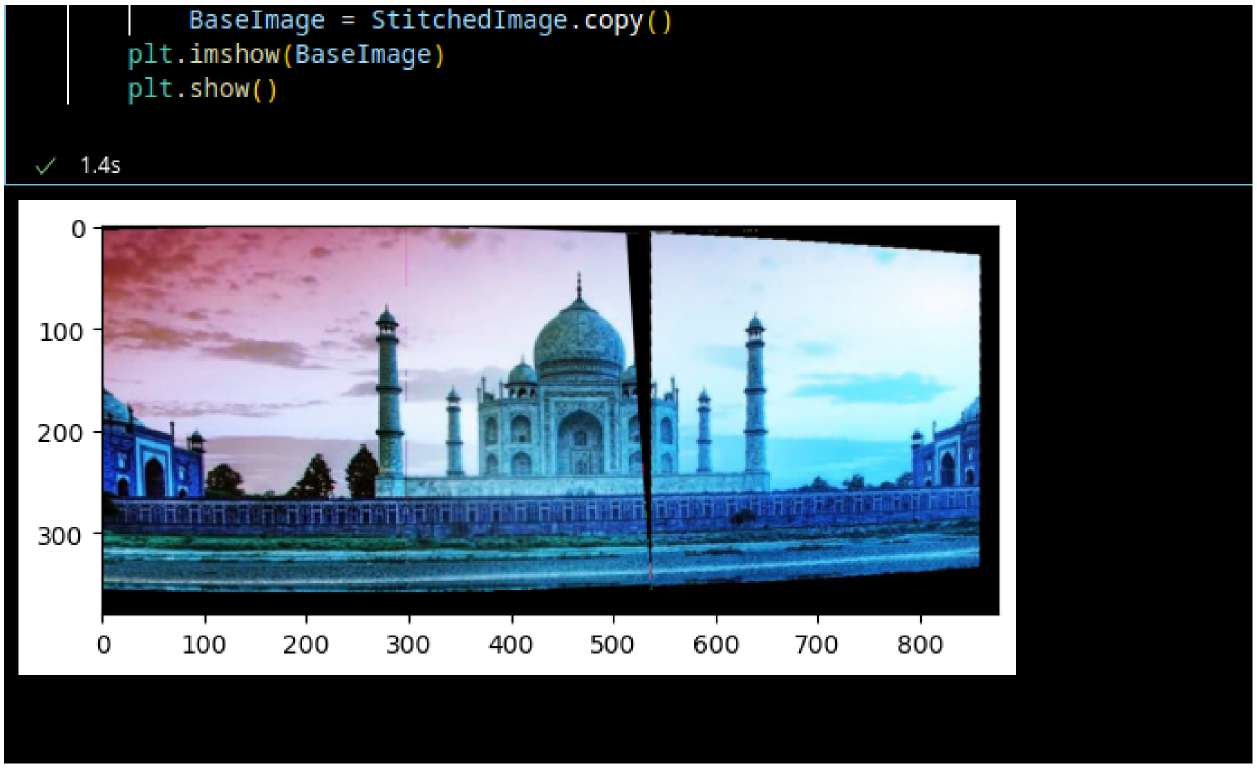
1. First project the sec_image to the cylinder.
2. Then find the matches using the **cv2.SIFT_create()** function that is used to extract the key points and descriptor and **cv2.BFMatcher()** function. **cv2.BFMatcher()** is bruteforce matches algorithm that is used to find the matching feature in two images.
3. Then find the good matching **Lowe's Ratio Test.**
4. Then find the homography matrix for the good matches in the images.
5. For the stitching, find the new frame of the stitched images.
6. Finally placing the images upon one-another using **cv2.warpPerspective** function.
7. Then return the stitched images.

**Algorithm For stitching the multiple images:**

- First project the first image on the cylinder.
- Call the stitching image function for every image except the first image.
- Then the result by the stitching function copy on the base image.

- Finally show the final output.

Output:



```
        BaseImage = StitchedImage.copy()
    plt.imshow(BaseImage)
    plt.show()
```

**Reference:**
[https://kediarahul.medium.com/panorama-stitching-stitch-multiple-images-using-opencv-python-c-875a1d11236d](https://kediarahul.medium.com/panorama-stitching-stitch-multiple-images-using-opencv-python-c-875a1d11236d)

[https://kediarahul.medium.com/panorama-stitching-stitching-two-images-using-opencv-python-c-3702e4993d8](https://kediarahul.medium.com/panorama-stitching-stitching-two-images-using-opencv-python-c-3702e4993d8)

[https://www.youtube.com/watch?v=taty6lPVcmA&list=PLjMXczUzEYcHvw5YYSU92WrY8IwhTuq7p&t=3026s](https://www.youtube.com/watch?v=taty6lPVcmA&list=PLjMXczUzEYcHvw5YYSU92WrY8IwhTuq7p&t=3026s)

https://stackoverflow.com/questions/51197091/how-does-the-lowes-ratio-test-work