Assignment 1
CSL7360: Computer Vision

Gaurav Sangwan (B20AI062)

Question1: Harris Corner Detection

I successfully implemented the Harris Corner detection algorithm from scratch in Python with the use of numpy and scipy library.

The key steps in my implementation are:

- Converting the image to grayscale to compute gradient.
- Computing Image Gradients, which can be configured in the "Scratch" class, as currently I have used Sobel Kernels, but one can also replace the self.d_x and self.d_y with Scharr, Prewitt or Roberts Cross operator.
- Computing Harris Matrix Components, by this I mean the computation of Ixx, Iyy and Ixy for this I used scipy.ndimage.gaussian_filter, with a sigma of 1.
- Computing Harris response, previous step's components were used to calculate determinant and trace, which further gave the final reponse given by the formula $det - k * (trace^2)$, here I have chosen k as 0.05 but one can modify it while creating the object for "Scratch" class.
- Threshold to identify Corners, By thresholding Harris reponse matrix we can get potential corners , and for my implementation I have set the thresholds as the fraction of maximum and minimum response values.
- Visualise the corners.

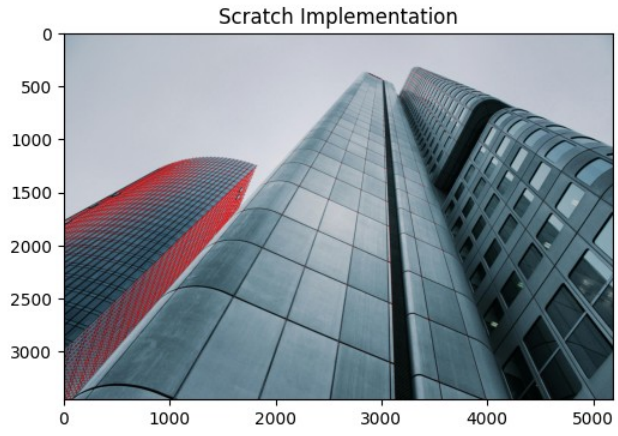
For my implementation, the user can also see the intermediate images before corner estimation by choosing verbose as "True", but if we only need the final corner highlighted image then user should keep verbose as "False".

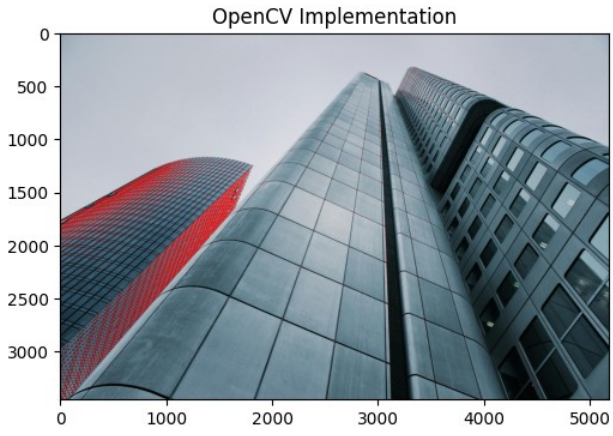
Overall, my implementation needs 3 parameters only,

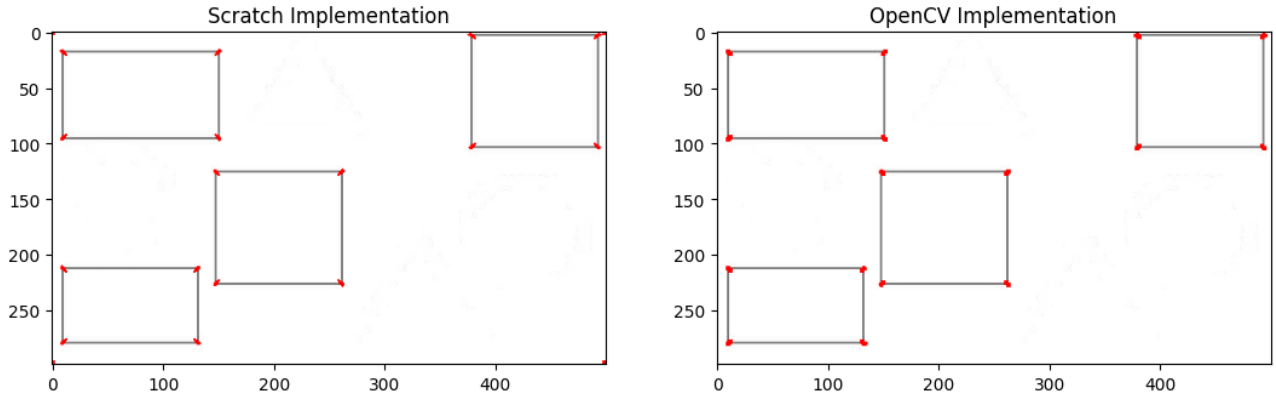
1. k: the empirical constant
2. THRESHOLD_CORNER: fraction of max response for corners
3. THRESHOLD_EDGE: fraction of max response for edges

Performance comparision with OpenCV

I compared the performance of my implementation with OpenCV's cv2.cornerHarris function on the provided test images. Below you can find the visual comparison of two images, rest can be found in the Appendix-I.







As seen in the image, both implementations detect most of the same prominent corners, but there are some differences in the exact corner locations and the number of corners detected. This is likely due to parameter choices.
And mine is noticeable slower since it is not optimized.

Question2:Stereo 3D Reconstruction

In this question, I implemented a stereo 3D reconstruction algorithm to reconstruct a 3D point cloud from a pair of stereo images captured from a stereo camera setup. The provided inputs were:

- Left stereo image (bikeL.png)
- Right stereo image (bikeR.png)
- Intrinsic camera matrices for left and right cameras (bike.txt)

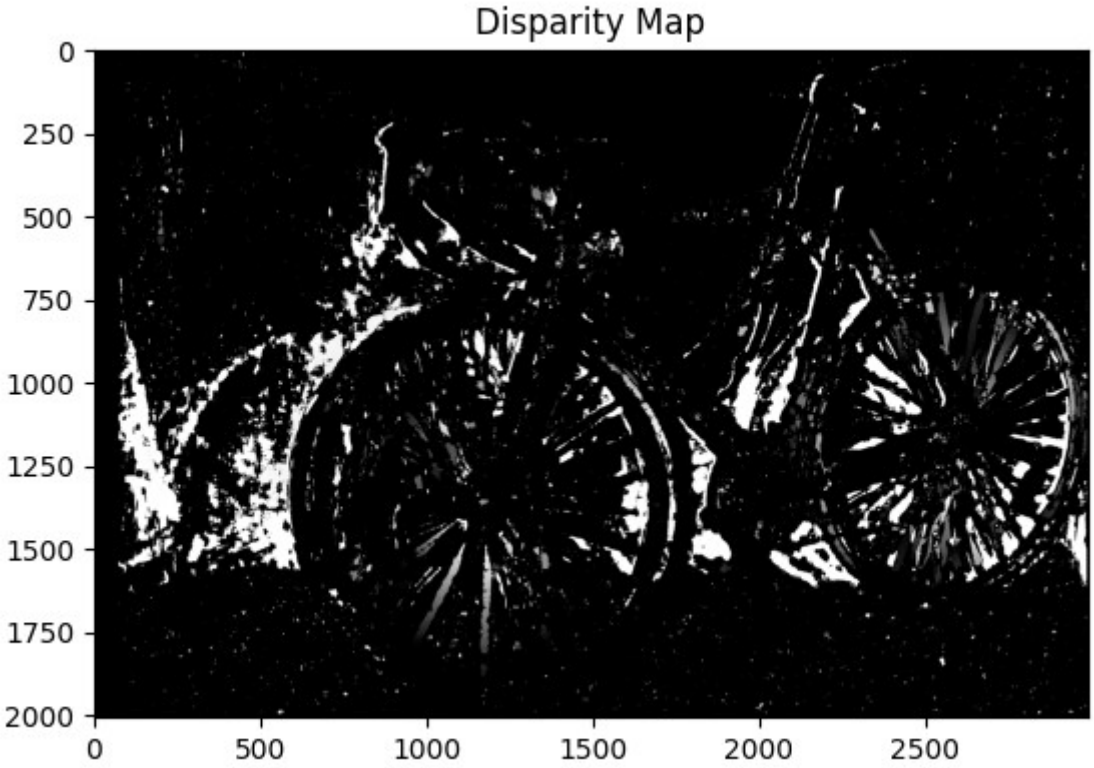
Overall approach involved the following key steps:

- Load the Stereo Images and Camera Parameters
  - Camera parameters are as follows:
    - baseline = 177.288
    - f = 5299.313
    - cx = 1263.818, cx_1 = 1438.004 and cy = 977.763
    - Intrinsic matrices can be seen in bike.txt as the above parameters are derived from there it self.
- Compute Disparity Map, by using the cv2.StereoBM_Create I computed the disparity map between the left and right images.
- Compute Depth Map, by using the formula $depth = (baseline * focallength)/disparity$, I computed the depth map.
- Compute 3d Point cloud, by using the above given parameters I obtained the Projection Matrix Q as follows: and then using that with cv2.reporjectImageTo3D obtained the 3D point cloud in “reconstructed.ply” file. Then Utilised the Open3D library to visualize that PLY file

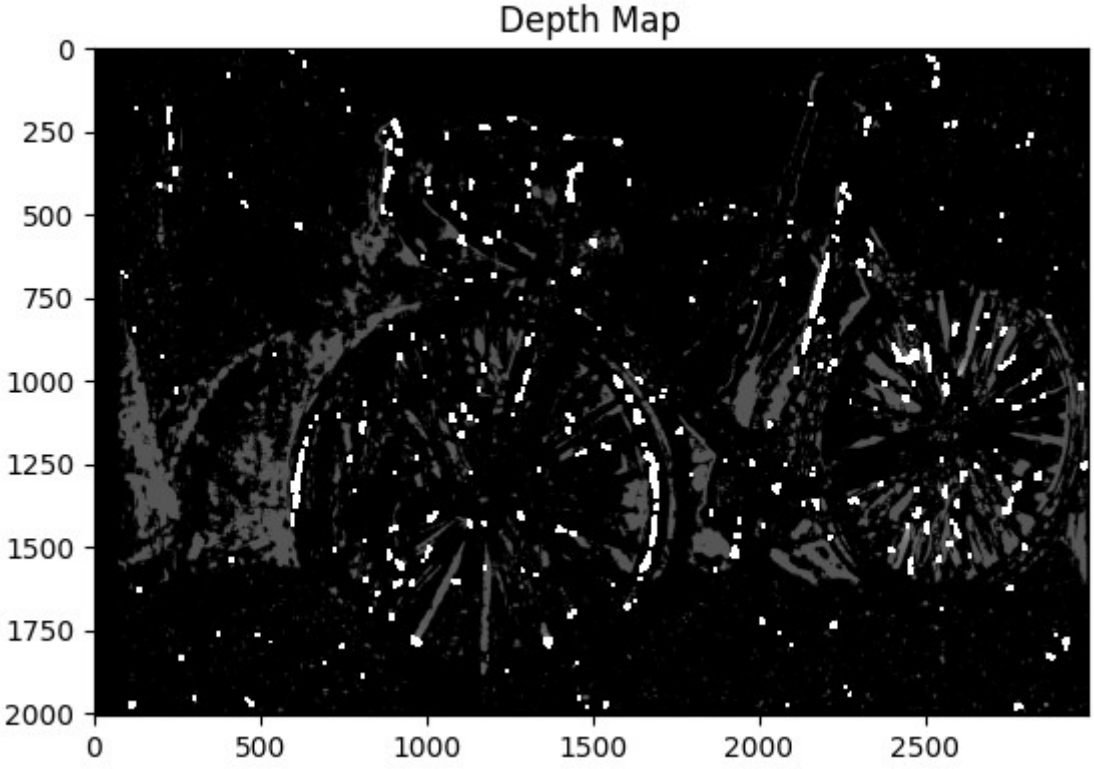
$$Q = \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 0 & f \\ 0 & 0 & -\frac{1}{T_x} & \frac{c_x - c_x'}{T_x} \end{bmatrix}$$

Results

Here is the computed disparity map:



Here is the computed depth map:



Here is the screenshot from the reconstructed point cloud.:

As seen in the point cloud, the 3D structure of the scene (bike, ground, background) is clearly reconstructed from the stereo pair of images.

Question3: Epipolar Geometry

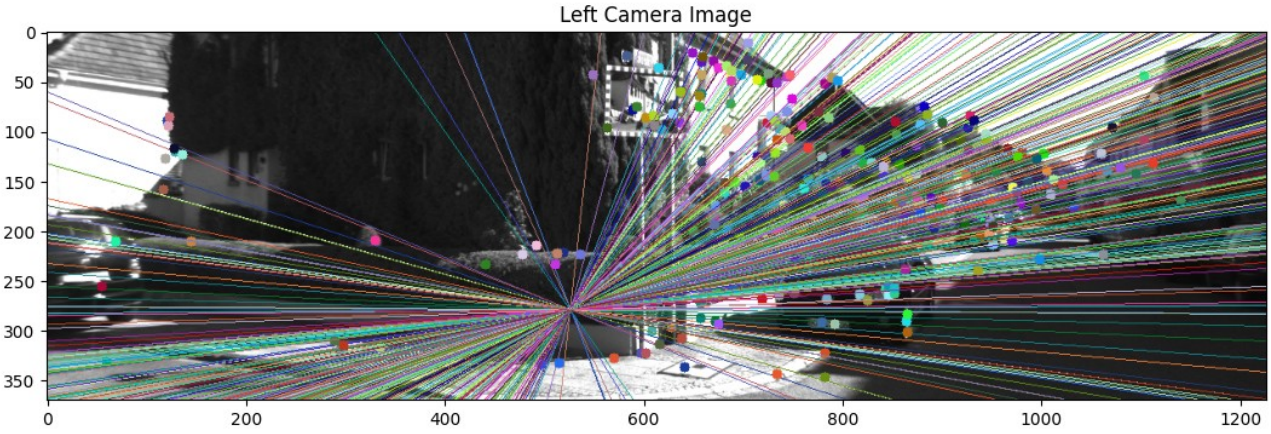
In this question, I explored the concept of Epipolar geometry utilised a fundamental matrix (F) to calculate epipolar lines, and visualize them in two images captured from a static scene.

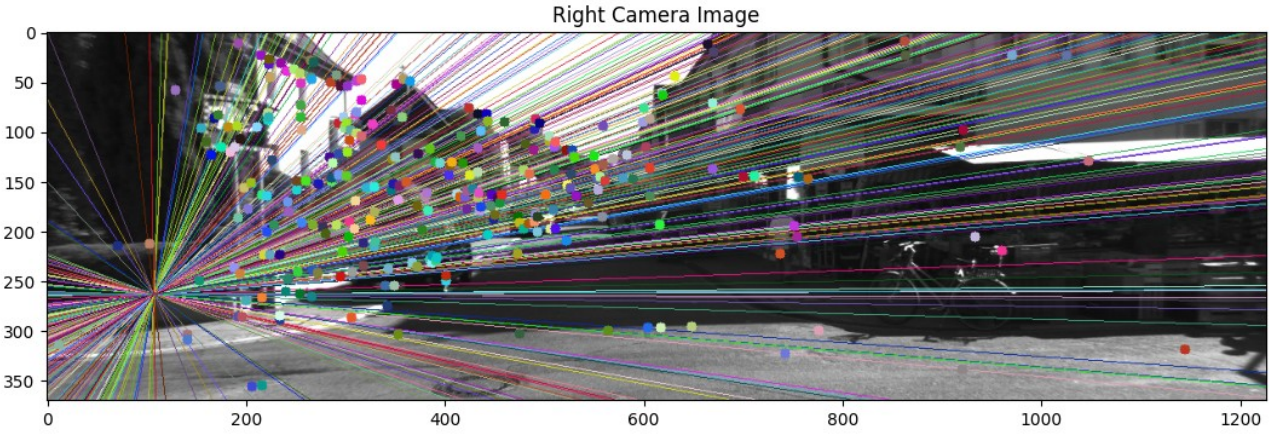
Overall Approach involved the following steps:

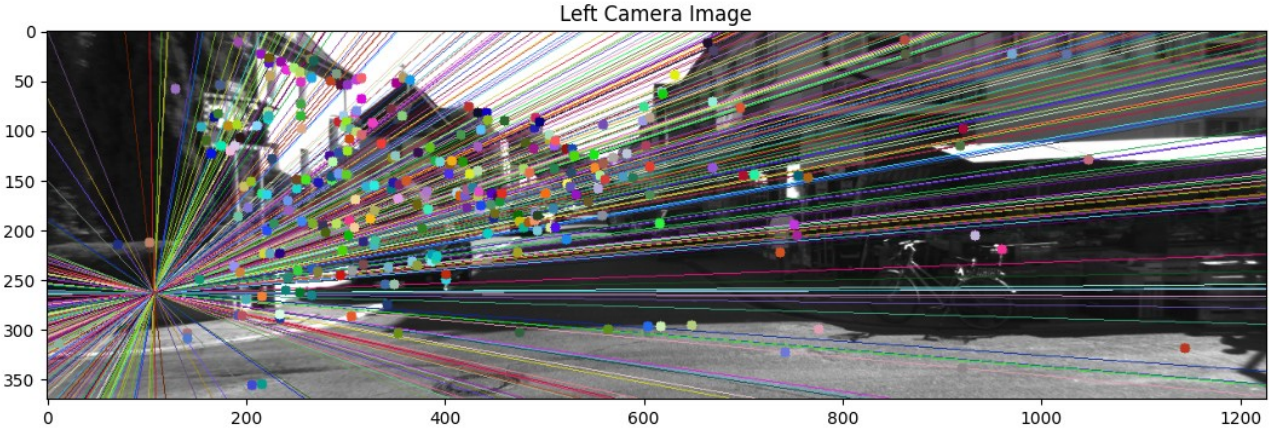
- Loading the images and fundamental matrix.
- Convert to Grayscale for feature detection and matching.
- Doing Feature Matching using FLANN and using Filter matches to extract the matched point coordinates in pts1 and pts2 array.
- Computing Epilines using the cv2.computeCorrespondingEpilines function
- Visualising the images for image1 to image2 and image2 to image1 in a top down subplot manner are images are in landscape mode.

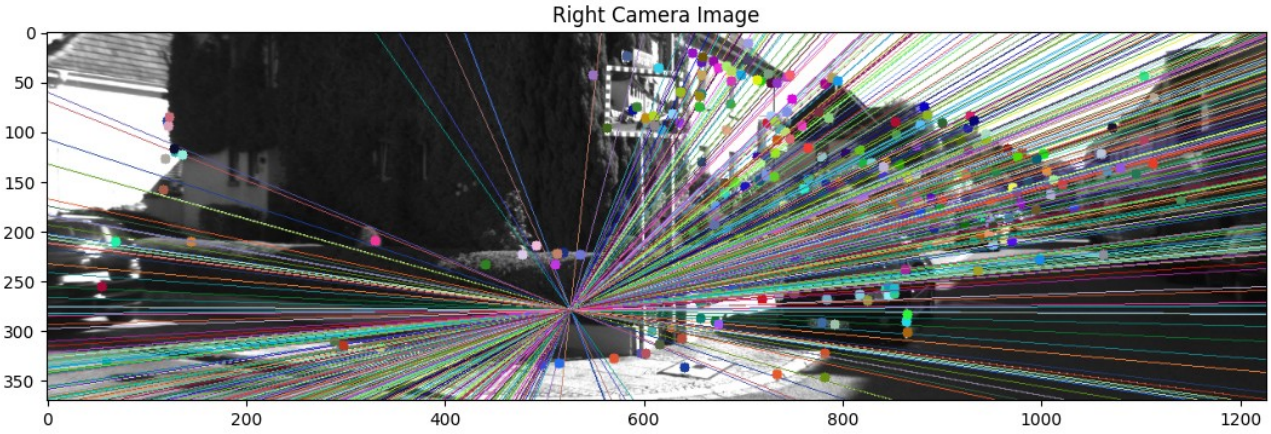
Results

Epipolar lines in both images looks as follows:









10 uniform points on Left Image and Right Image and corresponding pixels on other Images

```
Corresponding pixels (image1 to image2):
  - Point in 1: (0,-0.9864687919616699) -> Point in 2: (-0.9916645545838648,0)
  - Point in 1: (1,-1.1504178792238235) -> Point in 2: (-0.9920423122718891,0)
  - Point in 1: (2,-1.3143669664859772) -> Point in 2: (-0.992411295697386,0)
  - Point in 1: (3,-1.4783160537481308) -> Point in 2: (-0.9927714988794357,0)
  - Point in 1: (4,-1.6422651410102844) -> Point in 2: (-0.993122916534404,0)
  - Point in 1: (5,-1.806214228272438) -> Point in 2: (-0.9934655440760773,0)
  - Point in 1: (6,-1.9701633155345917) -> Point in 2: (-0.9937993776156504,0)
  - Point in 1: (7,-2.1341124027967453) -> Point in 2: (-0.9941244139615578,0)
  - Point in 1: (8,-2.298061490058899) -> Point in 2: (-0.9944406506191522,0)
  - Point in 1: (9,-2.4620105773210526) -> Point in 2: (-0.9947480857902313,0)
Corresponding pixels (image2 to image1):
  - Point in 2: (-0.9947480857902313,4) -> Point in 1: (0.31148083477408195,0)
  - Point in 2: (-0.9947480857902313,4) -> Point in 1: (0.31148083477408195,0)
  - Point in 2: (-0.9947480857902313,4) -> Point in 1: (0.31148083477408195,0)
  - Point in 2: (-0.9947480857902313,4) -> Point in 1: (0.31148083477408195,0)
  - Point in 2: (-0.9947480857902313,4) -> Point in 1: (0.31148083477408195,0)
  - Point in 2: (-0.9947480857902313,4) -> Point in 1: (0.31148083477408195,0)
  - Point in 2: (-0.9947480857902313,4) -> Point in 1: (0.31148083477408195,0)
  - Point in 2: (-0.9947480857902313,4) -> Point in 1: (0.31148083477408195,0)
  - Point in 2: (-0.9947480857902313,4) -> Point in 1: (0.31148083477408195,0)
  - Point in 2: (-0.9947480857902313,4) -> Point in 1: (0.31148083477408195,0)
```

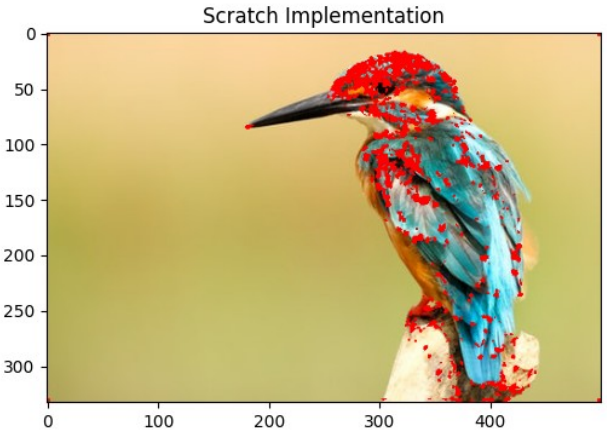
Observations

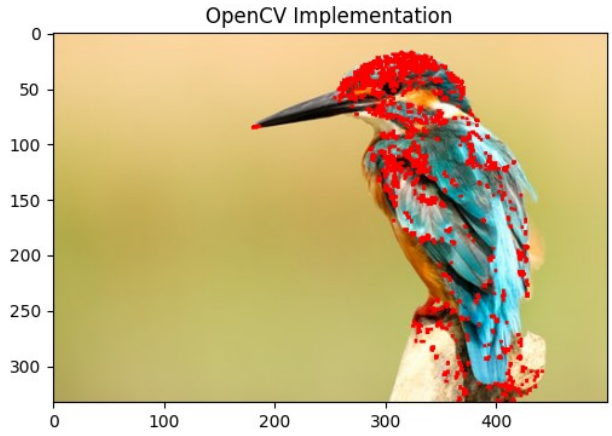
The output lists corresponding points between image 1 and image 2, and vice versa. Interestingly, all the corresponding points in image 2 have the same y-coordinate (0). This suggests that the epipolar line in image 2 might be horizontal (or very close to horizontal).

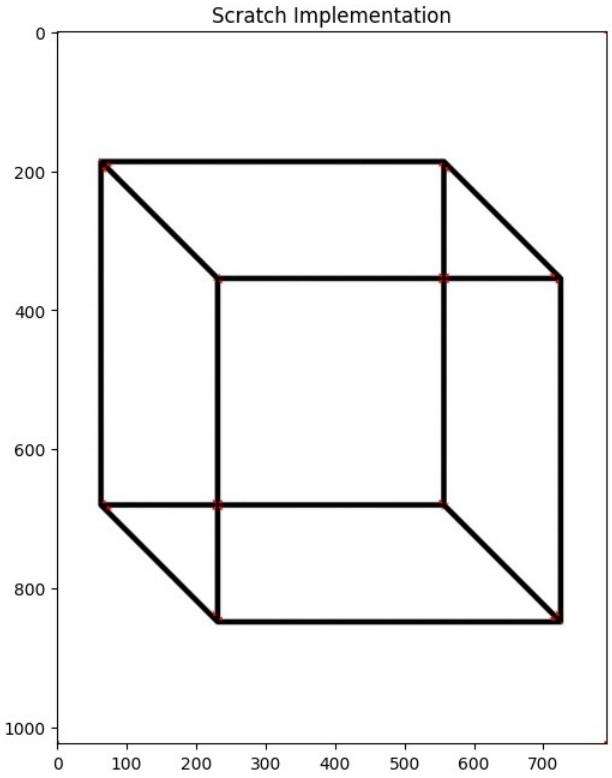
The output shows a one-to-one correspondence for most points in image 1. This is ideal, but in some cases, there might be multiple possible corresponding points on the epipolar line in the second image due to noise or inaccuracies, or some implementation inconsistencies.

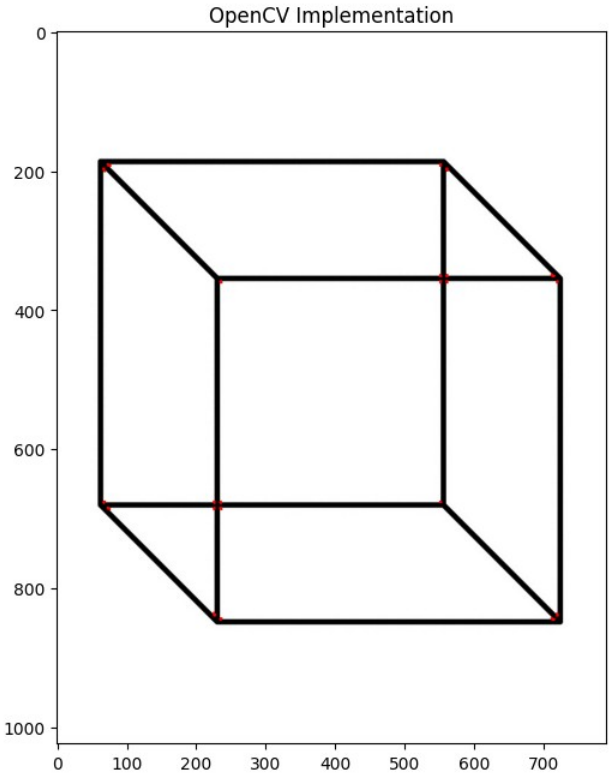
APPENDIX-I : Visual Comparison of my Harris Corner Detection with OpenCV's for all the images provided to us.

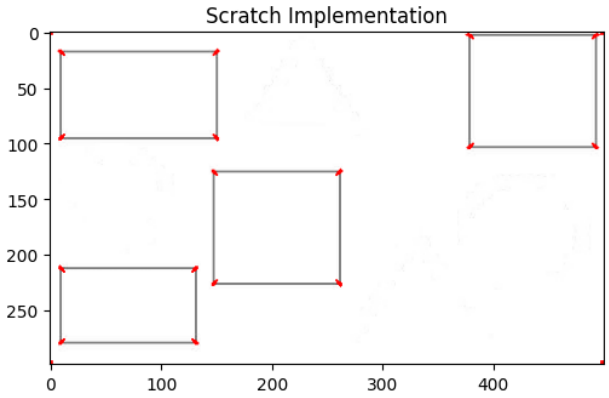
Images are taken from the following folder shared with us in the assignment document.

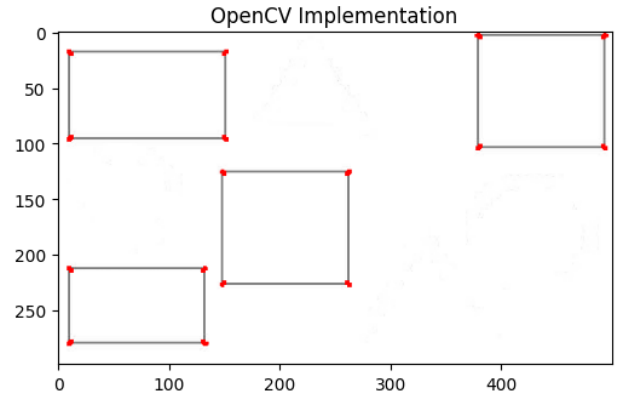


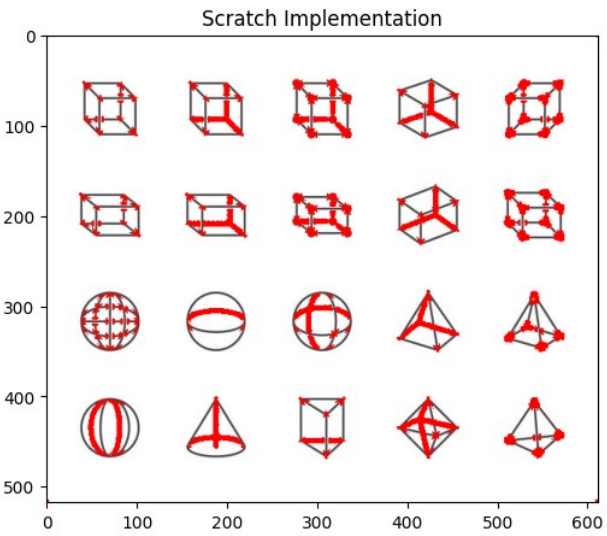


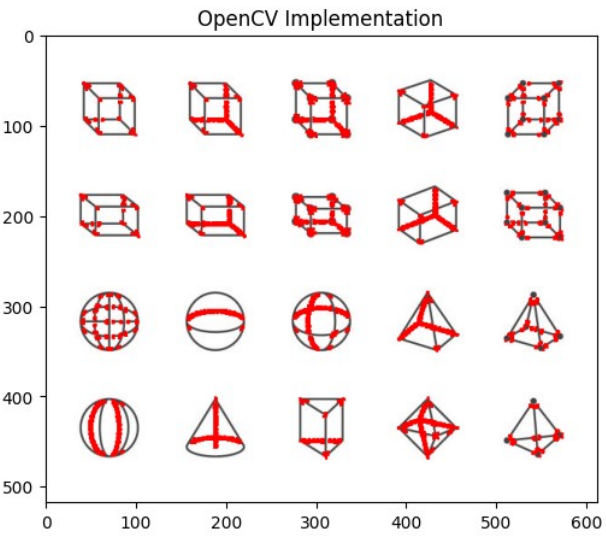


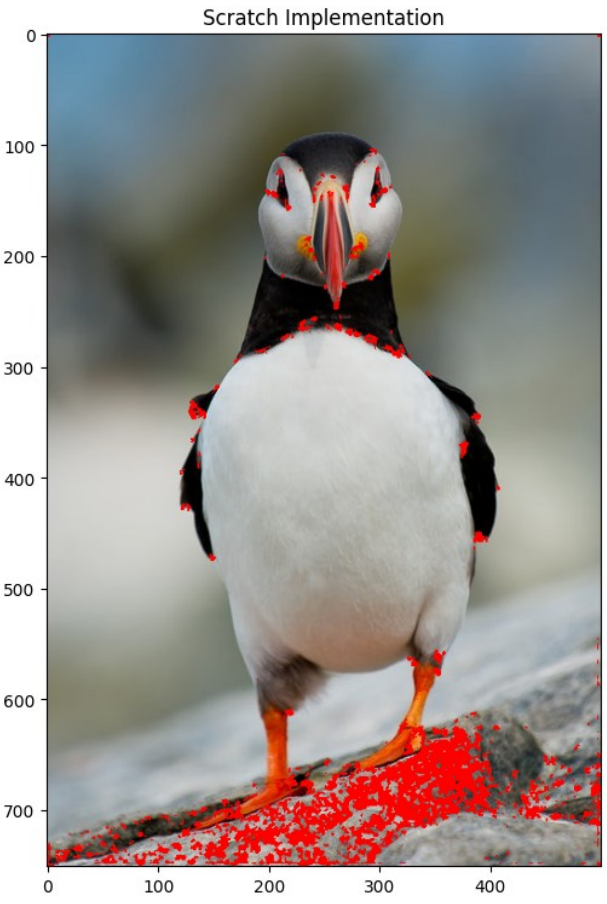


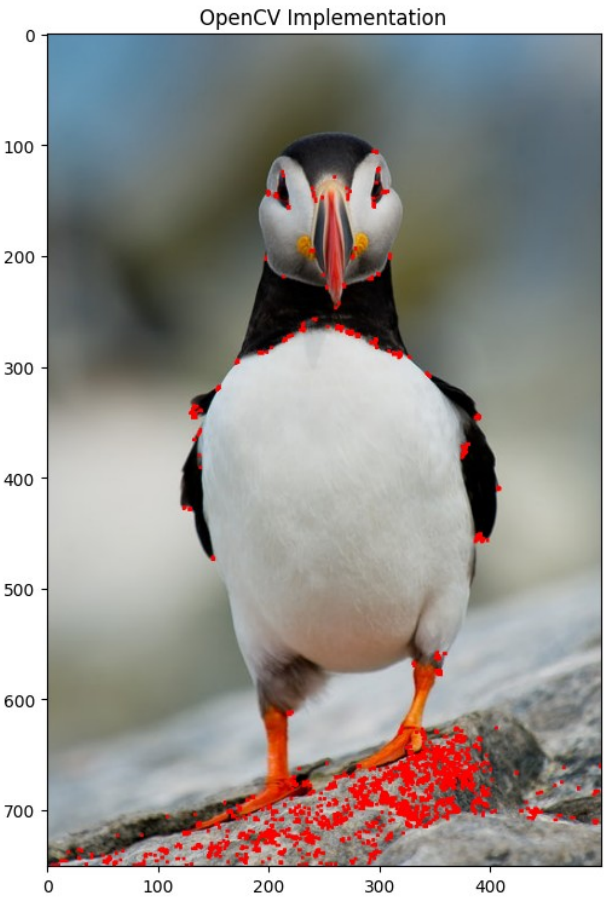


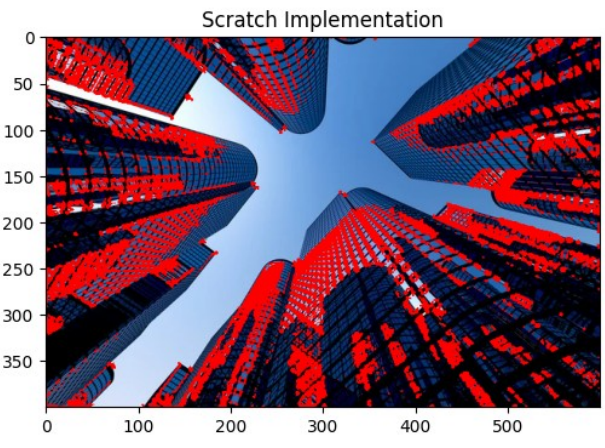


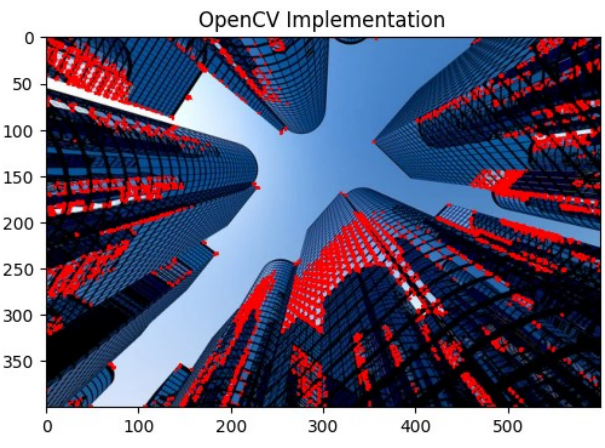

Scratch Implementation

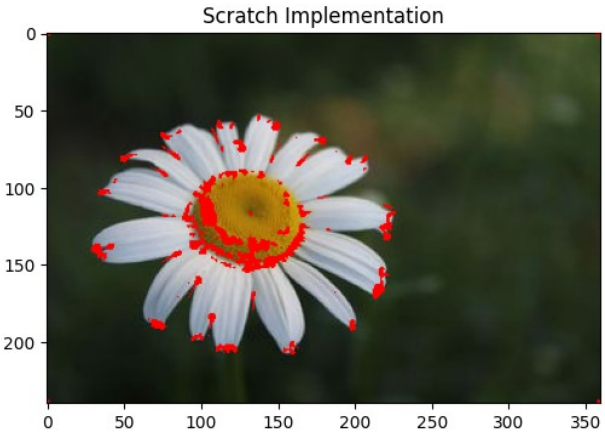

OpenCV Implementation

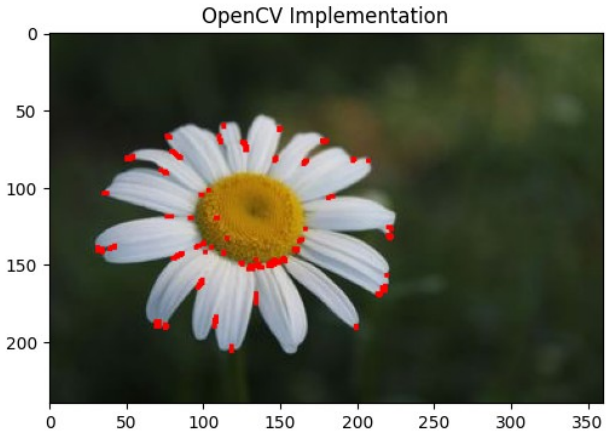

Scratch Implementation

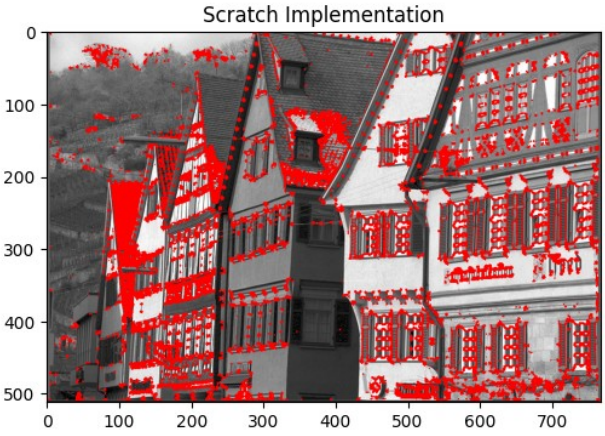

OpenCV Implementation

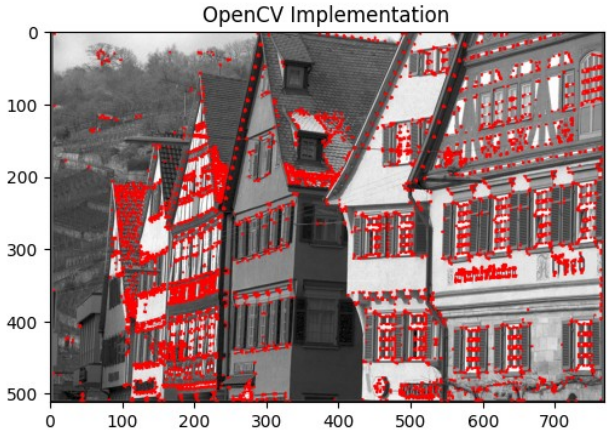

Scratch Implementation


OpenCV Implementation

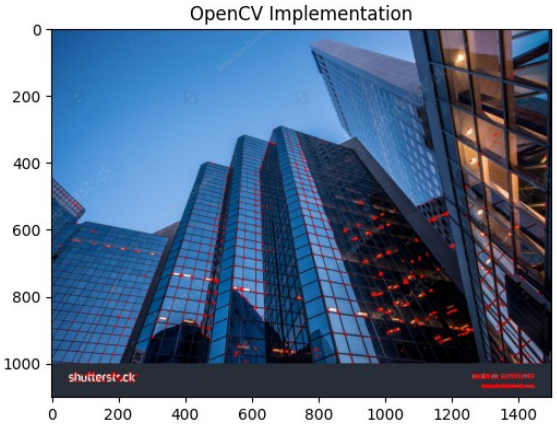

Scratch Implementation

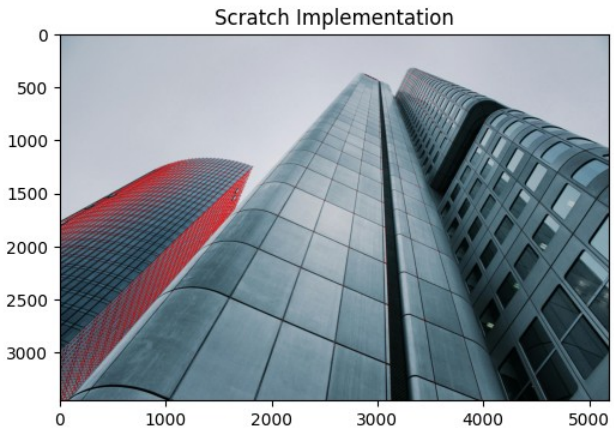

OpenCV Implementation

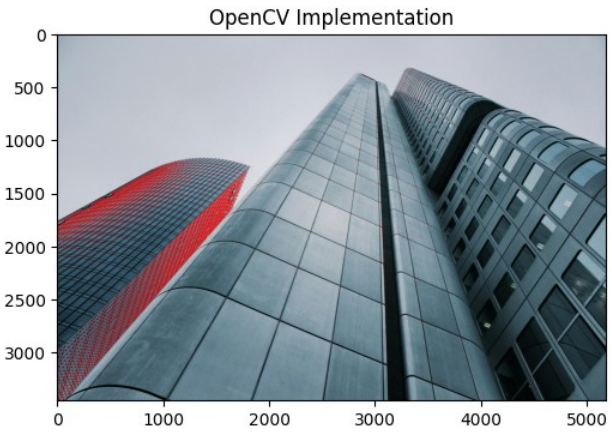

Scratch Implementation


OpenCV Implementation


Scratch Implementation


OpenCV Implementation


Scratch Implementation


OpenCV Implementation