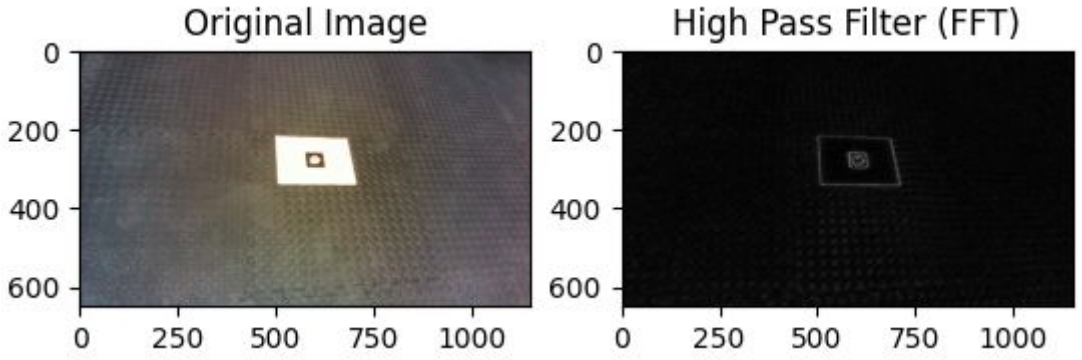
Problem 1 Detection

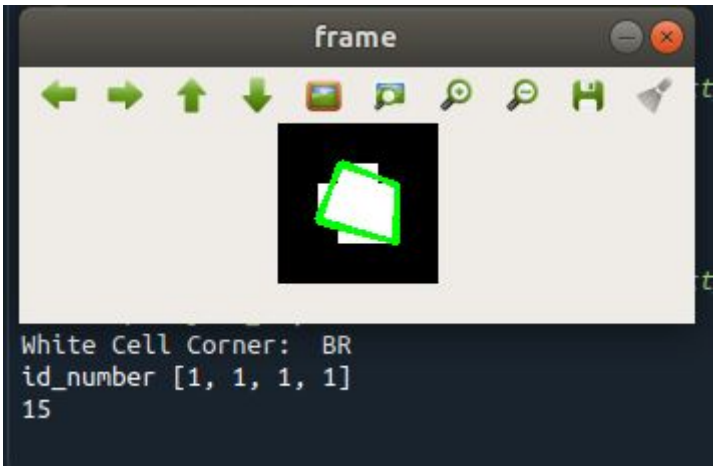
Part a. AR Code detection

I use the high pass filter in the frequency domain to detect edges of the AR tag. Applying the high pass filter removes all the frequencies below the center radius.



Part b. Decode custom AR tag

To detect the AR tag edges, I use the cv2.Canny(). After that, the contour of AR tags can be obtained by cv2.findContours(). Then, I extract the AR tag contour by using the cv2.arcLength() and cv2.approxPolyDP() to make sure getting some ‘bad shape’ perfect square. The tag can be decomposed into an 8*8 grid of squares. I assume the dimension of the grid is 80*80. Hence, the four corners of the outer grid will be 20*20. I take only 10*10 for tolerance. The outer white corner determines the tag’s orientation. And, each of the outer corners and the inner 2*2 grid corners will mark as '1' in its ID once detected as white and black as '0'. Furthermore, the outer white corner will affect the ID number as well. The diagonal side of the white corner will have the least sufficient number and increase its importance in clockwise order. In my case, I mark the least sufficient cell as '0' and increase 1 in clockwise order. Finally, with the combination of the outer and inner ID, the outer ID number multiply accordingly to the square of the inner ID number. The example is shown in the picture below.



Problem 2 Tracking

Part a. Superimposing image onto Tag

Unlike the previous example, this time the tag has to detect in a video file. I use the built-in cv2.bilateralFilter() for blurring and cv2.Canny() for edge detection. On the website of [Ref 1.], it says the cv2.bilateralFilter() has a nice property of removing noise while still preserving the actual edges.

Now, we have to warp our AR tag to let us be able to decode its tag. Unfortunately, we are not allowed to use cv2.warpPerspective() in the project. So, first, we have to calculate the homography matrix from the four AR tag corners to our desire output image. With the help of the supplementary pdf file and the previous homework, we know that the homography matrix can be calculated by solving the SVD of A matrix and rearranging its elements. Second, here is the tricky part, we get our source points' (AR tag) x, y coordinate by inversing our homography matrix and multiplying it by the destination points (desire out image). We have our warped tag image by now.

In the previous problem, we deal with the warping problem of the tag. Now, we have to warp the testudo image to the tag and adjust its orientation according to the coded orientation. The process is the same as the previous problem. After we detect the coded orientation, we can warp the testudo image to the

tag. Remember this time, we calculate the homography matrix from the testudo image to the tag. Then, I use the cv2.rotate() to rotate the testudo image.

Part b. Placing a virtual cube onto Tag

Same as the last problem, we find the tag first but this time we do not need to decode the tag. Once the tag is detected, we form a virtual cube on the tag. Therefore, we need the projection matrix to project the points. In the supplementary material provided, we know how to calculate the projection matrix. However, I am not using the entire projection matrix at once. Instead, I use the rotation and translation matrix obtained in the projection matrix. [Ref 2.] After getting these two matrices and the provided K parameter, I use the cv2.projectPoints() to project points onto the tag.

Thoughts

In Problem 2, the entire process is not quite stable. In the first, we did not use the built-in function which may have a better performance in Python. And, most importantly, noise plays a huge role in the detection process. However, I did not have the time to tackle the problem and to earn extra points for the solution.

Reference

1. https://www.pyimagesearch.com/2014/04/21/building-pokedex-python-finding-game-boy-screen-step-4-6/
2. https://docs.opencv.org/3.4/d9/d0c/group__calib3d.html#ga1019495a2c8d1743ed5cc23fa0daff8c