Project 1

Gokul Hari
Directory ID: hgokul@umd.edu

I. Problem 1

Question 1a

The task here is to detect the April Tag in any frame of Tag1 video (just one frame). Notice that the background in the image needs to removed so that you can detect the April tag. You are supposed to use Fast fourier transform (inbuilt scipy function fft is allowed) to detect the April tag. Notice that you are expected to use inbuilt functions to calculate FFT and inverse FFT.

Answer

To find the location of April Tag in a given image frame, I first computed the Fourier transformation (FFT) of the image to find the edges of the AR from the background. From the way that the FFT keeps track of the data, the amplitudes of the low frequency components end up being at the corners of the two-dimensional spectrum, while the high frequencies are at the center [1]. So, the spectrum is shifted to place the zero frequency at the center of the plot. The magnitude of this spectrum is shown in figure 3-b. Since edges of the AR tag are high frequency components, an appropriate high pass filter must capture the edges of the AR tag from the image frame. This can be achieved by blocking the low frequency components in the center of FFT spectrum as shown in figure 3-c, This high pass filter was designed by applying a circular binary mask with zeros in center and multiplying it with the FFT spectrum. The inverse FFT of this filtered spectrum is shown in figure 3-d , where I notices the edges are much prominent. I estimated the contours of this filtered image and selected the contour with maximum area to find the rectangular white space region around the AR tag. This is shown in 3-e. Finally, I found the inner contour of this rectangular white region to find the corners of our AR tag. I warped this AR tag from the image frame to a reference/world frame of 128×128 pixels as displayed in figure 3-f

Question 1b

You are given a custom AR Tag image, as shown in Fig. 1, to be used as reference. This tag encodes both the orientation as well as the ID of the tag.

Answer

To perform the encoding, I was required to split the reference marker to a equally spaced 8×8 grid and choose the inner most 4×4 grid, removing the black padding. However, as seen in figure 4-b, cropping directly to the inner 4×4 grid results in cropping out some of the white pixels and does not guarantee proper location of white blocks for reliable encoding results.

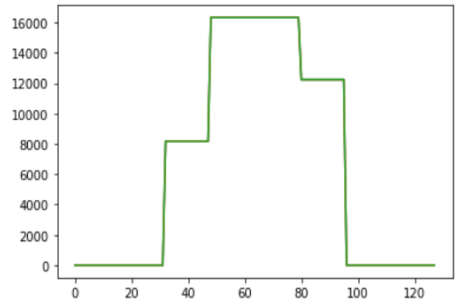

Fig. 1. Pixel intensity along x axis

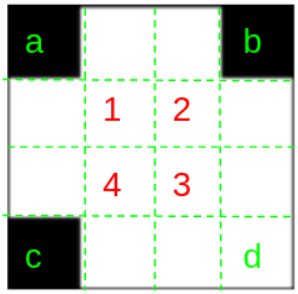

Fig. 2. Orientation and Decoding

So, I followed a different cropping mechanism that detects the location of the white blocks.

Cropping: To crop and remove the black paddings in the tag, I plotted the sum of all pixels distributed along the x and y axes of the AR tag. This is shown in figure 1, where I noticed that there is a steep rising intensity of pixels from zero at 30 and a falling intensity towards zero at 95 in the plot along x axis. Similary, the steep rising and falling pixel intensities along y axis is also noted. These locations signify the boundaries where the white blocks exist and the zero (black) regions can be cropped. By cropping the Tag from these regions, I remove the black padding with greater accuracy than merely choosing the inner 4×4 grid for decoding.

Rotating: Now that the black paddings are cropped, the inner white region is split to a equally spaced 4×4 grid. This grid is plotted on the reference marker is shown in figure 2. The color pattern of the corners grids from *a-d* are used to formulate the condition for an AR tag to be in reference position, ie corner grid *d* must be white. Any given AR tag will be rotated according to the position of the white block in one of the 4 corners.

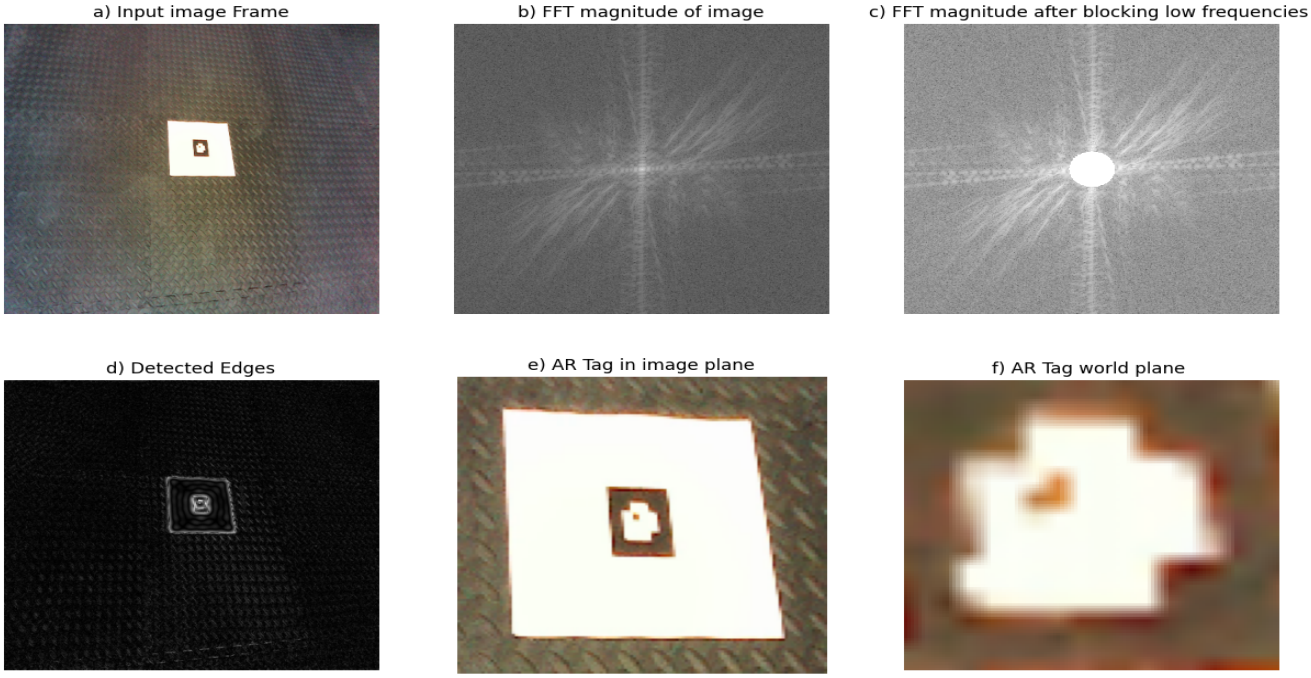

Fig. 3. Output of Problem 1a

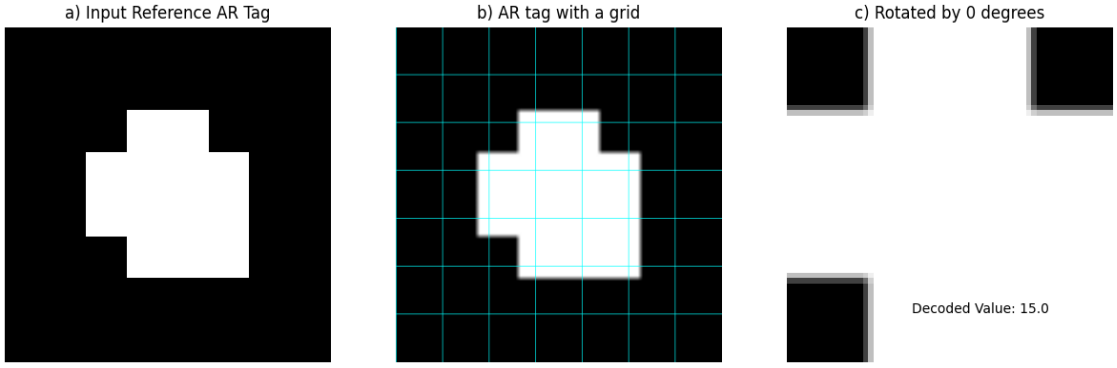

Fig. 4. Output of Problem 1b

Decoding: Once the tag is rotated to the reference position, the inner 2×2 grid is considered for decoding. Block numbered 1 is the least significant bit while the block numbered 4 is the most significant bit. A white block holds a HIGH level (1) and a black block holds a LOW level (0). In case of our reference marker, the Decoded output is (**1111**) which in decimal terms is **15.0**. The final output of my encoding scheme is shown in 4c.

II. Problem 2

Question2a

Pipeline Definition

The tag tracking pipeline has 3 major components:

- Tag corner detection
- Tag orientation estimation
- Template Alignment

Tag Corner Detection: For a given image frame in the video, we apply binary thresholding to reject the background, and estimate the contours in the image. I select the contours that fulfill a *Tag-contour* condition as our Tag contours. A contour is considered as a *Tag-contour* only if a contour consists of an inner child-level and only one parent-level contour. Any contour that has more than one parent level is rejected. This condition results in near-perfect Tag detection, rejecting possible outliers that can occur due to white background objects, as shown in figure 5. Next, I estimated the corners of the Tag-Contour using the Douglas-Peucker Algorithm [2], which can be implemented using cv2.approxPolyDP function in opencv. By tuning the contour perimeter based hyperparameter passed to this function, I was able to obtain 4 exact corners for every estimated Tag contour. The number of 'bad' frames (image frames that return incorrect number of Tag-contours **and**

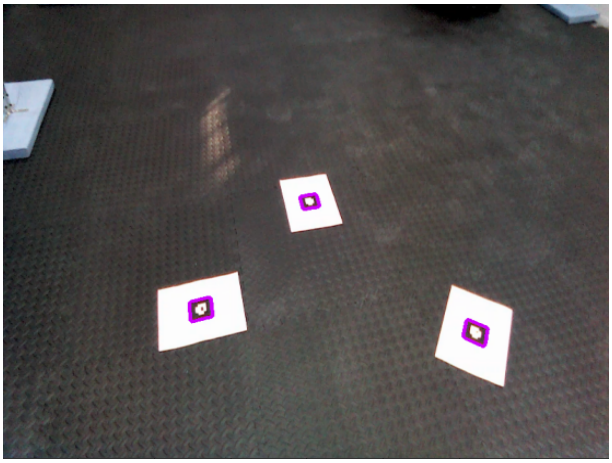

Fig. 5. Contours detected in multiTag video frame - 795

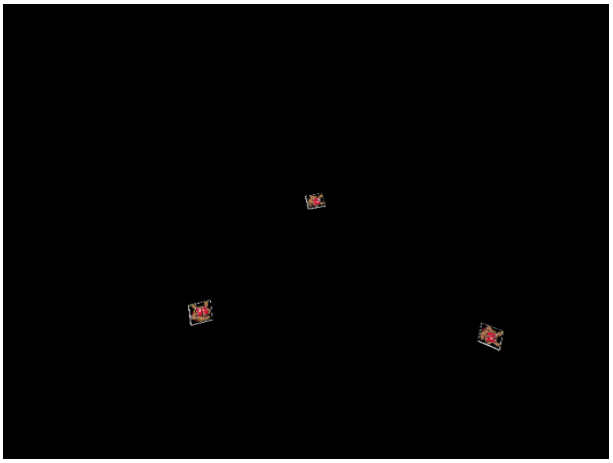

Fig. 6. Warped Testudoes in multi tag video frame-795

incorrect number of Tag-corners per Tag-contour) were lesser than 10 frames in all three single tag videos using this Tag corner detection method. To ensure smoother performance, the incorrect contours resulted in the bad frames were replaced by the correct set of contours estimated in the previous frame. An example of the detected Tags from the multiple tag video (frame no. 795) is in figure 5. This frame is chosen for this section of the report to showcase the performance of our tag detector despite the presence of background white outlier regions.

Tag orientation estimation: I followed the same decoding and orientation estimation procedure followed in section I. Only the first frame of the videos were decoded (last frame in case of multi-tags video), while the orientation of the tag is estimated for every frame of the video. This is necessary because, once the testudo template is aligned in the correct direction with reference to the reference marker in the first frame, the testudo loses the alignment as the camera angle changes as the video proceeds. In order to consistently maintain the alignment of the testudo template, in the correct direction, the testudo template's orientation is corrected every frame before superimposing on the top of the AR. Even though corner detection of multi tag video was possible with great accuracy, estimating the correct orientations throughout the video was not good, and so, in case of multi tag video, the orientations were not tracked throughout the video (tracking them results in spinning of testudo template). However, the template alignment and decoding was done in the last frame of the multi-tag video and it is shown in figure 7

Superimposing the template: To superimpose the template on every image frame, the homography *H* between the corners of the tag $C4_{tag}$ in these images and a chosen reference frame window $C4_{ref}$ of shape 128×128 is estimated. The testudo template is also resized to the same reference shape of 128×128. Using a custom built warping function, I warped the testudo template with H^{-1} to obtain the warped testudo output in the same shape as the input image frame, where all elements are zero except for the region of the tag corners, which contain

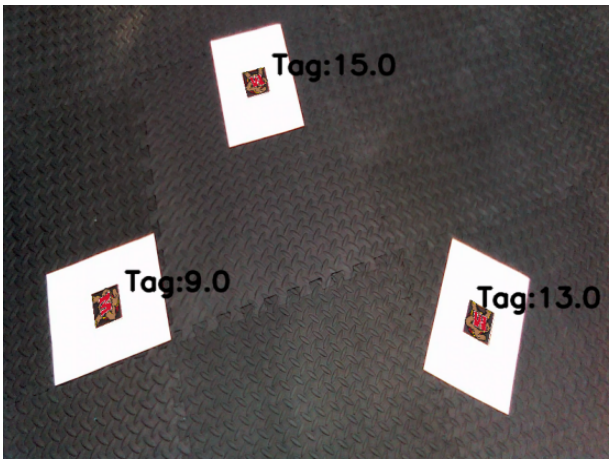

Fig. 7. Template Superimposed in multiTag video- last frame (1011)

the pixels corresponding to the testudo template. This is shown in figure II This warped testudo output and its binary mask was used to paste the template on the tag in every image frame of the video. These results are in folder Problem2a here. The final result of the last frame of the multiple tag video is shown in figure 7

Custom Warping function: To warp the image, I built a forward warping function. To explain this, let's term the source image frame coordinates as [a,b,1] (1 is padded here since image frame is only 2 dimensional) and destination/world frame coordinates as [x,y,z]. For every pixel from the source image located at [a,b,1], we obtain it's corresponding destination coordinate by applying the a 3×3 transformation matrix (H or H^{-1}, depending on the usage). Since the destination is also an image (of shape 128×128), I obtained the destination *image* coordinate as [x',y',1]. where x' = x/z and y' = y/z. Warping is achieved by copying the source pixel values from coordinate (a,b) to corresponding destination pixel coordinates at (x',y').

However, in forward warping, when the destination image area is bigger than the source image region to be warped,

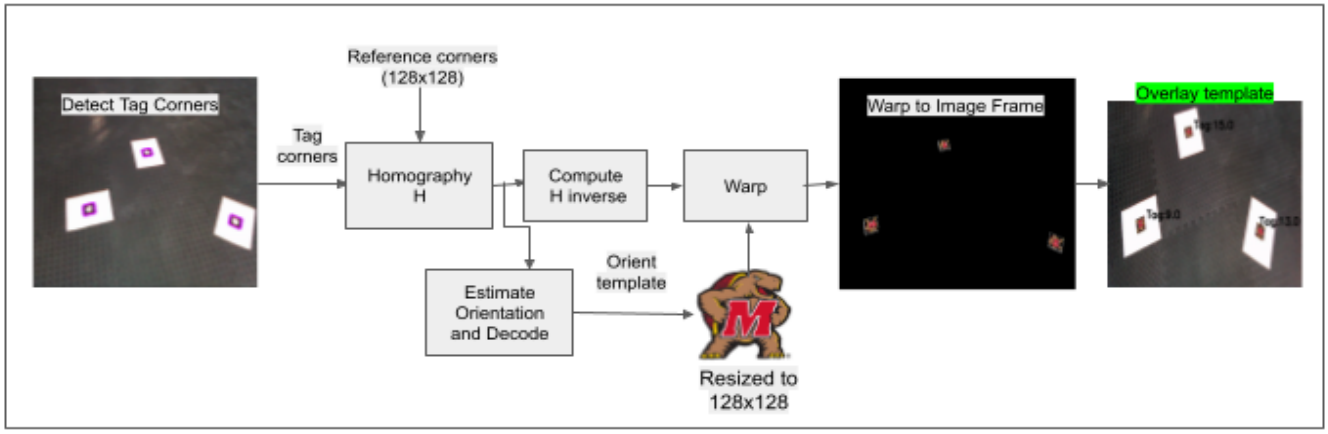

Fig. 8. Pipeline for Problem 2a

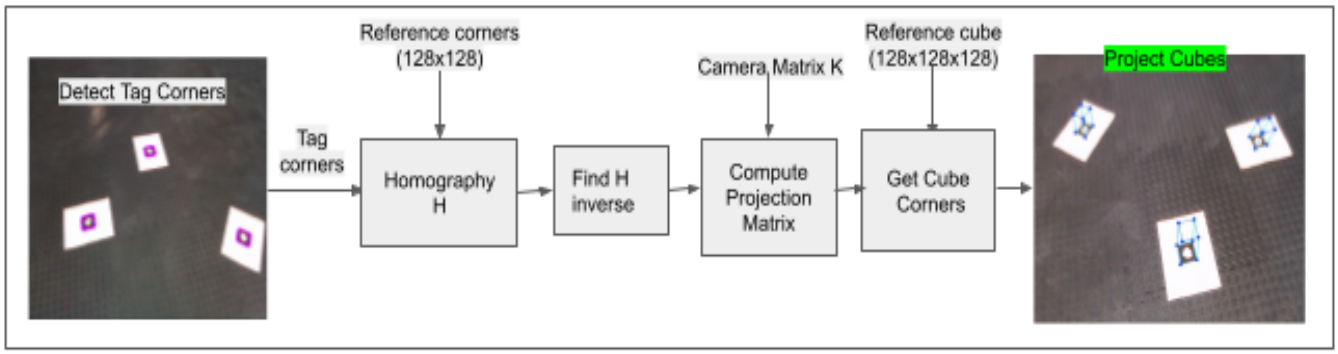

Fig. 9. Pipeline for Problem 2b

the resultant warped image has black 'holes' in it since these regions were left empty while copying the pixels as above. To solve this, as I did not have time to implement a warping function with bilinear interpolation of pixels, I chose a smaller sized destination frame (say 20×20) to obtain a relatively 'hole'-free warped output and resized to a larger frame size, with cv2.resize(), for using in the detection pipeline, as cv2.resize() function performs pixel interpolation in itself.

The overall block diagram of the pipeline to superimpose the template is shown in 8

Question2b

AR Cube Projection

To project a cube in the detected tag, the template superimposing pipeline in 8 is modified as shown in figure 9. After detecting the tag corners $C4_{tag}$, and computing the homography **H** between $C4_{tag}$ and an arbitrary square reference/world frame of size 128×128 with corners $C4_{ref}$, I estimated the projection matrix P for this homography H for every image frame, with the constant camera matrix K. The steps to compute projection matrix is given as follows:

- Define h1,h2,h3 as the 3 columns in the homography matix H

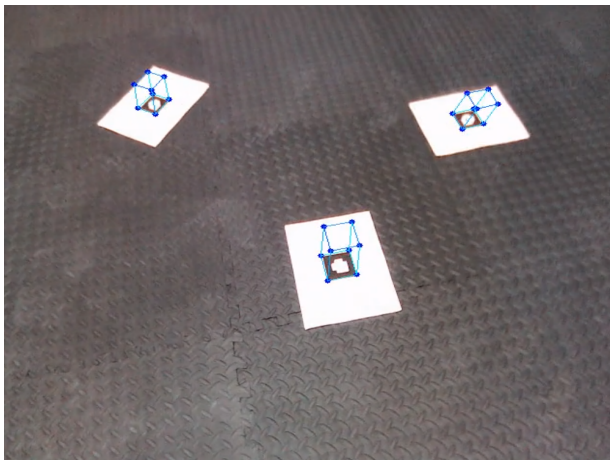

Fig. 10. Cubes projected from tags in the multiTag video frame

- Compute scale factor λ as $(\frac{(\left|\left|K^{-1}.h1\right|\right|+\left|\left|K^{-1}.h2\right|\right|)}{2})^{-1}$
- Compute $\tilde{B} = \lambda K^{-1} H$
- Compute B = $\tilde{B}(-1)^{|\tilde{B}|<0}$, i.e. B = $-\tilde{B}$ if determinant of $\tilde{B}$ is negative, else B = $\tilde{B}$. b1,b2,b3 are defined as the column vectors of B

- Calculate the rotation vectors r1 = b1, r2 = b2, r3 = $h1 \times h2$ (where $\times$ here, denotes vector cross product) and the translation vector t = h3.
- Collate the rotation and translation vectors to form the projection matrix P

To draw the cube, we define the cube coordinates of the reference/world frame ($C8_{ref}$) and multiply these cube coordinates with the Projection matrix P, to obtain the cube coordinates in the image frame ($C8_{img}$). By drawing the lines in the image that connect these cube coordinates, we draw a cube in the video that projects towards the camera. The video results are in folder Cube here The projected cube in one of the frames in the multiple tag video is shown in figure 10

The overall block diagram of the pipeline to project cubes on the April Tag is shown in 9

REFERENCES

[1] http://www.dspguide.com/ch24/5.htm

[2] https://en.wikipedia.org/wiki/Ramer%E2%80%93Douglas%E2%80%93Peucker_algorithm

[3] https://drive.google.com/drive/folders/12ir5Z2z3xvI6HCqRao5lhdgwi1wsK5N0?usp=sharing