Shon Cortes
ENPM 673 - Perception for Autonomous Robots:
Project 1 - AR Tag Detection, Decoding, Tracking/Projection and Homography

Detection:

The detection of an AR Tag required the use of the numpy, cv2, matplotlib, and scipy libraries. The program can be summarized by the following:
The input image is of an April Tag on a dark floor with a repetitive textured pattern. The detect method takes the image as an input and first converts it to a grayscale image. The gray scale image is then thresholded to return a binary image. The binary image is blurred and the fft is performed on the binary image. The fft converts the image into the frequency domain with the low frequency components in the corners. The converted image then has the low frequency components shifted to the center. A circular low pass filter is then applied to filter out the high frequency textured background from the ground. This filtered image can then have the low frequency components shifted back to the corners. After shifting back to the corners, inverse fft can is performed to return the isolated AR Tag image with the background filtered out. The final output shows the major steps in a plot. This process brought up a few interesting challenges. I had some trouble using the numpy and scypi libraries to use the fft and ifft functions. They would return magnitude spectrum that were not aligned with the expected output. It is suspected that there is some incompatibility with opencv's later versions and these functions. Figure 1 shows the final output of the detection program.

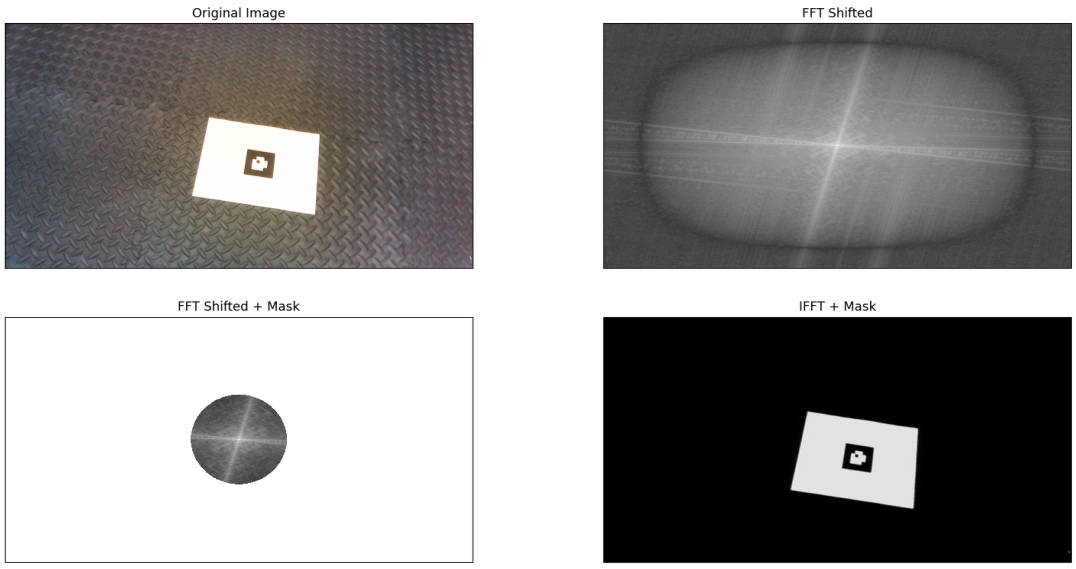

Figure 1: Shows the process of isolating the AR Tag from the background.

Decode:

To decode the AR Tag I used the numpy and cv2 libraries. The process involved taking the outermost corner of the isolated tag that is white to be the bottom right corner for an upright orientation. The inner square area of the tag tells us the tag ID. The inner corner square opposite of the outer white square describes the first number of the tag id, following the inner squares in a clockwise orientation the rest of the tag ID can be found. White indicates 1, black indicates 0. Figure 2 shows the reference tag tested along with its orientation and outer edge coordinates. For this tag the coordinates are the dimensions of the image. In the following sections I isolate the tag similar to in the detection program and retrieve the AR Tag corner coordinates to compute homography. The decode.py program takes an input image of an April Tag. It is first segmented into an 8x8 grid and the outer 2 most squares of the grid are removed to get a final 4x4 grid and a total of 16 segments. These segments are then thresholded to determine if they are white or black. The corners are then checked to determine the orientation of the Tag. The orientation is used to then determine the AR Tag ID. The tag ID and its edge coordinates are then displayed on the image.

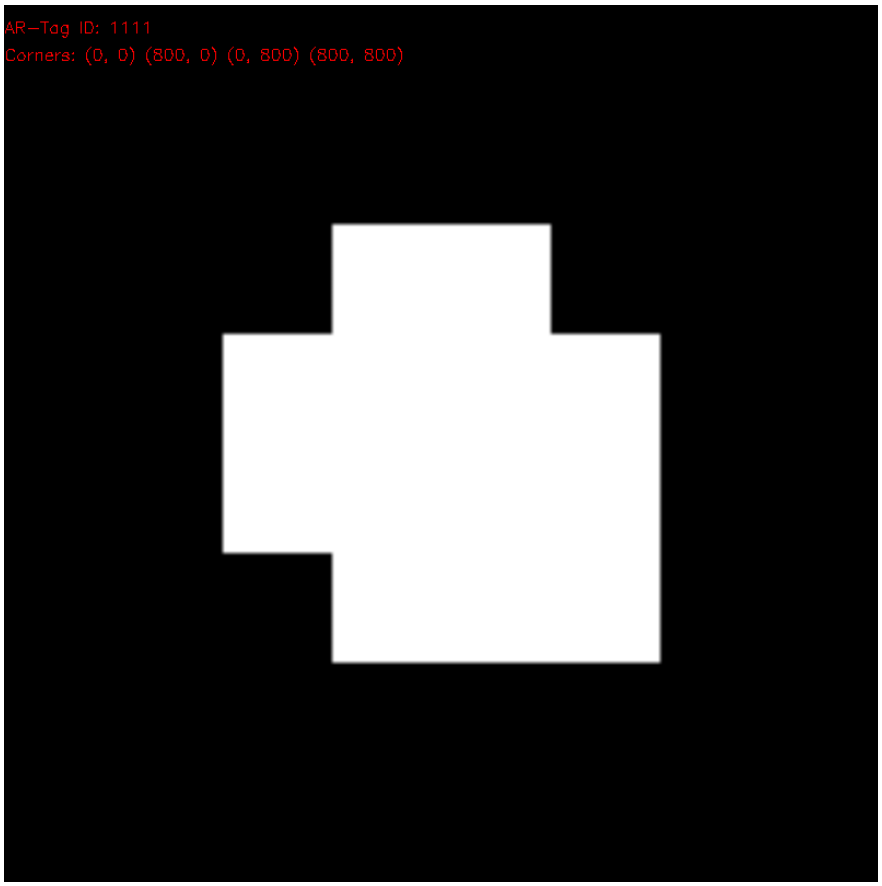

Figure 2: Shows the output from the decode.py program.

Tracking:

The tracking of an AR Tag and projection of the testudo image onto the tag involved combining the processes of the first two programs. I first needed to isolate the AR Tag to then decode it, telling me its coordinates and orientation. The tag orientation told me how to orient the testudo image and the coordinates were then used to compute the homography between the AR Tag and the testudo image. Using the homography, the testudo image is then projected onto the AR Tag in the frame. This posed a few interesting challenges. When finding the contours in the frame, I needed to filter out any contours that didn't have a parent or child. This allowed me to get rid of any additional noise still being detected. I was left with the paper and the AR Tag as contours however in order to isolate the tag I needed to threshold out the paper by filtering out the largest area contour detected in each frame. Another issue is that I took the tag corner coordinates to be the maximum and minimum values of the x and y coordinates for the filtered contours (x_max, y_min), (x_min, y_max), (x_min, y_min), (x_max, y_max). This defined a perpendicularly orientated square. When the frame shifts around the tag, the tag skews at an angle requiring the corner coordinates to not be the previously mentioned pairing. This is still an issue and needs more time to be dealt with. For the time being, if the program gets value error during its calculations it uses the previous frames values for the projection. This helps deal with when the contour filtering, filters out the AR Tag, doesn't detect anything at all, or can not decode the tag for orientation and ID. Future improvements will address how to deal with the frame shifting around the AR Tag.

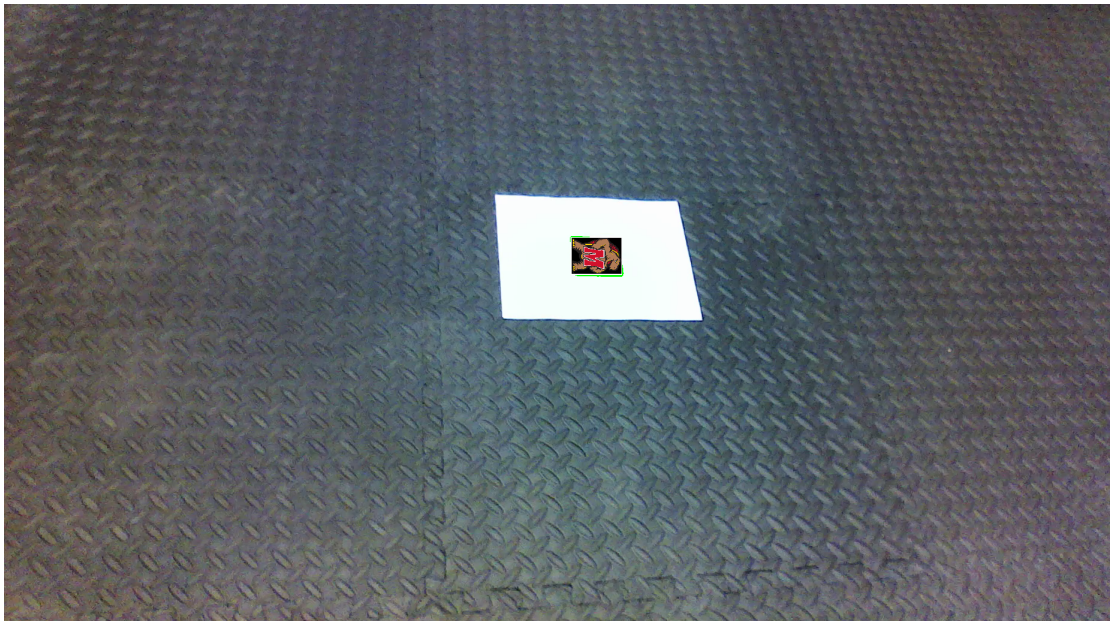

Figure 3: Shows a screenshot of the tag being projected onto the AR Tag in a video frame.

3D Cube Projection:

Projecting a 3D cube onto a 2D frame follows a similar process to the previous section. The AR Tag is detected, decoded, and a homography is calculated between the tag and the camera frame. I then needed to use the camera intrinsic parameters to define the K matrix which was provided for this assignment. To calculate the projection matrix required to project a cube to the tag, I needed to find a scaling factor, lambda, along with a rotation matrix and translation vector. Using these parameters I needed to then multiply the cube dimensions by the projection matrix to project the cube onto the AR Tag. I had trouble completing this part. I was able to get a square projected but the other planes of the cube did not show. Figure 4 shows the output I was able to achieve. Further understanding of how to apply the projection matrix for this application is needed. In future work I will improve on the process to achieve the proper cube projection.

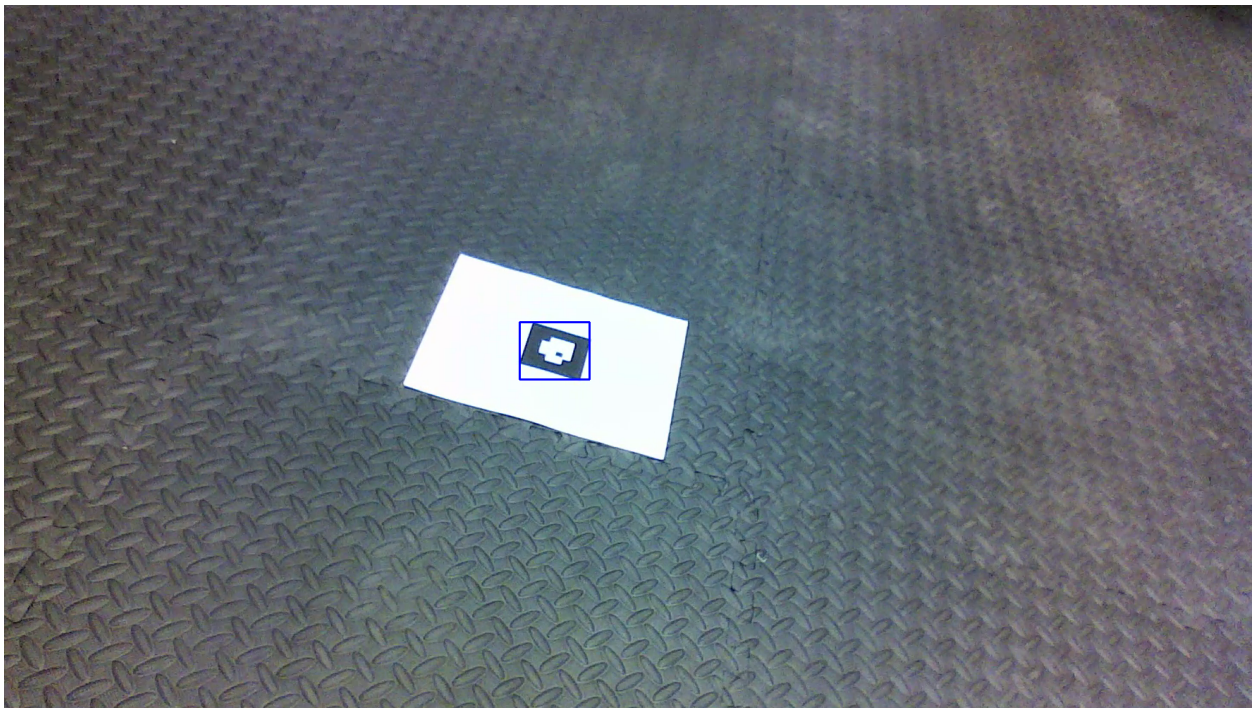

Figure 4: Blue plane being projected onto the AR Tag, Additional planes were unable to be projected to form a cube.