UNIVERSITY OF MARYLAND

18 56

Project 1

Perception for Autonomous Robots

March 2, 2022

Instructors:
S. Charifa

Student:
Jerry Pittman, Jr. (117707120)

Semester:
Spring 2022

Course code:
ENPM673

Contents

List of Figures

1 Generated Videos

Videos generated for the project are located here and are also hyperlinked in their appropriate sections.

2 Problem 1: Detection

2.1 Part 1: AR Code Detection

I spent a good amount of time trying to find a good frame that was not blurry and had the AR tag in a "good" orientation for function running and then determined that I would run the various functions on the entire video in order to make output videos demonstrating the outputs successful. I determined to use frame 133 as a test frame for creating images of the various functions used for this report to supplement the generated videos. I was unable to use the data from the Fast Fourier Transform (FFT) for future parts of the project (determining contours, tag coordinate determination, tag decoding, etc.) but was successful in it determining the location of the AR tag in the frame based on the gradients detected. I used the methodologies of [8] for making a plot that compared the images.

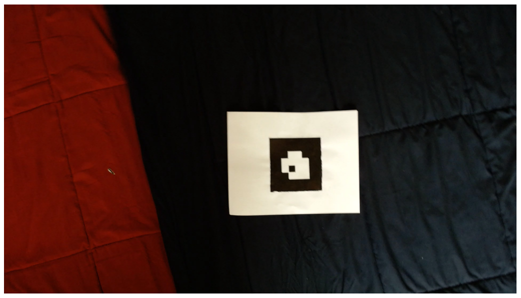

Figure 1: Frame 133 unedited.

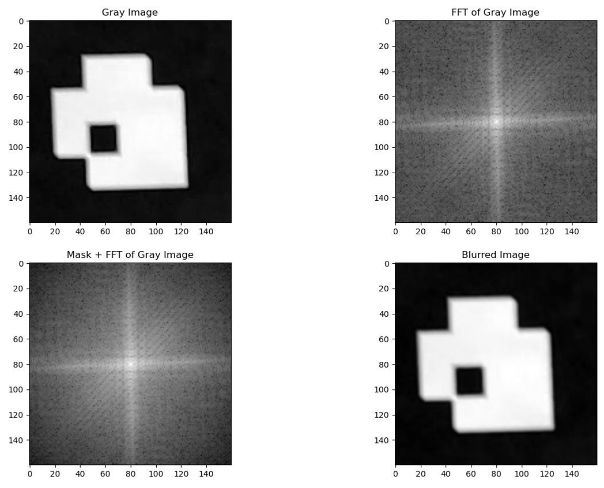

Figure 2: Frame 133 with FFT/IFFT for AR Tag detection.

2.2 Part 2: Decode custom AR Tag

For determining the contours of the AR tag I used the *opencv* functions threshold, and *findContours* after converting the image to grayscale and performing a *medianBlur*. I used the opencv functions *drawContours* and *goodFeaturesToTrack* in order to mark up the image with dots on the corners and contour lines on the contours in the video.

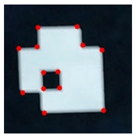

Figure 3: Frame 133 corners marked

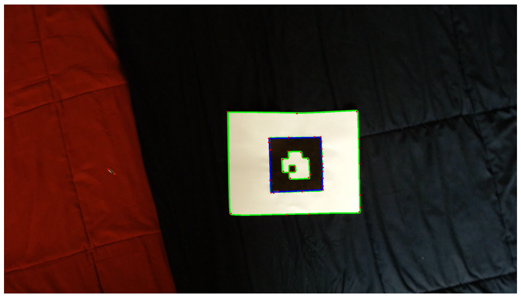

Figure 4: Frame 133 with Contours

I then used the *approxPoly* function to find the corners based on the contours of the AR tag. I also cropped the image to remove the background and contours due to the paper by enumerating through the contours found and removing contours that had a parent or child. I had difficulty doing this using recursively code and was manually cropping the image to remove the background but found [8] methodology to be more effective and clean and repeatable for any other starting video vice only for this one particular video.

Using the contours found, I found the polygon points and used those in determining the dimension of the to set the warped image to.

The corners of the tag in frame 1 are:

$$[876, 297], [1056, 342], [1016, 520], [833, 475]$$

.

I then conducted Homography using the corners found and the solved dimensions. For computing the Homography matrix, used equation 7 of [7] and the *SVD* function of the numpy library similar to what was conducted for Homework 1.

The Homography Matrix is as follows:

$$H = \begin{bmatrix} -1.051e-03 & 2.539e-04 & 9.964e-01 \\ 2.710e-04 & -1.084e-03 & 8.456e-02 \\ 4.507e-08 & -8.565e-08 & -1.103e-03 \end{bmatrix}$$

I then inverted the Homography Matrix for conducting a forward warp of the image/frames. I used the desired dimensions and using the numpy array *ravel* and *indces* functions to create a matrix of zeros. I then mapped the original image/frame into the size desired.

For decoding the AR tag, I took the 8x8 grid of the AR tag (which includes the black border) and cut it into a 4x4 grid. I then put binary text onto the 2x2 grid with a "1" representing white

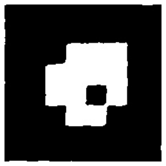

Figure 5: Frame 133 warped

and "0" representing black. In order to decode (i.e. know which cell is the least significant bit) I need to determine which corner of the 4x4 grid was white. Based on this orientation I could then concatenate a string of binary digits in the clockwise direction. The AR tag was decoded to be: 1110. I used the python embedded function of *int* to then convert that value to an integer: 14.

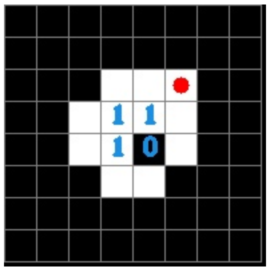

Figure 6: AR Tag Decoded and Labeled

3 Problem 2: Tracking

3.1 Part 1: Superimposing image onto Tag

First thing I needed to do was to recursively determine the orientation of the AR tag, which was performed as part of Problem 1.b in my *decodeARtag* function. I then used the *opencv* function of *rotate* in order to have testudo (provided on Canvas and located [6]) lined up based on the current orientation of the AR tag. I stuck to cardinal headings and rotations of 90 Clockwise, 90 Counter-Clockwise, and 180 degrees for simplification. The video is glitchy for this reason at times. For superimposing the *testudo.png* image onto the AR tag, I repeated the process from Problem 1.b for computing homography and warping of the image. The difference was that I need to determine the original shape of the *testudo.png* image first and needed to rotate prior to warping. A difficulty I had was to ensure only the *testudo.png* was seen on the paper in the video and not both it and the AR tag. I used a methodology of [3] and [4] and first replaced the AR tag with a black square then superimposed *testudo.png* over the black square.

3.2 Part 2: Placing a virtual cube onto Tag

I determined that I wanted to make the size of the cube to be the same dimensions (length and width) of the AR tag given the frame. I decided to do this vice setting discrete value of pixel dimension to try and ensure was as close to a cube as possible. I then solved for the homography of cube based on these dimensions and the same methodology/function used for Problem 1.b and Problem 2.a. I then referred to the class notes [5], Equations 15, 16, and 17 of [7], and the K matrix provided by the Teaching Assistants (TAs) to come up with the Projection matrix to be $P = K \cdot B$ where:

$$B = [b_1, b_2, b_3] = [r_1, r_2, r_3, t]$$

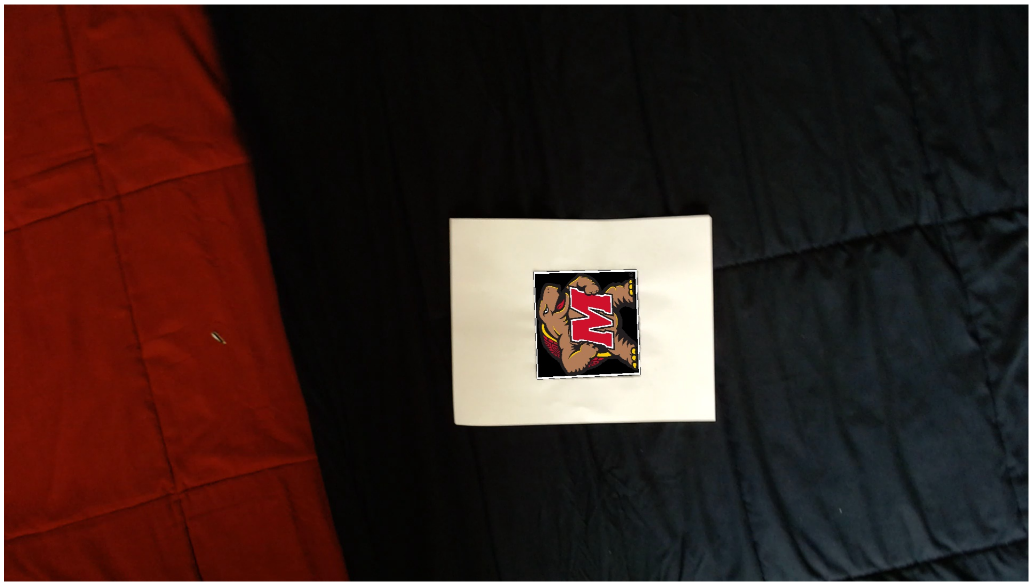

Figure 7: Testudo.png Superimposed over AR Tag in Frame 133

where

$$r_1 = \lambda * b_1, r_2 = \lambda * b_2, r_3 = r_1 \times r_2, t = \lambda * b_3$$

and

$$\lambda = (\frac{||K^{-1} \cdot h_1|| + ||K^{-1} \cdot h_2||}{2})^{-1}$$

To determine the Projected cube corners was to set the dimension of the cube to be equal to the dimensions of the AR tag found in Problem 1.b ($185 pixels$), solve for the Homography and Projection Matrixes of the cube and then conduct the math from [5] to project those points into the camera frame. The Cube Homography Matrix is as follows (ended up being the same as original Homography since inputs are the same):

$$H_{cube} = \begin{bmatrix} -1.051e-03 & 2.539e-04 & 9.964e-01 \\ 2.710e-04 & -1.084e-03 & 8.456e-02 \\ 4.507e-08 & -8.565e-08 & -1.103e-03 \end{bmatrix}$$

Projection Matrix of Cube (frame1) is:

$$P_{cube} = \begin{bmatrix} 1.621 & 3.917e-01 & 1.977 & -1.536e+03 \\ -4.180e-01 & 1.672 & 1.398 & -1.304e+02 \\ -6.953e-05 & 1.321e-04 & 2.044 & 1.701 \end{bmatrix}$$

I made a matrix of the camera's homogenous coordinates of AR tag (c1):

$$X_{c1} = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

I then scaled those coordinates into the world frame:

$$sX_s = H_{cube} \cdot X_{c1}$$

I then normalized to ensure the last row was equal to 1:

$$X_s = \frac{sX_s}{sX_s[2]}$$

I then made a 4x4 matrix of the points shifted in the world frame (with dimension of cube in negative Z-direction:

$$X_w = \begin{bmatrix} X_s[0] \\ X_s[1] \\ -dimension_{cube} \\ 1 \end{bmatrix}$$

$$X_{wframe1} = \begin{bmatrix} 0 & 2.02e+02 & 2.02e+02 & 5.889e-09 \\ 1.46e-09 & -2.784e-12 & 2.02e+02 & 2.02e+02 \\ -2.02e+02 & -2.02e+02 & -2.02e+02 & -2.02e+02 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

I then shifted those points to the camera frame using the Projection Matrix (c2) and normalized:

$$sX_{c2} = P_{cube} \cdot X_w$$

$$X_{c2} = \frac{sX_{c2}}{sX_{c2}[2]}$$

Which gave me the (incorrect) Projected Cube corners (frame1) to be:

$$[5,1],[4,1],[4,0],[4,0]$$

From here, I then had to draw contour lines connecting the top corners of cube (*cube_corners*) and the bottom corners of cube (*tag_corners*) recursively. I accomplished this by making arrays consisting of the four points of each side of the cube then using the *opencv* functions *drawContours* and *line* treating the points as tuples.

I had the most difficulty with this section of the project. Specifically, in the determination of the cube corners projection, and then drawing them properly. I reviewed the methodologies of [9], [8], and [3] and still could not resolve. I was specifically getting this image or variations of the cube either connecting to the top left corner of the image or twisting about to an arbitrary point on the left of the screen. I determined this to be due to my projection matrix calculations and subsequent projection of the cube points above the AR tag and not my plotting functions/processes.

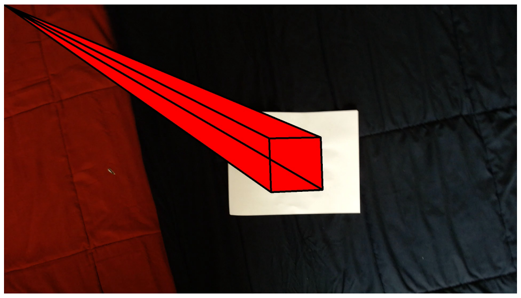

Figure 8: Bad Cube Projection in Frame 133

The solution to the projection of the cube points was that I was solving for λ in the wrong order and using the wrong values. I also found that my signs and values to be used were flipped as compared to the Homography equation of [7] (Equation 7). I also discovered I was using the size of the image vice the actual AR tag for Homography. I also discovered that my cube homography was treating the V.T matrix from the *svd* function as if it was V. Additionally, my projection matrix had some errors and followed the processes of [2] and [1]. Further, I discovered that my world frame coordinates matrix was using the incorrect values for the 3rd row (dimension of cube vs. a row of ones). My updated/corrected Homography and Projection Matrixes as well as a set of projected corners. Corrected Cube Homography Matrix:

$$H_{cube} = \begin{bmatrix} 5.773e-01 & 2.429e-13 & -3.297e-09 \\ 1.496e-12 & 5.773e-01 & -1.678e-09 \\ 2.442e-15 & -2.498e-16 & 5.773e-01 \end{bmatrix}$$

Corrected Projection Matrix of Cube (last Frame) is:

$$P_{cube} = \begin{bmatrix} -1.346e+03 & -4.898 & 9.321e+02 & 7.716e-06 \\ 4.934e & -1.355e+03 & 6.548e+02 & 3.928e-06 \\ -5.696e-12 & 5.659e-13 & 1.000 & -1.351e+03 \end{bmatrix}$$

Corrected Projected Cube corners (last frame) to be:

$$[1106, 423], [1249, 598], [1086, 748], [937, 570]$$

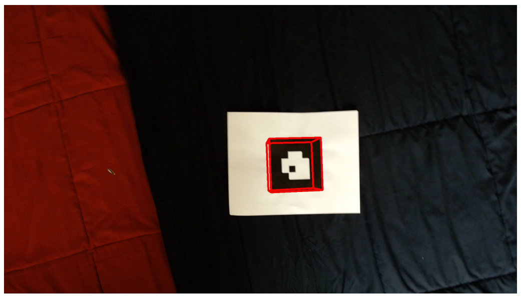
Figure 9: Virtual Cube placed on AR tag

The resulting video can be seen here.

4 Resources

4.1 My Github repository

https://github.com/jpittma1/ENPM673.git

4.2 References

Whole bibliography

[1] J. G. Acín. "Augmented reality github." (), [Online]. Available: https://github.com/juangallostra/augmented-reality.git. (accessed: 02.23.2022).

[2] ——, "Augmented reality with python and opencv." (), [Online]. Available: https://bitesofcode.wordpress.com/2018/09/16/augmented-reality-with-python-and-opencv-part-2/. (accessed: 02.23.2022).

[3] J. Albrecht. "Augmented reality in python." (), [Online]. Available: https://brianbock.net/augmented-reality-in-python/. (accessed: 02.16.2022).

[4] ——, "Enpm673-project1 git." (), [Online]. Available: https://github.com/jaybrecht/ENPM673-Project1.git. (accessed: 02.16.2022).

[5] S. Charifa, "Enpm673 spring 2022 notes," University of Maryland - College Park, MAGE, College Park, MD, Tech. Rep. Lectures, Jan. 2022.

[6] ——, "Testudo.png." (), [Online]. Available: https://www.clipartkey.com/view/mobTiJ_university-of-maryland-testudo/. (accessed: 02.14.2022).

[7] ——, "Theory behind homography estimation notes," University of Maryland - College Park, MAGE, College Park, MD, Tech. Rep. Project 1 addendum, Feb. 2022.

[8] S. Kakde. "Ar tag github." (), [Online]. Available: https://github.com/sakshikakde/ARTag-Detection-and-Tracking.git. (accessed: 02.14.2022).

[9] G. A. Kumar. "Enpm673 2020." (), [Online]. Available: https://github.com/govindak-umd/ENPM673.git. (accessed: 02.06.2022).

[10] MIT. "Singular value decomposition (svd) tutorial." (), [Online]. Available: https://web.mit.edu/be.400/www/SVD/Singular_Value_Decomposition.htm. (accessed: 02.07.2022).