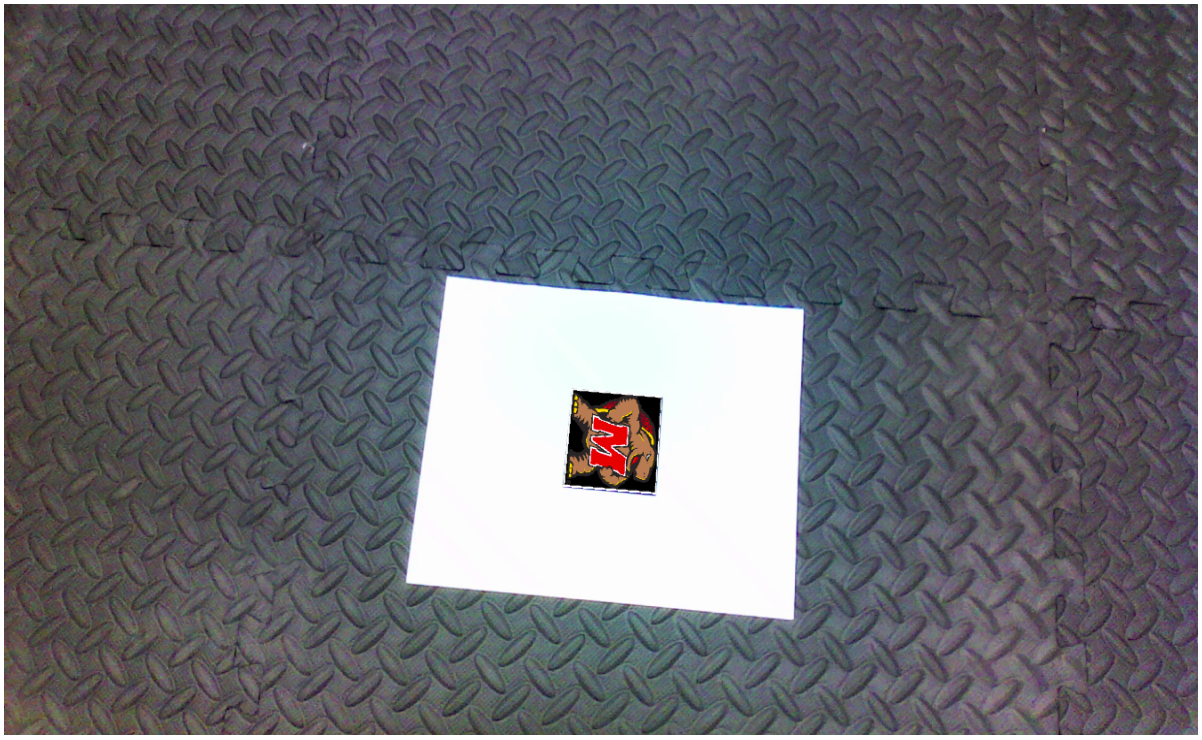
PROJ1 REPORT

ENPM673 - Perception For Autonomous Robots - Spring 21



Paras Savnani

08.03.2021

University Of Maryland College Park

Problem 1:

Objective: To detect and narrow out the AR tag region in the video frames and decode its orientation and ID to uniquely identify the tag.

1A:

One_a.py

For this problem, we have to detect the AR tag using the Fast Fourier Transform technique. Fourier Transform is used to analyze the frequency characteristics of various filters. Further, it is used to eliminate the repetitive noise/ non repetitive noise, depending on the application. For a sinusoidal signal x(t):

$$x(t) = Asin(2\pi ft)$$

we can say f is the frequency of signal, and if its frequency domain is taken, a spike can be seen at f.

- We consider an image as a signal which is sampled in two directions. So, taking a Fourier transform in both X and Y directions gives you the frequency representation of the image.
- For our problem we use numpy to take fourier transform (numpy.fft.fft2()).
- **NOTE: The low frequency component will be at the top left so, we have to shift it by N/2 in both directions to bring it in centre.**
- The fourier transform will give the result in 2 channels i.e **real** and **complex** and finally we will take the magnitude spectrum of the image to visualize the frequency domain.
- The central area will be much brighter than the rest of the image showing the higher concentration of low frequency content.
- For our problem to separate the background from the foreground we have to detect the edges of the white paper. So, we should use the high pass filter to get the edges and filter other content.

- To apply this filter we will generate a mask whose centre values are zero, and all other values are one. And we will multiply this with our output and take the inverse of the result. Also, we will shift the image back by N/2 in both directions.
- Finally, we will apply canny edge detection on the output and separate the white paper with AR tag from the background.

Problems Faced and Solution:

We used a high pass filter and as the background is repetitive, this filter will also pass a good amount of noise, so we have to set the correct radius of the mask to get the best results.

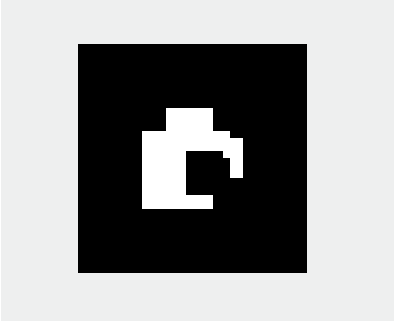
1B:

One_b.py

CODE EXPLANATION AND PROCEDURE:

For this part of the problem, we have to decode the orientation and ID of the AR tag, and we could use that function to decode the information in question 2. Normally, detecting the corners would provide satisfactory results, if the AR tag is not distorted, but warping the image in question 2, will distort the images and thus sharp corners won't be preserved. So, **average/mode** of the pixels in the given ROI is the robust way to detect the orientation and the ID.

- This file has a function to calculate the mean of the pixels in an ROI. Also, it has a function to detect orientation and the Id.
- Initially, some threshold and morphological operations are applied to
- The function detects the original parameters of the tag using the mean(if the average > 127.5, it is considered 1, else: it is 0).
- Finally, the orientation and the ID are right shifted and the corresponding actions are recorded, till the orientation is upright (i.e the white ROI is in the bottom right), and they are returned to use in the video frames.



Sample AR Tag from a video frame (Clearly not an ideal scenario for detecting corners)

Problem 2:

Objective: After detecting the AR tag uniquely, superimpose the testudo image in the correct orientation also, use the Projection matrix to project a cube onto the AR tag.

CODE EXPLANATION AND PROCEDURE:

Essential part of this problem is to detect the four corners of the AR tag using a robust and generalized detection pipeline, which will work on all the four videos. Following approaches were tried and tested to detect the corners of the AR tag:

1. **Corner detection** using **Harris corner/ ORB detector/ Shi Tomasi** Good features: (SIFT and SURF were not used as they are patented)

   Advantages:

   - Easy to implement
   - Computationally efficient
   - Can give sub pixel accuracy

   Disadvantages:

   - It is not robust

- Number of corners/features varies in each frame

2. **Hough Transform** to give the lines after canny edge detection:

   Advantages:

   - Robust method, gives accurate results for different frames
   - Corners can be detected as the intersections of the lines

   Disadvantages:

   - Hough transform gives multiple overlapping lines and separating the prominent lines for the weak lines is tough. Slope and constant can't be used to separate them, because the variation in them is so small.
   - Computationally costly algorithm
   - Coding complexity also increases as we have code to choose the prominent lines from all the lines based on some criteria.

3. **Contour Detection and Hierarchy** :

   Advantages:

   - Easy to implement
   - Moderately robust (It can be made more reliable with morphological operations)
   - Computationally efficient
   - Corners of the contours can be withdrawn using the ApproxPolyDP technique / Extreme corners of the contours / Taking a rotated bounding box around it.

   Disadvantages:

   - Establishing a generalized relationship between hierarchies in each frame can be a difficult task.
   - Doesn't work good with outliers (gives contours in noise)

Analyzing and implementing the above approaches, it was found that the contour detection works best for this use case, with least number of false positives.

utils.py

This file contains the helper functions to perform the superimposition of the testudo and detect orientation of the Ar tag. Following are the function descriptions for the helper functions, which are used in 2a and 2b.

svd:

1. This file has the custom svd implementation. This is a generalized function which can be used to find the svd of any matrix. The function decomposes the X(m*n) matrix into following parts:

$$X = U\Sigma V^T$$

2. To calculate the V^T matrix, we will take the dot product of X^T and X matrix.

$$X^T X$$

3. Then, we find the eigenvalues and eigenvectors of the above matrix and sort both of them based on the eigenvalues. (**NOTE: Numpy does not return sorted eigenvalues and eigenvectors, so this step is necessary for implementation of svd.)**
4. And then we take the transpose of the eigenvectors matrix to get V^T matrix.
5. To find the sigma matrix, we stack up the square roots of the sorted eigenvalues along the diagonals to get the Σmatrix.
6. Similarly to get the U matrix we take the dot product of X and X^T matrix.
7. Then, we find the eigenvalues and eigenvectors and sort them based on the

eigenvalues.

8. Finally, for the case where (m>n), U can have complex values, so we take the real part of the U matrix and return the U, Σ and V^T matrices.

get_homogrphy :

1. This function is used to get the homography matrix (H), for this we have to solve the homogeneous equation **AH=0**.
2. We construct the A matrix using the source points and destination points and we input it in the svd function.
3. This equation is solved using the svd function, and we have to take the last singular vector of the V_t matrix returned by the svd function.
4. Finally we have to reshape it to a 3*3 matrix to get the H matrix.

warp_perspective:

1. This function is used to warp the testudo image to the destination points in the video frame.
2. Here, we multiply the H matrix with each coordinate of the source image in homogeneous coordinates.
3. Then, we normalize the new coordinates and assign the intensity values of the original coordinates to these new coordinates.
4. Finally, we return the warped image (**NOTE: If we are using a bigger ROI to map to a smaller region, we will have out of bound indices, which maps to other regions in the image, so we should ignore these values using a try/except block**.)

orientation_detection:

1. This function is an intermediary function and it is used to return the orientation and id of the Ar tag.
2. For inverse warping we map the AR tag to a smaller region of 32*32, to prevent holes in the Ar tag, when mapping from a smaller region to a bigger region.
3. After, the homography and warping operations, we resize the image to

64*64 using linear interpolation, to facilitate better detection of the regions to decode the id and the orientation.

Two_a.py

1. Initially, the user is prompted to enter the number to play a specific video for the code.
2. Then, the testudo image and corresponding video is loaded and the testudo is resized to a smaller size to avoid more number of multiplications when warping the image
3. In the main while loop, the frames are received from the video and resized to 70% to increase the playback speed, also it is then grayscaled to do image processing operations.
4. Further aggressive thresholding is done, and noise is removed using the **opening** morphological operation with a custom 5*5 averaging kernel.
5. Next step is to find contours on the output image and use hierarchy matrix to narrow out the AR tag.
6. The hierarchy matrix gives the following list for each contour (**[Next, Previous, First_Child, Parent]**), we try to get the second child for each use case as the AR tag contour(assuming boundary of the frame as the parent contour).
7. If multiple second children are found then it means it's the multiple Tags video else, it is a single tag video.
8. So, after getting the second child list, we iterate through it and approx the second child contour with cv2.approxPolyDP function with the epsilon value to get four points.
9. So, we get the destination points from the approx contour and we generate source points from the shape of the testudo in **arbitrary order.**
10. Now, if we get 4 destination points, we find the bounding box around them and shift the destination points in accordance to the new bounding box and input them with a new sliced image into the orientation detection function.
11. This function returns the current orientation of the AR tag, its ID, its

threshold image and the actions required to correct the **arbitrary source points** to align the testudo.

12. We correct the source points list by left shifting them according to the actions list.
13. Next, we input original destination points and the corrected source points to get the homography matrix (H).
14. Finally, we input this H matrix in the custom warp perspective function with testudo and image frame as parameters, and we display the frame.

Problems faced and solutions:

- When I did the warp transform for the whole video frame, it slowed the video frame rate considerably, so to eradicate this issue I only warped the Region of Interest i.e the Ar tag to the destination points.
- When detecting the second child of the contour, the results were not consistent when the **white paper was cut by the frame,** so to eliminate this issue separate cases to filter out the second child were written.
- When using the extreme corners technique for the contour corners, sometimes the leftmost and the topmost corners coincided, similarly the rightmost and the bottommost corners, so I used approx PolyDP to get the corners.
- When projecting a smaller image to a bigger region, we can see holes in the images, so to eliminate this we can project it to a smaller region then interpolate it to get a better image.

Two_b.py

1. This problem is an extension of the the 2a problem, initially it also prompts the user to choose the video to run and it defines the K matrix (intrinsic parameters of the camera).
2. Next piece of code is similar to the part **Two_a,** where the 4 corners are detected with orientation and id of the AR tag.

3. Furthermore, for this problem we calculate the Projection Matrix(P), for that we have to decompose the H matrix into Rotation and Translation matrix.
4. We calculate the B matrix as:

$$\overline{B} = \lambda K^{-1} H$$

5. We find the scaling factor (λ) using the formula:

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

6. Now, we can compute the rotation and translation matrices using the following equations:

$$r_1 = \lambda b_1$$

$$r_2 = \lambda b_2$$

$$r_3 = r_1 x\, r_2$$

$$t = \lambda b_3$$

7. Finally, we compute the Projection Matrix as:

$$P = K\,[R|t]$$

8. To get the Z coordinates of the cube we multiply the Projection Matrix with the homogeneous source points with negative z values(as the top left corner of the cube is taken as origin).
9. A very rudimentary version of Kalman filter is applied on the z_points and the KF.predict() and KF.update() methods are called to get the points after the filter.
10. Next, we draw the edges of the cube to visualize it in 3 dimensions.

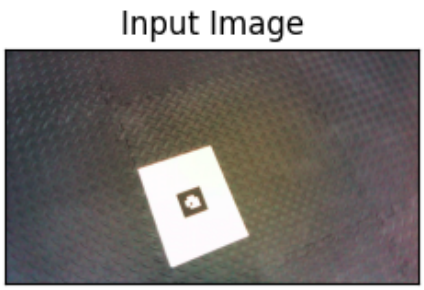
Kalman_filter.py

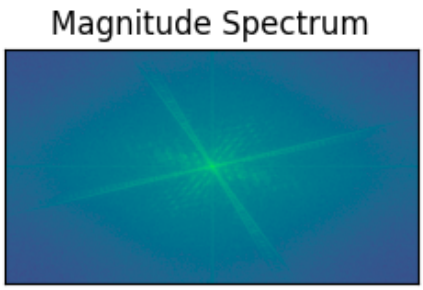
1. For applying the Kalman Filter, we will generate the state transition matrix(A) and control matrix(B). They have dt (sampling time) as their values in the matrix.
2. Then, we generate the transformation matrix (H), to map the measurement without considering the velocity.
3. Further, process noise covariance matrix is generated, which is multiplied by σ^2 which is the std. deviation of acceleration.
4. In the 2-D Kalman filter, we suppose that the measurement positions x and y are both independent, so we can ignore any interaction between them so that the covariance x and y is 0. We look at only the variance in the x and the variance in the y.
5. A class initiating all these matrices is generated with two functions predict() and update().
6. The predict() function updates the time state and calculates the error covariance.
7. Also, the update() function calculates the Kalman Gain and updates the error covariance matrix.

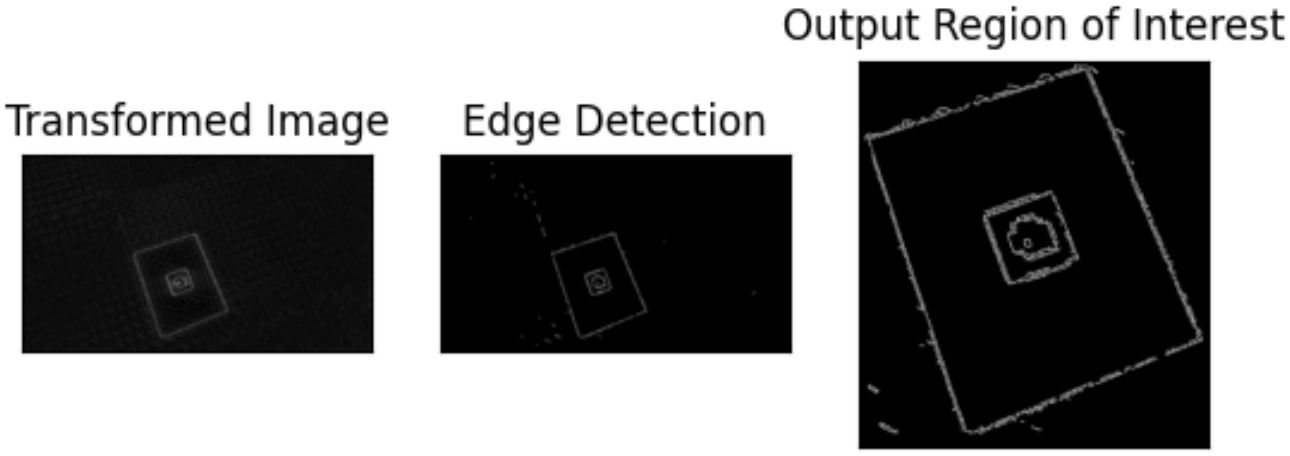
   Reference:
   https://machinelearningspace.com/2d-object-tracking-using-kalman-filter/

RESULTS

1. Problem 1A:



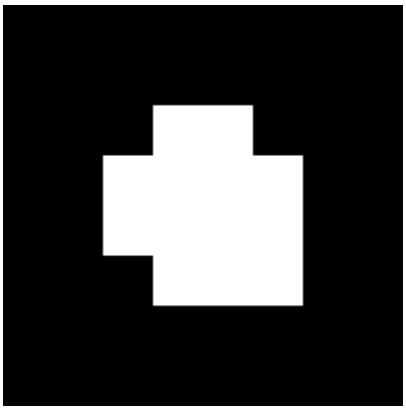




As we can see, a high pass filter to detect the edges of the white paper and the Ar Tag gives good results to separate out the desired region. Also, the magnitude

spectrum shows the **radial concentration of low frequency content** to generate an ideal mask.

2. Problem 1B:

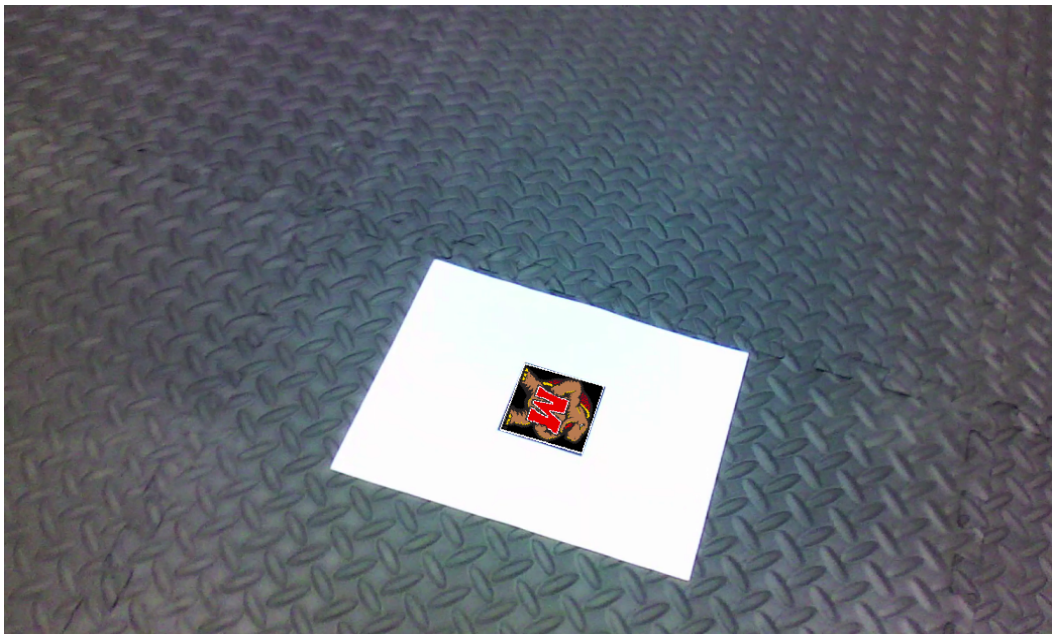


The output of this problem will be :

Orientation(Anticlockwise order- Bottom Left to Top Left): **0100**

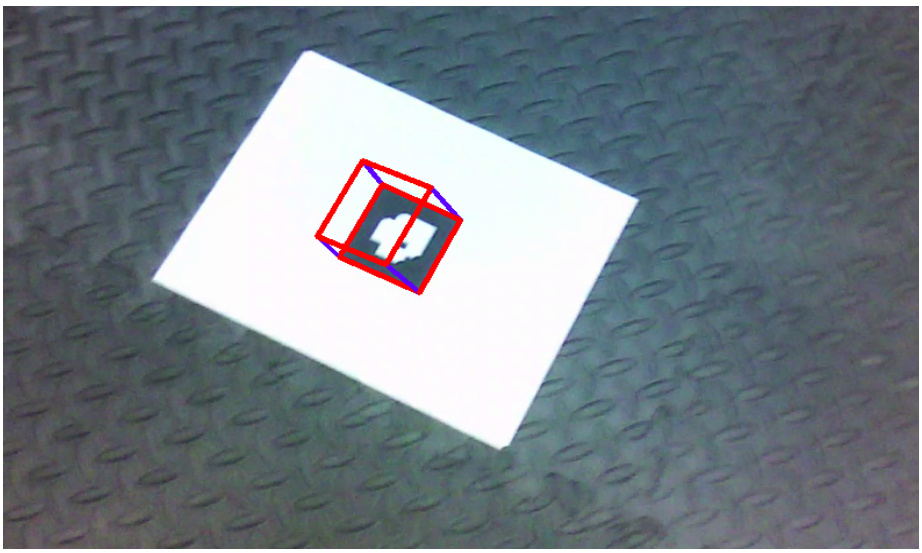
Tag ID with respect to the upright orientation: **1111**

3. Problem 2A:

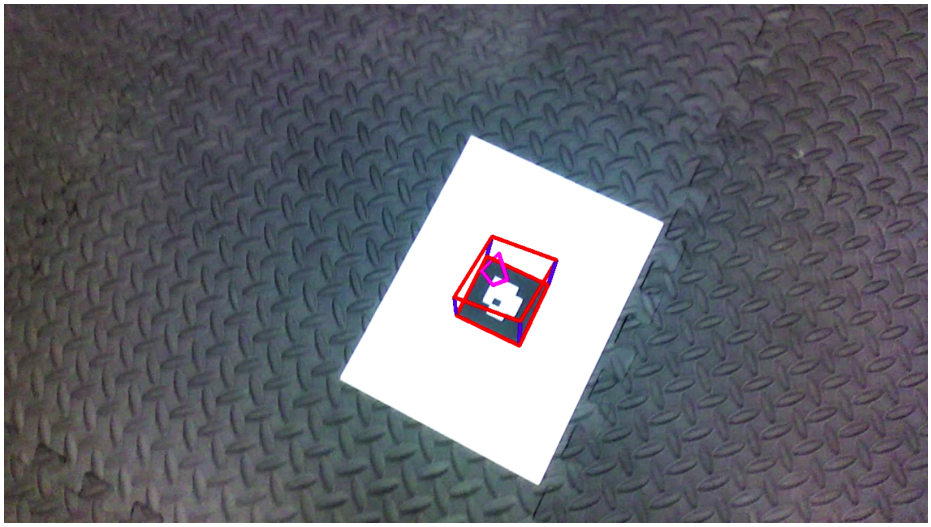


Example video frame from the Tag2.mp4 video, showing the superimposed testudo image aligned with the upright position of the AR tag.

4. Problem 2B:



Example video frame from the Tag1.mp4 video, showing the 3D cube superimposed over the AR tag.



Kalman filter is applied to this problem to get the z coordinates of the cube, the pink polygon represents the predicted coordinates according to the Kalman filter.

REFERENCES

- https://math.stackexchange.com/questions/494238/how-to-compute-homography-matrix-h-from-corresponding-points-2d-2d-planar-homog
- https://www.geogebra.org/?lang=en
- https://math.berkeley.edu/~hutching/teach/54-2017/svd-notes.pdf
- https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_contours/py_table_of_contents_contours/py_table_of_contents_contours.html#table-of-content-contours
- https://docs.opencv.org/master/de/dbc/tutorial_py_fourier_transform.html
- ENPM 673, Robotics Perception - Theory behind Homography Estimation
- CSCE 441: Computer Graphics - Image Warping - Jinxiang Chai
- https://machinelearningspace.com/2d-object-tracking-using-kalman-filter/