ENPM 673 - Perception for Autonomous Robots
Project 1 - Augmented Reality

Abstract

AR Tag is a fiducial marker system that is used primarily in ViewPoint Estimation and Tracking which is an important aspect of Robotics Perception and Robot Navigation. The Estimation involves both detection and tracking of the tag to later perform several image processing techniques.

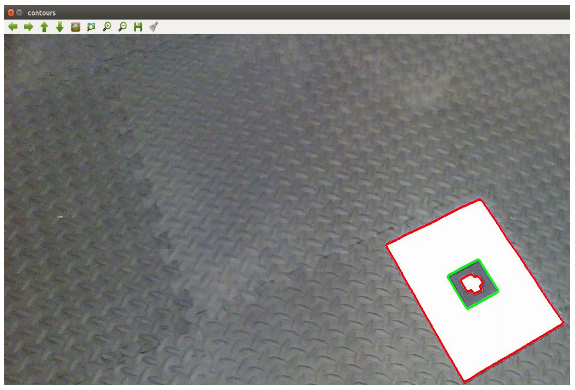
1. Detection

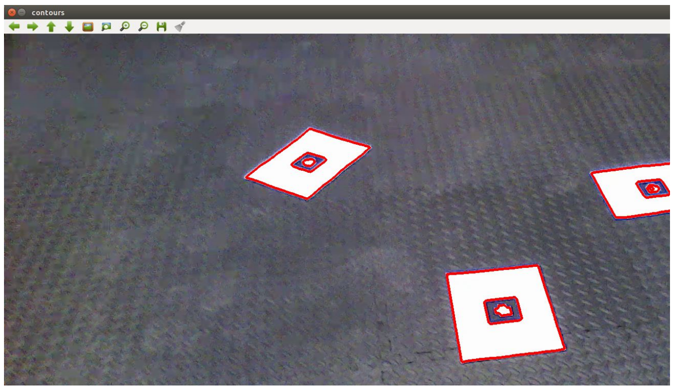
The detection process involves finding the AR Tag from the given video input sequences and decoding the encoded binary representation of *id* and *orientation* of the tag.

1.1 Homography Pipeline

Homography is used to represent a transformation between two planes in the projective space, i.e, it is the transformation that projects the AR tag from the world coordinate system to a polygon in the image plane, viz., pixel coordinates. The pipeline to detect the homography is as follows:

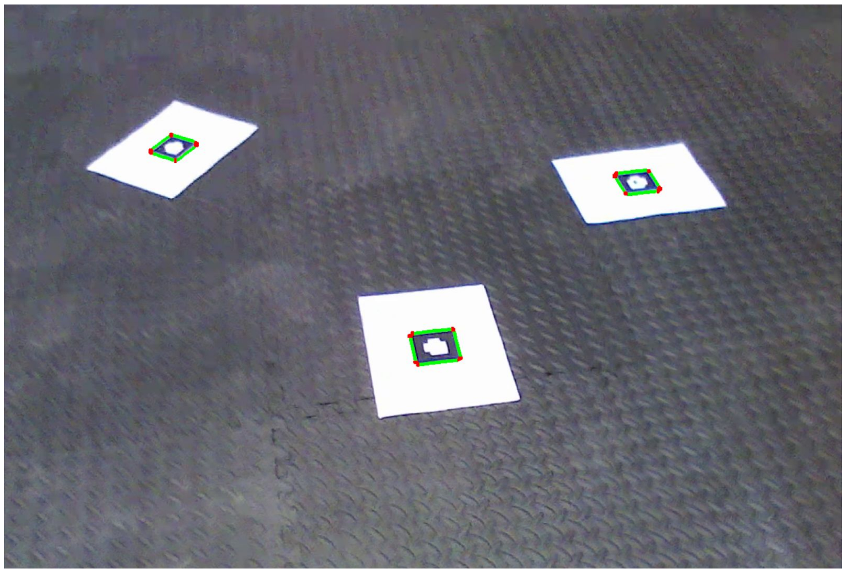
- For every image frame in the input video sequence, we convert each to a ***grayscale*** and smoothen the image by using ***Gaussian filter*** with the width and height of the of the gaussian kernel which should be positive as well as odd, because it is highly effective in removing gaussian noise in the image.
- Since we are dealing with grayscale image, it is easier to tell at any point if a pixel is black or white and so we choose a ***threshold*** value such that if the pixel value is greater than the threshold, it is assigned to be white, if not black.
- To obtain the corner points of the tag, [we try to get the ***contours*** in the smoothened image with the above chosen threshold for better accuracy, which will give the line along the boundary having same color or intensity.] This is especially efficient when the tag is on a white paper and background is black.



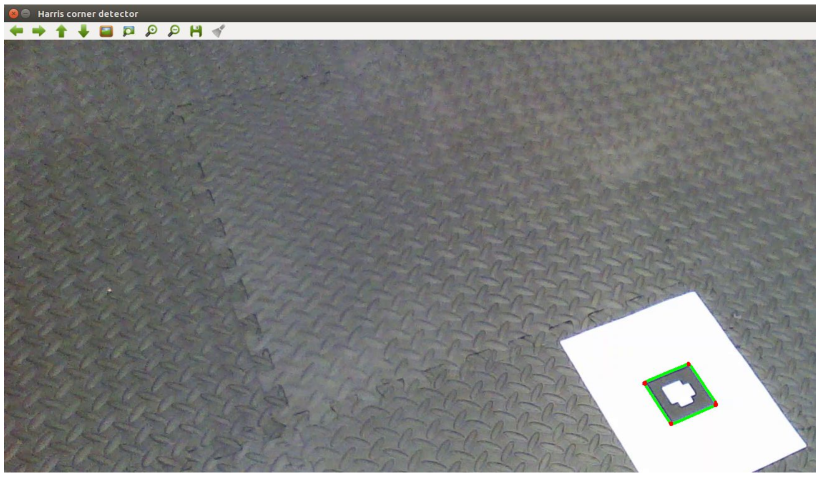


- We have got the contours to detect the white paper in the image. However, here the tag contour shape (child) is inside the white paper contour shape (parent). Representation of the above relationship can be obtained from getting ***hierarchy*** of the contours and using the option *cv2.RETR_TREE* gives the full hierarchy of the nested contours.

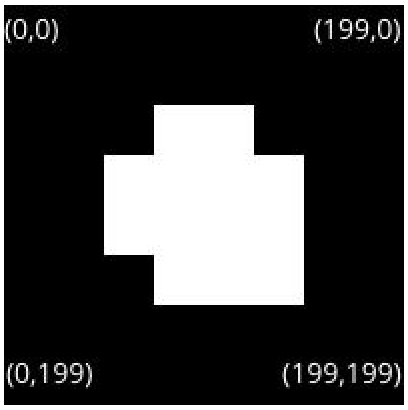
- With the hierarchy we can find the inner/child frame which corresponds to the AR tag.



- This contour is used to extract the corner points. Contour is preferable since there are less number of edges and corner points in an image.
- Above removing white noises also shrinks our object frame. Hence we dilate it so that the object area increases which can be used to draw a contour to visualize the tag frame.
- To further refine the detected corners with maximum accuracy, we use sub-pixel accuracy with a defined neighbourhood size to search for the corners.



- In order to get the corner points corresponding to particular corners of the reference image, we choose a convention. We input the reference image in anti-clockwise direction.



- The corner points computed from the frame contours are sorted.

Issues Faced:

- We started with the following pipeline where, Initially we used a custom built corner points detection function. This inputs contours to the harris filter to detect the corner points.
- Then detected the corner points which are regions in the image frame with largest intensity variation in all directions. We used the *Harris Corner Detector* to extract the corners of the frame which essentially looks for the difference in intensity in all directions.
- But in multiple instances, we get multiple corner points over the edges. To select the actual corner, we find the maximum and minimum along x and y axis to sort it in clockwise direction.
- However the algorithm we implemented was not robust enough to handle all the quadrilateral cases like the square, where maximum and minimum along an axis is the same point.
- The incorrect order of points caused errors in the homography matrix, which resulted in out of bound world coordinates. To counter this we utilized the function `cv2.approxPolyDP()` to get the exact corner points.

1.2 Homography Estimation

- With the detected corners, we can compute the homography `H` given by

$$\boldsymbol{H}_w^c = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{32} & h_{32} & h_{33} \end{bmatrix}$$

- We construct a matrix A in `Ah = 0` such that

$$\begin{bmatrix} {x_k}^{(w)} & {y_k}^{(w)} & 1 & 0 & 0 & 0 & -{x_k}^{(c)}{x_k}^{(w)} & -{x_k}^{(c)}{y_k}^{(w)} & -{x_k}^{(c)} \\ 0 & 0 & 0 & {x_k}^{(w)} & {y_k}^{(w)} & 1 & -{y_k}^{(c)}{x_k}^{(w)} & -{y_k}^{(c)}{y_k}^{(w)} & -{y_k}^{(c)} \end{bmatrix} \begin{bmatrix} \boldsymbol{h}_1^T \\ \boldsymbol{h}_2^T \\ \boldsymbol{h}_3^T \end{bmatrix} = \mathbf{0}_{2\times 1}$$

- Next we decompose A using SVD such that

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}^T$$

- The vector `h` can then be obtained using

$$\boldsymbol{h} = \frac{\left[v_{19}, \ldots, v_{99}\right]^T}{v_{99}}.$$

- Finally we reconstruct the elements of `h` into a 3x3 matrix using reshape function.

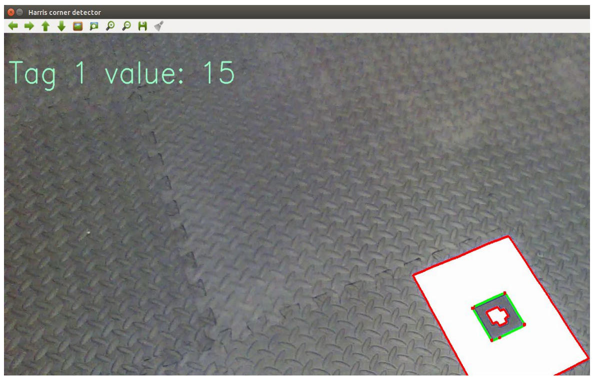
1.3 Using the Homography Transformation
- Each and every world coordinate has to be transformed into camera coordinate frame.
- Also the pixel values of the world coordinates (from video feed), has to be given to the corresponding pixel coordinates of the camera frame.
- To obtain the data encoded in the tag image, we use a threshold of 220 (obtained after experimenting) transformed_image[$X_{(c)}$[1]][$X_{(c)}$[0]] = frame[$X_{(w)}$[1]][$X_{(w)}$[0]]

Issues faced

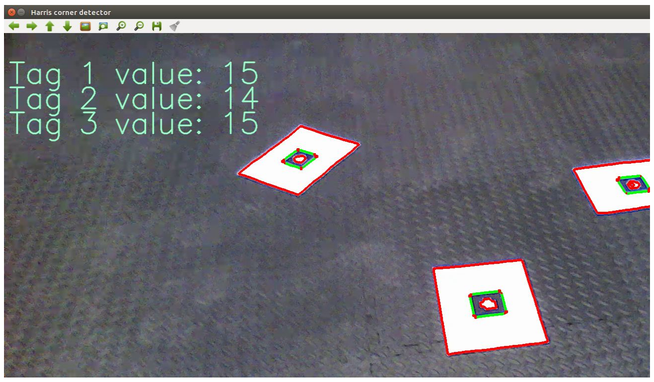
'A' matrix constructed using corner points turned out to be singular at times. This was due to detection of wrong corner points. We updated the getCorner_points function to sort the corner points on all situations to avoid this issue.

1.4 Encoding Scheme
- Once we have the transformed image after the homography, based on the given encoding scheme, we split the transformed square image of MxM size into a 8x8 matrix in format of a grid to eliminate any false detections.
- The 200x200 image is transformed into a 8x8 grid and the color intensity is decided by the mean value of pixels found there.
- by trial and error method to differentiate white and gray/black boxes) to assign a binary value to it.
- The padding of 2 cells width is along the borders. In the 4x4 inner cell of interest to us, we know that the *orientation* is such that the lower-right corner is a white cell. Hence, we rotate the frame such that we obtain the desired orientation and also the actual orientation of the given image in the input sequence.

- Lastly, after the padding and orientation, the innermost 2x2 cell is known to be binary representation of the tag's ID. The order of significance goes down in clockwise direction starting from the left most bit. Since, white cells are 1 and black cells are 0, we go clockwise to arrange the cell data in binary such that we get the full binary data of significant bits with respect to its orientation encoded in the AR Tag and hence its actual integer value.



Tag 1 value: 15



Tag 1 value: 15
Tag 2 value: 14
Tag 3 value: 15

2. Tracking

Tracking involves extracting the AR tag from the dynamic view and decoding the data in it. Here, we have to decode the tag throughout the given video input sequences. Also, the tag has to be superimposed with another image and a virtual cube has to be placed on the tag based on the tag's orientation and position, also known as the pose.

2.1 Superimposing image on tag

The steps followed include :

- Homography matrix `H` is computed between the four corners of the tag and the corresponding four corners of the given template image `Lena.png`.

- To superimpose the Lena image into the video frame, we have find the pixel coordinates on the frame corresponding to points on lena.

$$X_{(c)} = H\, X_{(w)}$$

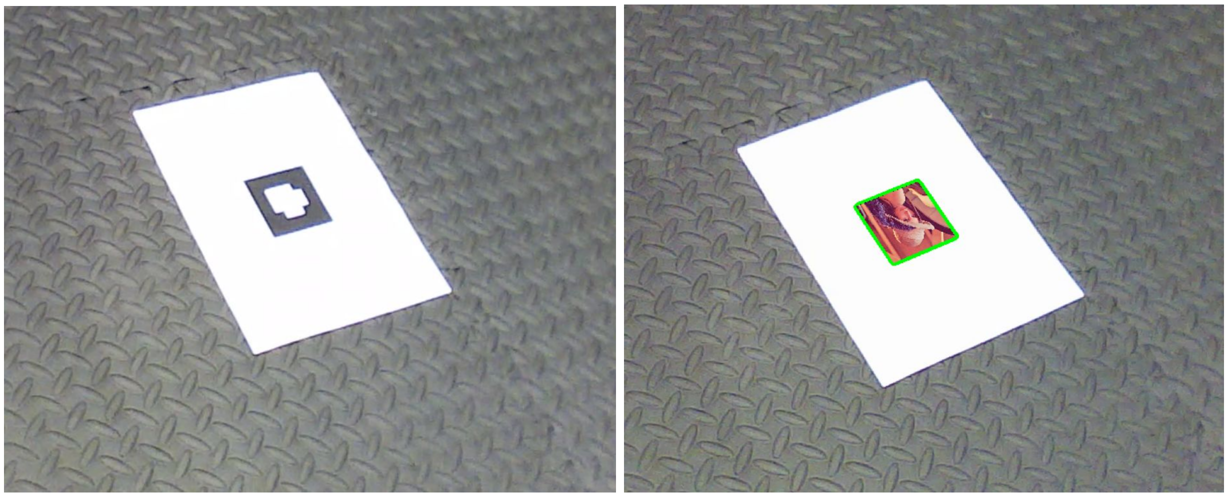
$$X_{(w)} = H^{-1}\, X_{(c)}$$

$X_{(c)}$ varies as in range o f 512x512 (Lena image). $X_{(w)}$ for each and every pixel of Lena is computed using the inverse transform and the pixel values from Lena are assigned there.

frame[$X_{(w)}$[1]][$X_{(w)}$[0]] = Lena[$X_{(c)}$[1]][$X_{(c)}$[0]]

- But this problem encountered is that the orientation of the Lena doesn't match the orientation of AR tag. So to solve this issue, the decoding function was used. The number of 90 degree rotations needed to get the white square in the inner 4x4 grid to the lower-right corner was counted.

- Next template image (Lena) was rotated by the same number of rotations before the inverse transformation and it resulted in Lena oriented same as the upright position of the tag.



2.2 Placing virtual cube on tag

The homography that we derived above can be thought of as a transformation that projects a square marker from the world coordinate system onto a polygon in the image plane. But to project a 3D cube we need to get a 3D transformation between the camera frame and the world frame. The following steps are followed:

- Firstly, homography `H` between the world coordinates and the image plane using the corners of the AR Tag in the reference marker image and the corners of the AR Tag that is in the input frame was computed.
- Projection matrix gives the relation between 2D camera image points and the 3D world coordinate points.
- In order to transform a 3D point from the camera coordinate system to a 2D point in camera's image plane, the intrinsic camera matrix is used (this is constant).

- As the camera's view changes we need to compute the rotational and translational components of the transformation between 3D point from the world coordinate system to the 3D point in the camera's coordinate system.
- Since homography is transformation between world and camera frame's 2D plane , we compute the rotational vectors in those particular direction are taken.

$$\begin{aligned}\boldsymbol{x}^{(c)} &= \boldsymbol{P}\boldsymbol{x}^{(w)}\\ \boldsymbol{x}^{(c)} &= \boldsymbol{K}[\boldsymbol{r_1}, \boldsymbol{r_2}, \boldsymbol{r_3}, \boldsymbol{t}][x^{(w)}, y^{(w)}, 0, 1]^T\\ \boldsymbol{x}^{(c)} &= \boldsymbol{K}[\boldsymbol{r_1}, \boldsymbol{r_2}, \boldsymbol{t}][x^{(w)}, y^{(w)}, 1]^T\end{aligned}$$

Assuming the rotational matrix as B,

$$\begin{aligned}\lambda\boldsymbol{H} &= \boldsymbol{K}\widetilde{\boldsymbol{B}}\\ \widetilde{\boldsymbol{B}} &= \lambda\boldsymbol{K}^{-1}\boldsymbol{H}\\ \boldsymbol{B} &= \lambda\widetilde{\boldsymbol{B}}(-1)^{|\widetilde{B}|<0},\end{aligned}$$

The scaling factor is computed from

$$\lambda = \left(\frac{||\boldsymbol{K}^{-1}\boldsymbol{h}_1|| + ||\boldsymbol{K}^{-1}\boldsymbol{h}_2||}{2}\right)^{-1}$$

Using the following equations the projection matrix is computed

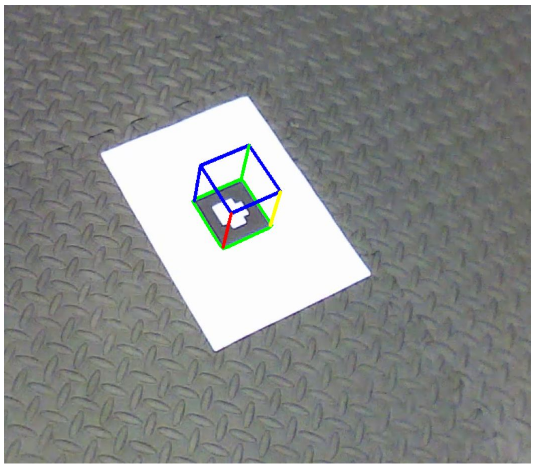
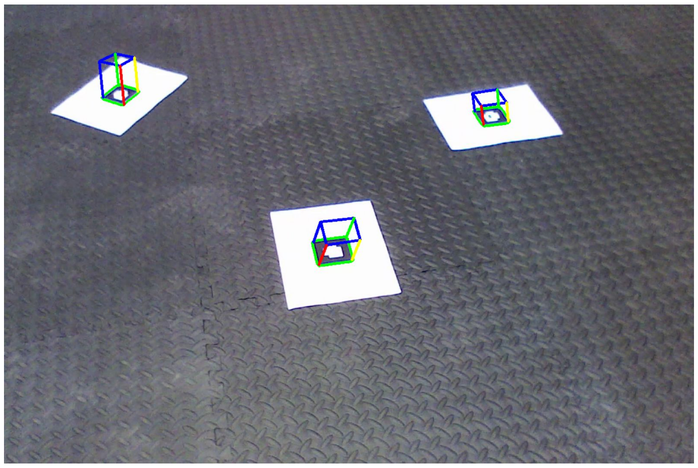
$$\boldsymbol{r}_1 = \lambda\boldsymbol{b}_1 \, , \boldsymbol{r}_2 = \lambda\boldsymbol{b}_2 \, , \boldsymbol{r}_3 = \boldsymbol{r}_1 \times \boldsymbol{r}_2 \, , \boldsymbol{t} = \lambda\boldsymbol{b}_3 .$$

- Since a 3D cube is a simple structure with 8 corner points and lines joining them, we assume that the virtual cube is lying on the` top of the marker which is the negative Z direction in the input cube, we can get the remaining four corners of the cube.
- Finally, we can now transform all the corners of the cube to the image plane using the projection matrix `P`.

Issues faced

But the transform produced points that were out of bounds to the image plane. After, many attempts we found that we used the transformation H from $X_{(w)}$ to $X_{(c)}$. Since the cube coordinates(like [200,0,200,1]) are $X_{(c)}$ to $X_{(w)}$. So, we computed the inverse transform to provide it as an input to the projection matrix. This resulted in the proper 3D cube visualisation.

ENPM 673 - Perception for Autonomous Robots
Project 1 - Augmented Reality

How to run code

1. Unzip the folder which has the code and input sequences
2. Each of the following code parts asks you to select the input sequence you would like to run the code with.
3. Run `python ./part_1.py` for AR Tag detection (1)
4. Run `python ./part_2.1.py` for Superimposing Lena image on AR Tag (2)
5. Run `python ./part_2.2.py` for Placing virtual cube on AR Tag (3)

Project Team

1. Nantha Kumar Sunder (UID: 116104777)
2. Nithish Kumar (UID: 116316958)
3. Rama Prashanth (UID: 116428941)

References

1. Computer Vision, A Modern Approach, Forsyth and Ponce (http://cmuems.com/excap/readings/forsyth-ponce-computer-vision-a-modern-approach.pdf)
2. https://docs.opencv.org/3.1.0/dd/d49/tutorial_py_contour_features.html
3. https://www.pyimagesearch.com/2016/03/21/ordering-coordinates-clockwise-with-python-and-opencv/
4. https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_features_harris/py_features_harris.html
5. https://bitesofcode.wordpress.com/2017/09/12/augmented-reality-with-python-and-opencv-part-1/