ENPM-673
Perception for Autonomous Robots
Project 1

Group Members

Amrish Baskaran(116301046)

Arpit Maclay(116314992)

Bala Murali Manoghar Sai Sudhakar(116150712)

Task 1- Detection:

The first task of the project is to detect the AR tags with their orientation and ID.

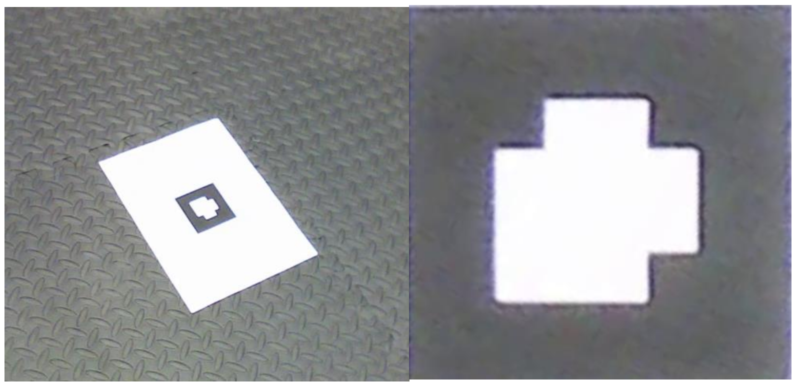
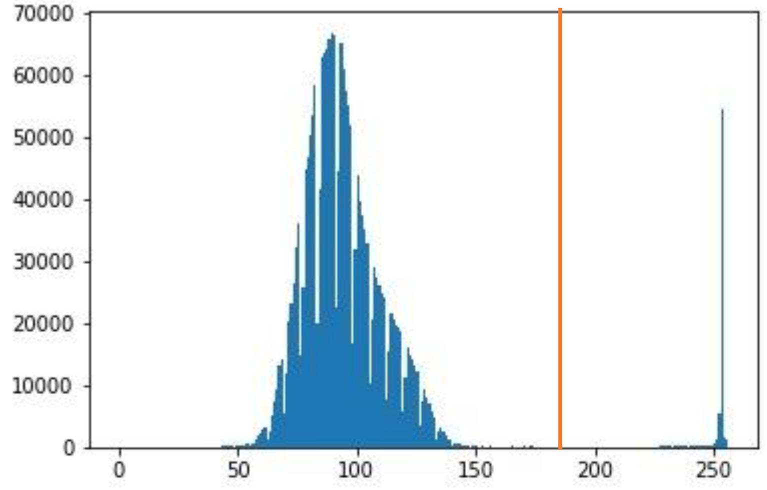


Fig.1 AR tag

A video with the AR tag is provided. The video is read frame by frame and for each frame the detection algorithm is executed.

Pipeline

1. **Preparing Frame**
   To find the contours the frame must be converted to greyscale. This image is converted to binary so that the contour detection in reliable. To fix the thresholding limit, histogram of grey scale image is analyzed a sweet value which has the maximum separation of pixel densities



2. **Finding Corners**
   To get the corners of the AR tag a function *cv2.findContour* which returns all closed regions in the frame. By using cv2.approxPolyDP the contours are approximated to a polygon. Based on the

result rectangular regions are filtered out with contour representing only the corners. To get the AR tag region we use the hierarchy option of cv2.findContours which is in the form of a tree as set using the flag cv2.RETR_TREE and is set to having some Parent Contour (Possible due to white paper's contour).

3. **Homography**

After the detection of the corners of the tag, we use the homography to convert the image coordinates of the tag to the world coordinates (Reference frame).

$$H = K[r_1, r_2, t]$$

Where K- camera calibration matrix
r_1, r_2 −ccolumns of rotation matrix
$x^{(w)}, y^{(w)}$- coordinates of the destination point in world frame

To get the homography matrix 4 points on the image coordinates and the corresponding 4 points on the world coordinates are required.
Using the above formula a system of equation is obtained and converted to matrix form

$$AH = 0$$

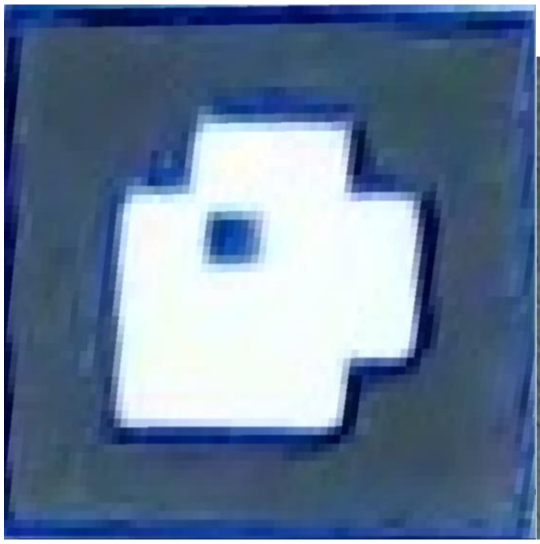
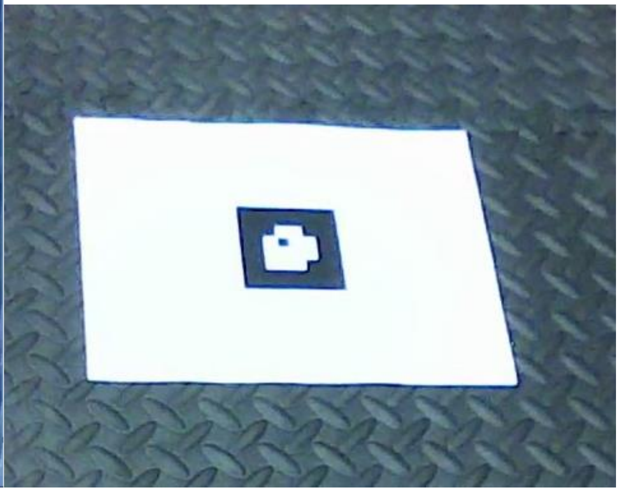
where $H = [h_{11}\ h_{12}\ h_{13}\ h_{21}\ h_{22}\ h_{23}\ h_{31}\ h_{32}\ h_{33}]^T$ represents the Homography matrix. The matrix need not be decomposed further as a reference frame is being used.
Here $h_{33} = 1$ since it is planar transform. The above system is solved using SVD decomposition of matrix A.

```
def getHomography(src,dst):
    rowSet = []
    for i,j in zip(src,dst):
        rowSet.append(SubMatrix(i,j))
    A = np.vstack((rowSet[0],rowSet[1],rowSet[2],rowSet[3]))
    U, s, V = np.linalg.svd(A)
    H= V[-1, :]/V[-1,-1]
    H=np.reshape(H,(3,3))
    return H
```

```
def SubMatrix(srcPt,dstPt):
    xw = srcPt[0]
    yw = srcPt[1]
    xc = dstPt[0]
    yc = dstPt[1]
    output = np.array([[xw,yw,1,0,0,0,-xc*xw,-xc*yw,-xc],
                      [0,0,0,xw,yw,1,-yc*xw,-yc*yw,-yc]])
    return output
```

Here the inputs to the function in general are points on source and corresponding points on the destination to which transform has to be made.

4. Transforming images

To transform image from one coordinate to another we need to apply the Homography to every point on the source image and giving it the corresponding pixel value. An image of known dimensions is obtained which will be used for further processing.

$$[x^{(c)}, y^{(c)}, 1]^T = H[x^{(w)}, y^{(w)}, 1]^T$$

where (w) represents source image and (c) represents destination image.

```
def warpPerspective(src,h,x,y):
    h = np.linalg.inv(h)
    dst = np.zeros((x,y,3),dtype = np.uint8)
    for i in range(x):
        for j in range(y):
            X = (h[0,0]*j+h[0,1]*i+h[0,2])/(h[2,0]*j+h[2,1]*i+h[2,2])
            Y = (h[1,0]*j+h[1,1]*i+h[1,2])/(h[2,0]*j+h[2,1]*i+h[2,2])
            if X>=0 and X<src.shape[0] and Y>=0 and Y<src.shape[1]:
                dst[j,i] = src[math.floor(Y),math.floor(X)]
    return dst
```

For better implementation the inverse of the homography matrix is Multiplied to destination image pixel coordinates and the corresponding source image coordinates used and pixel values are obtained. This helps us to avoid cropping the transformed image and reduces number of iterations. Further information of the formula used can be found here.
Using this function the AR tag image is obtained in reference frame coordinates.

5. AR tag Decoding

The AR tag image must be converted to an 8x8 matrix to be decoded. This is achieved by thresholding and resizing the image.

```
arTag = cv2.resize(dst0, (8,8))
(thresh, arTag) = cv2.threshold(arTag, 150, 255, cv2.THRESH_BINARY | cv2.THRESH_OTSU)
```

1. **Orientation**

   The orientation of the AR tag depends on the value of the points after padding elements are removed. In reality, the transformed image might not be a square and might be stretched and skewed due to distortions as shown below. Thus, the absolute positions in 8x8 matrix cannot hardcoded. To reliability obtain the corner elements, we out a bounding box and taking the corners of the bounding box. This will give the orientation of the AR tag.

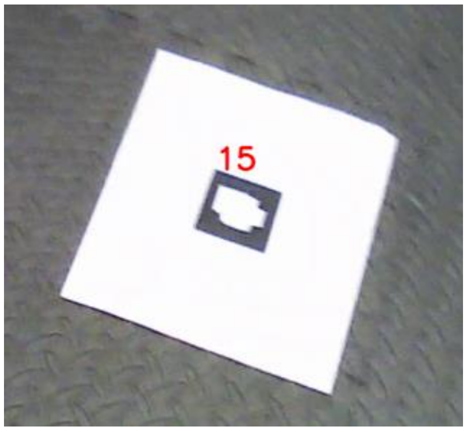
```
detectedCnt, hierarchy = cv2.findContours(arTag, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
        contours=detectedCnt[0]
        for i in detectedCnt:
            if cv2.contourArea(i) > 4 and cv2.isContourConvex(i):
                contours=i
        [[min1,min0]] = np.amin(contours,axis = 0)
        [[max1,max0]] = np.amax(contours,axis = 0)
        orientationIndex = [[min0,min1],[max0,min1],[max0,max1],[min0,max1]]
```

2. **Encoding**

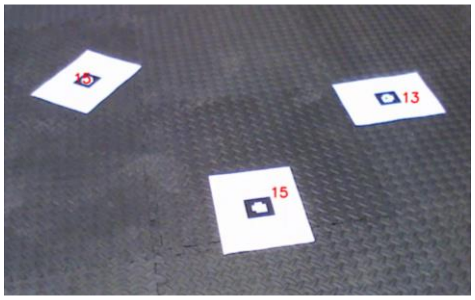
   The center 2x2 matrix of the 8x8 AR tag is used to get the binary encoding based on the orientation. The encoding scheme mentioned as per problem description is followed.

```
for i in [[3,4],[4,4],[4,3],[3,3]]:
            if arTag[i[0],i[1]] == 255:
                arCode.append(1)
            else:
                arCode.append(0)
        rotate(arCode,orientationCor)
for i in arCode:
            arNum = str(i) + arNum
        print(arCode)
        print(arNum)
        arNum = int(arNum,2)
        print(arNum)
```

6. **Displaying AR Tag**
   The obtained AR code as mentioned above is displayed in the frame. The code us displayed in near the AR tag.



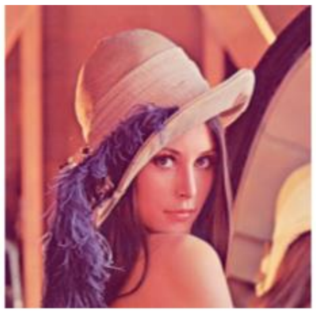
7. **Multiple Tag detection**
   The above pipeline is repeated on the same frame till all the contours are analyzed and AR tags are marked.

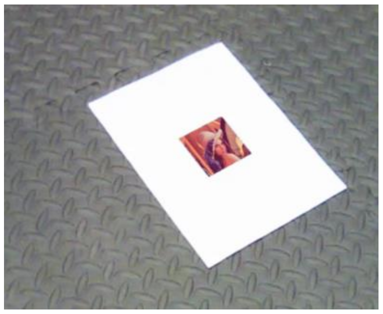


Task 2- Superimposing image onto the tag:

To place the template image (Lena.jpg) on the AR tag, we load the image then we orient it according to the orientation encoding obtained from AR tag as a reference. The Homography from oriented reference frame to camera image is found by taking the four corresponding corners.

As the camera moves every frame of the video is iterated over to find homography such that for every frame a different homography matrix is obtained to convert the coordinates of image on to the AR tag.

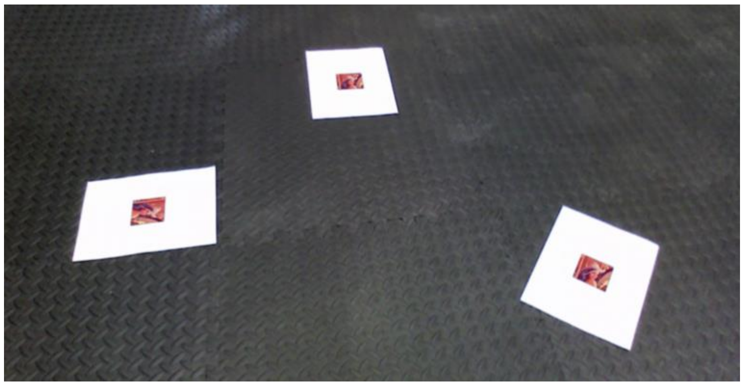




```
M = getHomography(np.float32(lenaCor),ptsOr)
    linaOnImage = warpPerspective(lena, M, frameCopy.shape[1],frameCopy.shape[0])
    _, mask = cv2.threshold(linaOnImage, 1, 255, cv2.THRESH_BINARY)
    mask_inv = cv2.bitwise_not(mask)
    mask_inv = cv2.cvtColor(mask_inv,cv2.COLOR_BGR2GRAY)
    frame_bg = cv2.bitwise_and(overLayBase,overLayBase,mask = mask_inv)
    linaOnImage = cv2.add(linaOnImage,frame_bg)
```

Pipeline:

1. Find homography with source as reference coordinates (Lena image) and destination as AR tag region on image coordinates.
2. Apply transformation using homography to the lena image.
3. Convert a copy of the transformed image to greyscale with very low threshold and create a mask. This will give us an image with a white region in the lena region.
4. Invert the mask and use this with a bitwise and operator to suppress the AR tag region.
5. Add these the transformed lena image to this to get the required augmentation.
6. The above pipeline is repeated on the same frame till all the contours are analyzed and lena image is superimposed on all AR tags.



Task 3-Placing a virtual cube on the tag:

To Project any point in the world coordinates to image coordinates we require a projection matrix. This is obtained from the homography matrix given by

$$H = K[r_1, r_2, t]$$

And the camera calibration matrix given by

$$K = \begin{bmatrix} 1406.08415449821 & 0 & 0 \\ 2.20679787308599 & 1417.99930662800 & 0 \\ 1014.13643417416 & 566.347754321696 & 1 \end{bmatrix}^T$$

The projection matrix is given by

$$P = K[r_1, r_2, r_1, t]$$

The projection matrix comprises of a Rotation matrix and a translation vector and the Calibration matrix. The homography matrix comprises a part of this rotation matrix and calibration matrix. Using this info, we can get the remaining part of the rotation matrix using few properties of a rotation matrix, i.e., columns of a rotation matrix are orthonormal and determinant is 1.

$$[R|T] = K^{-1}H$$

Normalizing the vectors,

$$L = \sqrt{\|H_{C1}\| \times \|H_{C2}\|}$$

$$R_1 = \frac{H_{C1}}{L}$$

$$R_2 = \frac{H_{C2}}{L}$$

Take rotation and translation vectors separately

$$R_3 = R_1 \times R_2$$

$$T = \frac{H_{C3}}{L}$$

Finding orthonormal basis since R1 and R2 may not be perpendicular

$$c = R_1 + R_2$$

$$p = R_1 \times R_2$$

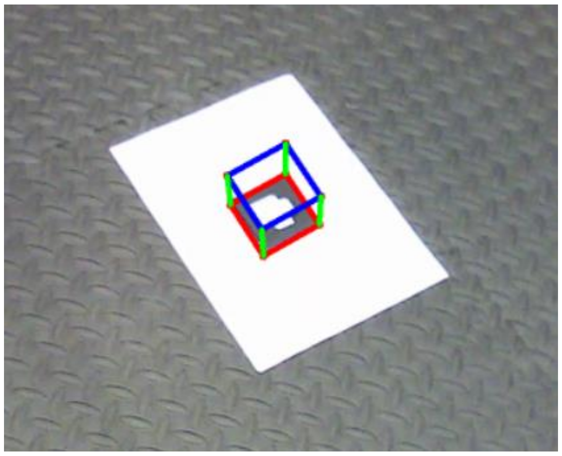
$$R_1^* = \frac{1}{\sqrt{2}}\left(\frac{c}{\|c\|} + \frac{d}{\|d\|}\right)$$

$$R_2^* = \frac{1}{\sqrt{2}}\left(\frac{c}{\|c\|} - \frac{d}{\|d\|}\right)$$

$$R_3^* = R_1^* \times R_2^*$$

$$R = [R_1^* \quad R_2^* \quad R_3^*]$$

$$P = [R_1^* \quad R_2^* \quad R_3^* \quad T]$$

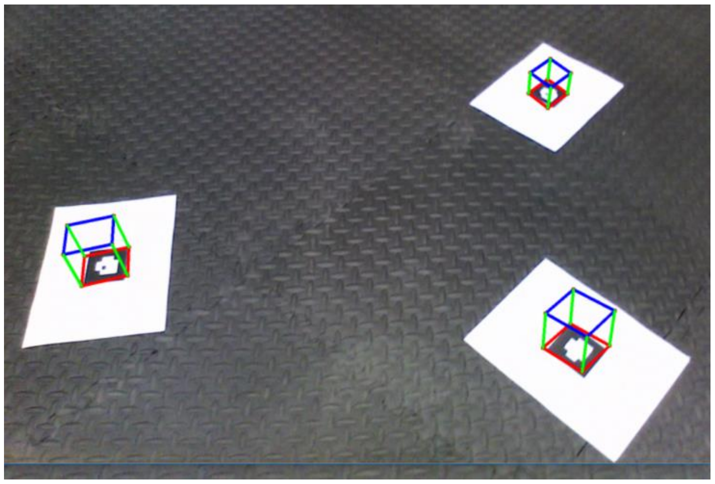


The above methodology and python function is based on this tutorial.

```
def projection_matrix(camera_parameters, homography):
homography = homography * (-1)
rot_and_transl = np.dot(np.linalg.inv(camera_parameters), homography)
col_1 = rot_and_transl[:, 0]
col_2 = rot_and_transl[:, 1]
col_3 = rot_and_transl[:, 2]
l = math.sqrt(np.linalg.norm(col_1, 2) * np.linalg.norm(col_2, 2))
rot_1 = col_1 / l
rot_2 = col_2 / l
translation = col_3 / l
c = rot_1 + rot_2
p = np.cross(rot_1, rot_2)
d = np.cross(c, p)
rot_1 = np.dot(c / np.linalg.norm(c, 2) + d / np.linalg.norm(d, 2), 1 / math.sqrt(2))
rot_2 = np.dot(c / np.linalg.norm(c, 2) - d / np.linalg.norm(d, 2), 1 / math.sqrt(2))
rot_3 = np.cross(rot_1, rot_2)
projection = np.stack((rot_1, rot_2, rot_3, translation)).T
return np.dot(camera_parameters, projection)
```

Pipeline

1. Find homography with source as reference coordinates and destination as image coordinates of the AR tag.
2. Using the camera calibration matrix and homorgraphy get the Projection matrix.
3. Determine the points on the reference coordinates of the cube.
4. Multiply each point of the reference coordinates of the cube with the projection matrix to obtain their respective camera coordinates.
5. To project the cube on image plane divide x,y of each point by the z value.
6. Draw lines between the required points in the image plane to get the required cube projection.
7. The above pipeline is repeated on the same frame till all the contours are analyzed and cube is placed on all AR tags



Running the code

1. Unzip the folder
2. Place "Tag0.mp4", "Tag1.mp4", "Tag2.mp4", "multipleTags.mp4" and "Lena.png" files inside this folder.
3. Run the appropriate .py file for respective videos.

PS: We have attempted to write the wrapPerspective function but the function was too slow to execute and we are getting few errors due to transformation mapping coordinates outside image size. The function definition is available in the file.

References

1. Warp Perspective - https://docs.opencv.org/2.4/modules/imgproc/doc/geometric_transformations.html#void%20warpPerspective(InputArray%20src,%20OutputArray%20dst,%20InputArray%20M,%20Size%20dsize,%20int%20flags,%20int%20borderMode,%20const%20Scalar&%20borderValue)
2. Projection matrix - https://bitesofcode.wordpress.com/2018/09/16/augmented-reality-with-python-and-opencv-part-2/