Project 1

Mrugesh J. Shah
117074109

A. JAMES CLARK
SCHOOL OF ENGINEERING

ENPM 673

—

PERCEPTION FOR

AUTONOMOUS ROBOTS

—

Dr. Mohammed Samer Charifa

Contents

Problem 1

The first problem primarily focuses on the detection of the April Tag and decoding it for the Tag ID and orientation of the tag id. This will be used for detecting the orientation data of the tag in the next proble.

1.1 Part 1

1.1.1 Introduction

The task is to detect the Tag in any one frame of the given videos, but for just a single frame. The catch here is that we are NOT supposed to use any built-in function for edge detection and use the FFT (Fast Fourier Transform) to generate the edged image for contour detection.

Allowed functions are built-in FFT function from OpenCV, NumPy or SciPy.

1.1.2 What is FFT in Image Processing?

FFT stands for Fast Fourier Transform, it is an algorithm to calculate the DFT (Discrete Fourier Transform). It is based on Fourier series, invented by mathematician named Jean-Baptiste Joseph Fourier in 1822.

The concept of Fourier Series is that it converts any analog or digital signal to a sum of Sine and Cosine functions of different harmonic frequencies. It is similar to Taylor's series the only difference is Taylor series factorize a signal or function in polynomial form while Fourier series does same in form of trigonometric form.

Applying FFT to an Image will give us the real and imaginary components which represent image in frequency domain. Which will be useful further filtering of the image based on frequency characteristics.

1.1.3 Edge detection

The edge in context of image is caused by change in either of these factors

- surface normal discontinuity
- depth discontinuity
- surface color discontinuity
- illumination discontinuity

Figure 1. is the illustration of each type of factor that causes an edge in an image, in nutshell, a change in gradient of the image. The higher the change of gradient, higher is the chance of that pixel being an edge.

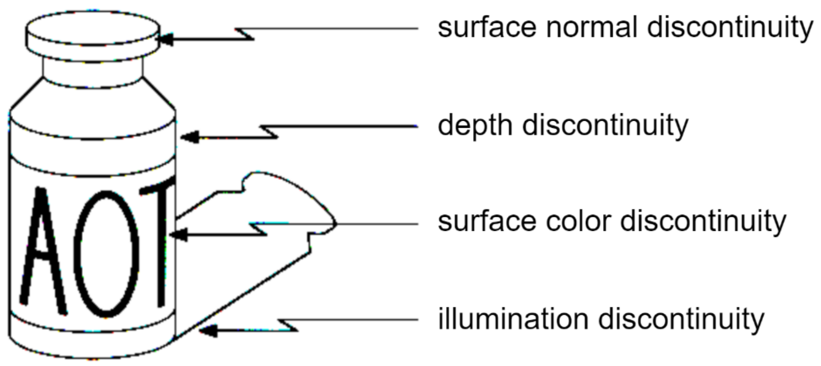

surface normal discontinuity
depth discontinuity
surface color discontinuity
illumination discontinuity
AOT

Figure 1. Factors causing Edge

First, we must understand what gradient is in an image, but we have to understand effect of blurring on an image and how it helps us with the edge detection.

The Edge detection works by looking for change in gradient, but what else causes change in gradient? “NOISE” causes change in gradient of an Image, thus, without using blurring we will have edges on the noise pixels as well.

Blurring Image will make sure that the high frequency noise has been removed form the image, and the edge detection works flawlessly on the image.

WARNING!!
Blurring image too much may cause the loss of desirable data.

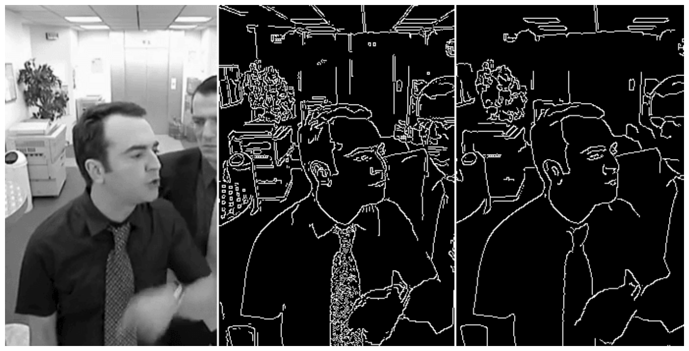


Figure 2. Effect of blurring on an Image edge detection

In figure 2, we can see that the blurring more will cause the edges in character’s tie to not detect. Blurring more will cause more features to not to be detected in the final frame.

The gradients are of many types based on the direction at which they measure the change of intensity. Figure 3. Will talk in more detail about this concept.

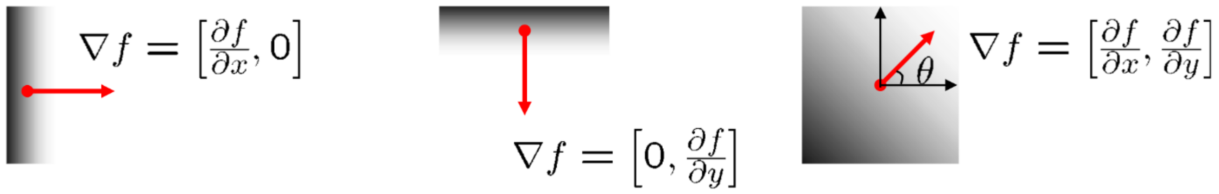

$\nabla f = \left[\frac{\partial f}{\partial x}, 0\right]$
$\nabla f = \left[0, \frac{\partial f}{\partial y}\right]$
θ
$\nabla f = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right]$

Figure 3. Gradients

The first gradient represents changes in X-Axis, this will detect the vertical edge and will not work with the horizontal edges. The second gradient represents change in Y-Axis, this will detect the horizontal edges and will not work with vertical edges. Both the gradients will work but work poorly with the third type of edge, which is an edge at an angle. For this, we will have to create a gradient that measures changes in both the axis. That's what third gradient represents.

1.1.4 Edge detection using FFT

The FFT is used to detect the edges by elimination the low frequency components of an image. The idea is similar to a high pass filter. The masking is used to eliminate the low frequency components. For better understanding, let's see what FFT does return when implemented on an Image.

FFT will return an array with real and imaginary component that represents the Fourier transform of the image, which is tough to visualize, thus we have to understand the Magnitude spectrum.

Magnitude spectrum is a spectrum that has low frequency at the center and the further you go; the frequency will be increased. The pixel intensity represents the value at that frequency. The calculation of magnitude spectrum is given by Eq. 1.

$$Magnitude = 20 \times \log\left(\sqrt{real^2} + imag^2 + 0.00001\right)$$

The 0.00001 will make sure that the there will not be a log(0) situation, and 20 is the scaling factor.

After plotting the image we have to remove the low frequency noise form the image, for that we will remove the center portion of the magnitude spectrum. The mask that we are going to use here is a circular mask. That has the radii of 80 pixels.

This will only let the high frequency data to pass through which only consists of the data about edges of the image.

1.1.5 Inverse FFT

After applying filter we have the high frequency data, which will give us the edges. We have to convert this from, the FFT to a normal image. This means we have to conver the image from real and imaginary component to a 8 bit unsigned integer format so that it can be supported by the OpenCV for visualization.

Inverse FFT is going to give a scaled data because we have added 0.000001 while finding the magnitude spectrum to avoid the log(0) condition. Thus we have to scale the data to 0 to 255 which is the minimum and maximum for 8 bit unsigned integers.

1.1.6 Detection Pipeline

In this section, I will talk about my implementation of the algorithm, in python with snippets of the derived results.

First, I captured the video data and choose a single frame from the video to process the tag, for this case, I took the 50th frame in Tag1.mp4 file. The frame is shown in Figure 4.

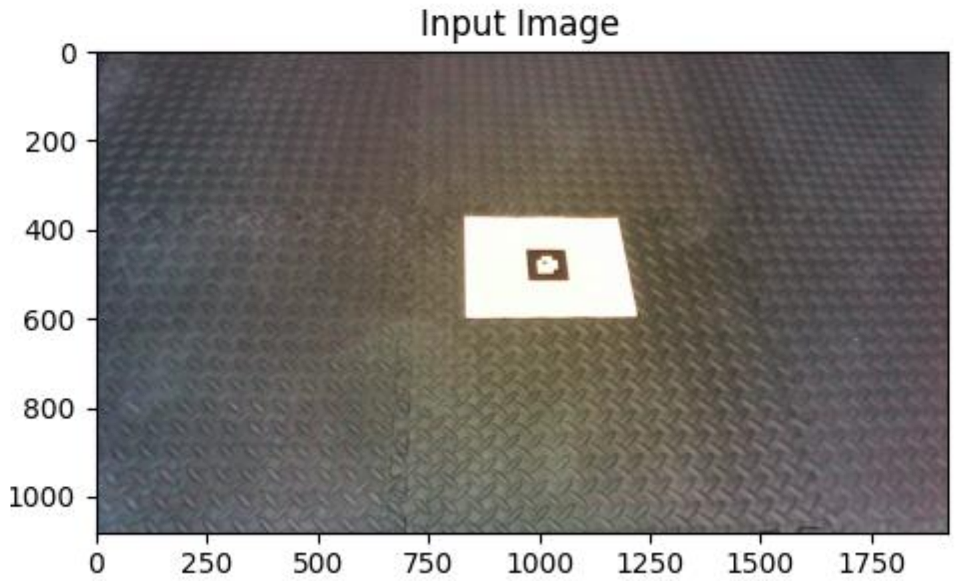


Figure 4. Chosen frame no. 50

Then I implemented the FFT by using the OpenCV's cv2.dft() function, which uses FFT algorithm to calculate DFT. The result image has the zero frequency components on the corner of the image, thus I had to use np.fft.fftshift() function to put the low frequency component to the center of the magnitude spectrum.

In Figure 5. The FFT transformed and centralized magnitude spectrum is shown.

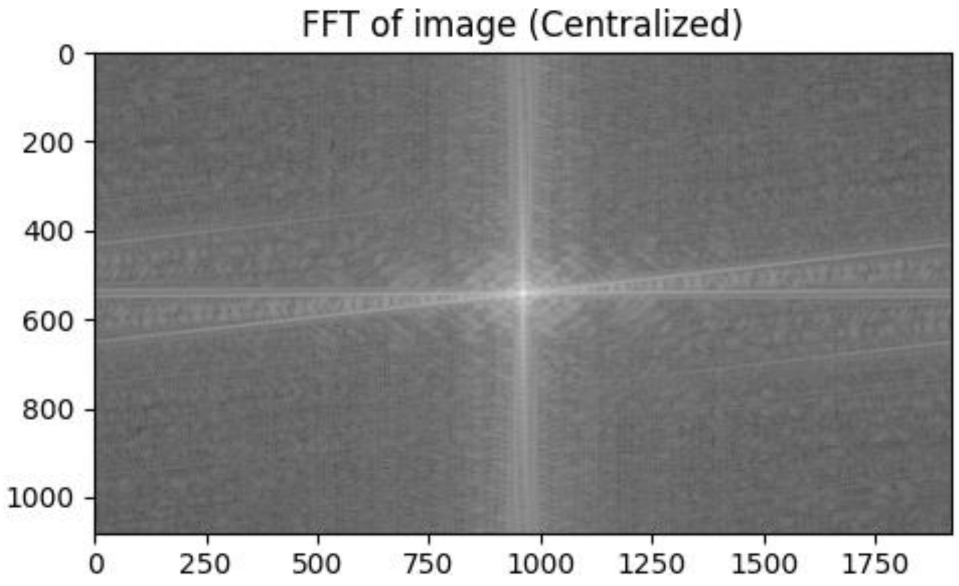


Figure 5. FFT of Image

Now, we have to implement the High pass filter to the FFT of image to remove the image's low frequency components. The mask here used is a circular mask which is better than the square mask but is worse than the gaussian mask which will have faded edges and a shape like circle. The equation of the circle is given by,

```
masking_portion = (X - CenR) ** 2 + (Y - CenC) ** 2 <= radii*radii
```

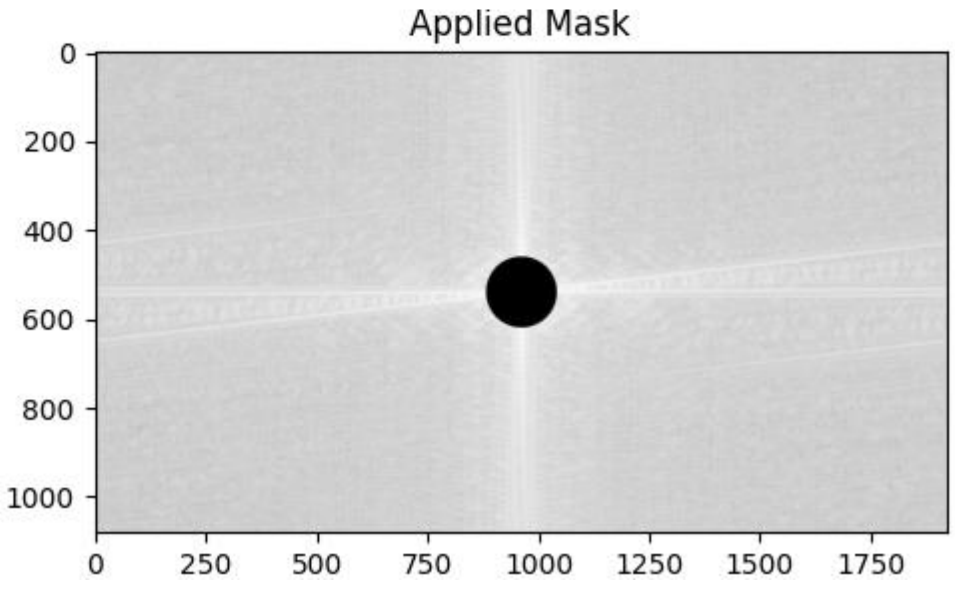


Figure 6. Masked Image

Note:- The image is a bit lighter, as the Matplotlib works by scaling the magnitude in min-max format, and as now the min value in image is zero, the respective values of all pixels is increased.

Before applying inverse FFT, the scaling has to be done as we have added a DC shift to calculate the magnitude. The scaling ha to be done manually as using cv2 function will just convert the type of data and cause the loss of data by oversaturation of image. The code uses self defined function,

```
convert(img, target_type_min, target_type_max, target_type):
```

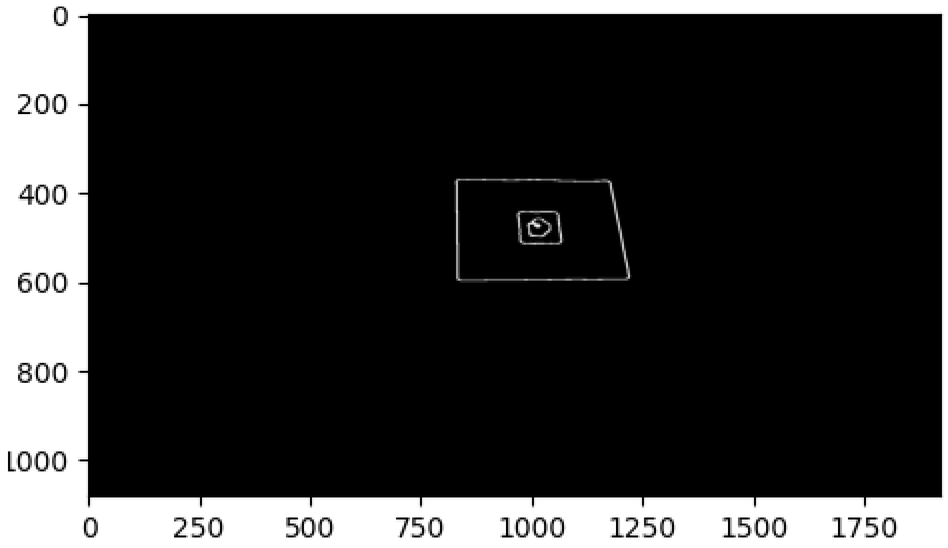


Figure 7. Edge detection

Now, we have to implement contour detection on the image which will return the Tag coordinates which we will use to crop the Tag out of the image. This can be done by accessing the desired pixels from the image.

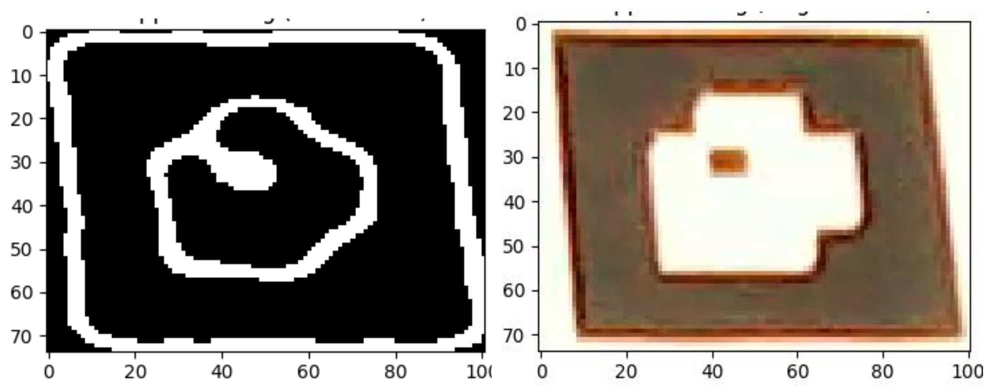


Figure 8. Showing detected tag.

1.2 Part 2

The task here is to decode the tag given as reference, and generate the orientation data as well as the Tag ID, that consists or inner 4 pixels of the grid and outer most corner pixels respectively.

1.2.1 Introduction

The given tag is called an April Tag. For this particular type of Tag, the gird is of 8x8 with outer most two are padding, and internal 4x4 grid consists of the data. The padding is provided for better contrast for the edge detection.

1.2.2 What is April Tag?

The April Tag has applications in the domain of AR (Augmented Reality), Robotics, and calibration of cameras. The 3D position, identity, and orientation can be deduced form the photo of an April Tag and intrinsic parameter of the camera.

The given Tag is a simple square type but there are more complex with increased grid size and different shapes.

1.2.3 Orientation and Tag ID

There has been given an image of the tag for reference to which we have to decode and give the orientation and tag ID data, the image has been shown in Figure 9.

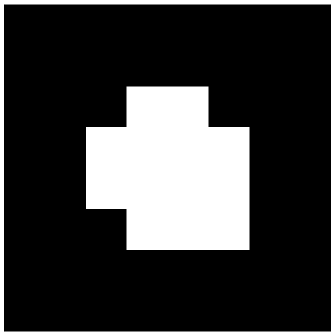


Figure 9. Reference ID

We can Imagine a grid of 8x8 over the reference image which will give us 64 bits of data with location. The grid is shown in Figure 10.

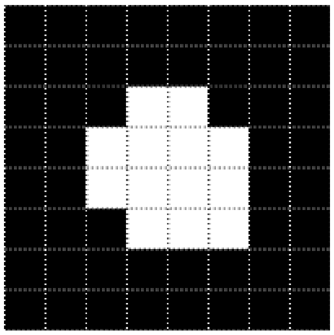


Figure 10. Grid over reference Tag

The outer most 2 pixels are padding and removing them will give us just the data we want. The next step will be to remove the padding.

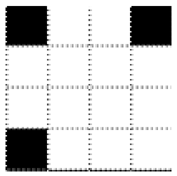


Figure 11. Removed padding

The inner most 4 bits or the 2x2 grid size will give the Tag ID, in clockwise fashion. Which means the MSB (Most Significant Bit) will be the first pixel from top right, for the given reference all of them are 1, the Tag ID will be “1111”.

The corner bits represent the orientation, there is only a single corner pixel which is white, and that is our orientation data. We have to detect the location of the white corner pixel. The returned data will be either of the following

- BR – Bottom Right
- TR – Top Right
- TL – Top Left
- BL – Bottom Left

For this case its in BR (Bottom Right) position.

1.2.4 Detection Pipeline

The program first fetches the image and converts it into a binary image with threshold of 250. The result image will be measured for it’s shape with img.shape[x] function. This will decide the grid’s pixel size corresponding to the image.

In our case, the image is of 200x200 pixels, thus each of 64 pixels in grid will represent a 25x25 pixels of the image. Note that there is a offset of 12.5 pixels to get the middle pixel of each cell.

Then we will remove the padding by removing the first and bottom two rows as well as first and last two columns. The data we have now will look like following.

$$Grid\_unpadded = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix}$$

Accessing each element of array and decoding it with simple If… else... statements will give us the Tag ID, and orientation, which I have printed on the Tag its self. The Figure 12 represents the solution.

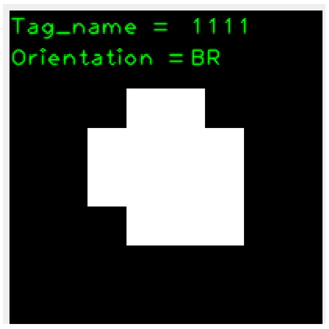


Figure 12. Result

Problem 2

In this problem, we have to implement what we archived in problem 1 in the real world application.

2.1 Part 1

The video of a April Tag on the floor is given and the camera is moving in random motion, the task is to show the Testudo image on the tag in proper orientation all the time.

2.1.1 Introduction

The challenging thing here is to detect the contour of the April Tag and decoding it properly to get the orientation of the tag. For that, we have to detect the proper contour and for that we need to understand the hierarchy.

2.1.2. Hierarchy of contours

The cv2.Findcontours() function will gives us all the contours of different sizes and shapes. But, we want to have contour that has a specific characteristic. For that we have to understand what parent-child relation ship of contours is.

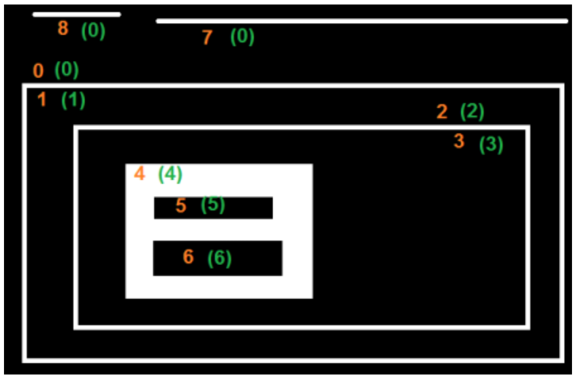


Figure 13. Hierarchy of contours

Here, while using the find contour function I passed an argument, cv2.RETR_TREE. Which will give us the details of all the contours with the index of their parent and their child. The issue we are facing here is that we are detecting several contours but we want a contour which has a parent as well as a child.

For example, in our case, the parent will be the paper on which the Tag is printed and the child will be the inner Tag without the padding. If a contour has both a parent and a child, then it is an contour of interest.

2.1.3 What is Homography?

The idea behind homography is to project an image from our perspective to the perspective of the camera. The homography matrix is a 3x3 matrix with [3, 3] element usually 1 representing its scale invariance.

Homography matrix is used to project an image on a plane that is not aligning with the current frame of reference either parallelly or perpendicularly. To put it into simpler words, let’s say we have an image in rectangular shape, and we want it to place on a plain that extends to horizon. The shape of image will now become trapezoidal.

It is used to add the 3D effect to a 2D image as well. This matrix will transform the image to give it a perspective that matches with the scene. One of the major field that uses Homography matrix is Augmented Reality (AR).

The use of the matrix is to convert the homogenous coordinates of the one frame to another with the help of the Homography matrix. For that, you have to generate the matrix and that will require at least four points pair. The following is the homography matrix calculation method.

$$A = \begin{bmatrix} -x_1 & -y_1 & -1 & 0 & 0 & 0 & x_1 * x_{p1} & y_1 * x_{p1} & x_{p1} \\ 0 & 0 & 0 & -x_1 & -y_1 & -1 & x_1 * y_{p1} & y_1 * y_{p1} & y_{p1} \\ -x_1 & -x_1 & -1 & 0 & 0 & 0 & x_2 * y_{p2} & y_2 * x_{p2} & x_{p2} \\ 0 & 0 & 0 & -x_2 & -y_2 & -1 & x_2 * y_{p2} & y_2 * y_{p2} & y_{p2} \\ -x_1 & -x_1 & -1 & 0 & 0 & 0 & x_3 * x_{p3} & y_3 * x_{p3} & x_{p3} \\ 0 & 0 & 0 & -x_3 & -y_3 & -1 & x_3 * y_{p3} & y_3 * y_{p3} & y_{p3} \\ -x_1 & -x_1 & -1 & 0 & 0 & 0 & x_4 * y_{p4} & y_4 * x_{p4} & x_{p4} \\ 0 & 0 & 0 & -x_4 & -y_4 & -1 & x_4 * y_{p4} & y_4 * y_{p4} & y_{p4} \end{bmatrix}, \quad x = \begin{bmatrix} H_{11} \\ H_{12} \\ H_{13} \\ H_{21} \\ H_{22} \\ H_{23} \\ H_{31} \\ H_{32} \\ H_{33} \end{bmatrix}$$

The x_x, y_x points are the source and x_{px}, y_{px} are the destination points. NumPy’s SVD function will take care of solving for the homography matrix. But, we have to reshape the matrix to 3x3 shape for further calculation.

```
U, S, Vh = np.linalg.svd(A)

H_col = Vh[-1, :] / Vh[-1, -1]

H = H_col.reshape(3, 3)
```

2.1.4 What is Warping of Image?

Now that we have the homography matrix from source to destination, we can do the perspective transform of the image. The perspective transform or the warping of the image will give us the image that looks like as if seen from a different angle. The Figure 14 will give us the example of how the image warping works

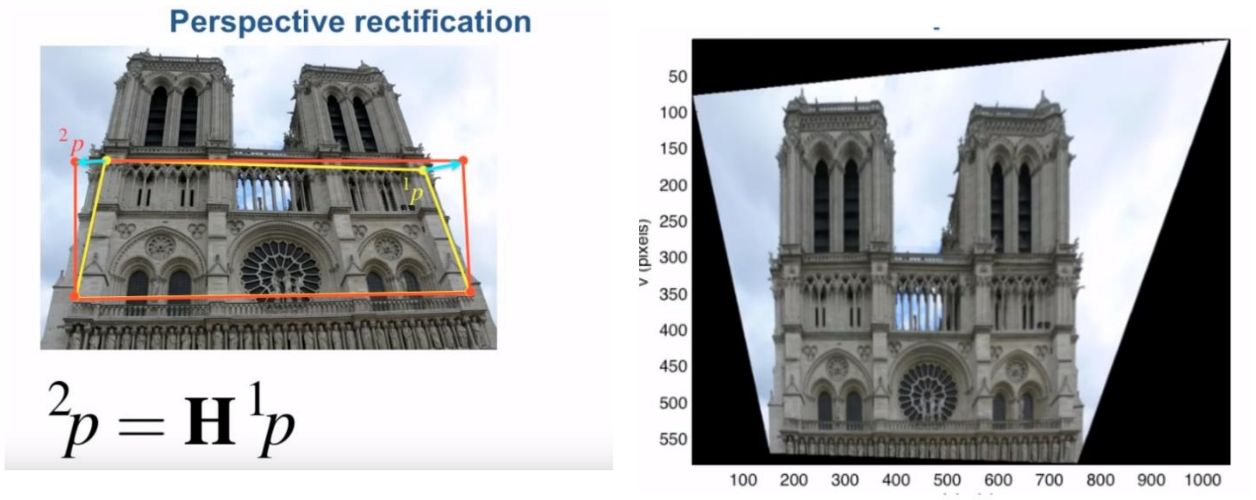
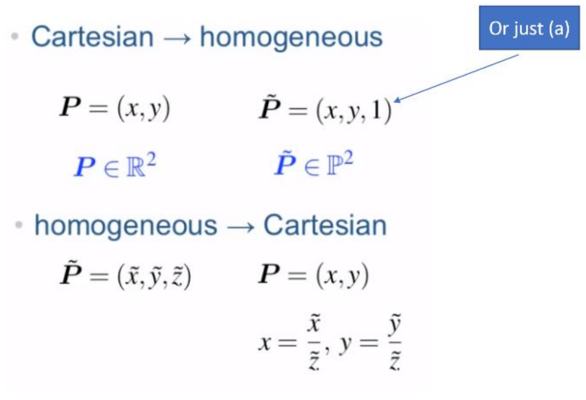


Figure 14. Perspective Transform

Here we have multiplied each pixel with the homography matrix, this will give us the new location of the pixel in warped image. It is important to note that all the calculation is happening in the Homogeneous coordinates and thus we have to understand the conversion from homogenous to cartesian coordinates. Figure 15. Explains how to do that.



- Cartesian → homogeneous

$\boldsymbol{P} = (x, y)$ $\quad$ $\tilde{\boldsymbol{P}} = (x, y, 1)$ Or just (a)

$\boldsymbol{P} \in \mathbb{R}^2$ $\quad$ $\tilde{\boldsymbol{P}} \in \mathbb{P}^2$

- homogeneous → Cartesian

$\tilde{\boldsymbol{P}} = (\tilde{x}, \tilde{y}, \tilde{z})$ $\quad$ $\boldsymbol{P} = (x, y)$

$$x = \frac{\tilde{x}}{\tilde{z}}, \; y = \frac{\tilde{y}}{\tilde{z}}$$

Figure 15. Conversion of coordinate systems

The use of warping is to get the bird eye view of the April Tag to apply the decoding algorithm to get the orientation data.

2.1.5 Pipeline

At the beginning we have to give user a prompt to select the video on which user wants to place the tag on. There are four option for that and simple If… else… is used to fetch that corresponding video.

The first step is to run the video frame by frame and apply the edge detection algorithm and find all the available contours on a given frame. Then we have to use the hierarchy matrix to get the contour that has a parent and a child. Then we have to sort the contours based on size and then the first three contours will be out contours.

The cv2.approxPolyDP() will make the contours a polygon and if that polygon has four sides, then it is our contour. The final list of contours is the our Tag. We can find the corners easily as we applied cv2.approxPolyDP().

Now, what we have is something like Figure 15.

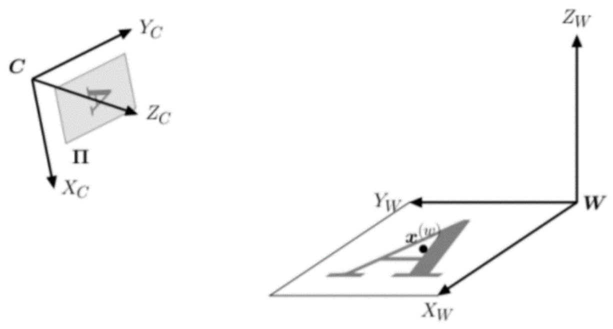


Figure 15. Tag coordinates

Now, we know the size of the image that we want to project, which is 500x500, Figure 16. is the testudo image we want to project.

Figure 16. Testudo

We have to find the homography between source coordinates from the image to [(0,0), (500, 0), (500, 500), (0, 500)].

```
H = find_homography(np.float32(src_pts), np.float32(dst_pts))
```

The homography matrix can be used to get the perspective transform using the following function

```
warped = warp_perspective(H, frame, 500, 500)
```

The warped image will look like figure 17. After binarization of it.

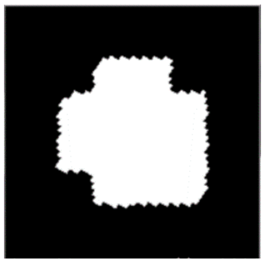


Figure 17. Detected April Tag
(Bird-eye view)

The next step is to detect the orientation, for which I updated the algorithm from Problem 1, Part 2 to take the mode point of the cell, rather than the mid or center point. The results showed improvement when detecting a blurry tag.

According to orientation data, we use simple If… else… structure to rotate the source image it self in proper orientation, so now we just have to super impose the image on the frame. But, simple superimposing the image will cause errors or black spots on the image, as shown in supplementary document. For that, we have to use masking. The mask of the inverse warped image of Testudo (Figure 16.) is placed on the frame. Now, when there is a black area where the inverse warped image is to be located, we can just use cv2.bitwise_and() to make sure it has no noise because of the background. The reverse warping will require inverse of the Homography matrix that we used to get the bird-eye view. The result of the final warping will look like Figure 18.

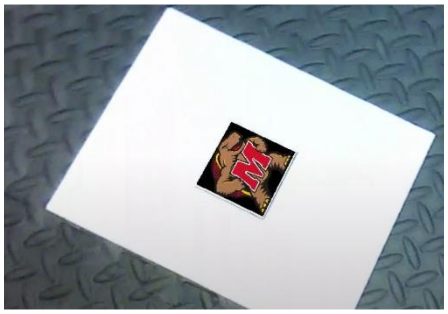


Figure 18. Projected Tag

2.2 Part 2

The task here is to place a cube over the detected tag. The cube must look 3D. The user selects the video on which he/she want to place the cube. The choices are same as Problem 2, Part 1.

2.2.1 Introduction

To accomplish this task, only the homography will not work, as the homography matrix gives us only the 2D projection of the image. The cube is a 3D element; thus we need one more matrix that consists the data of the camera parameters such as,

- Principle point coordinates (P_x, P_y)
- Focal Length f
- Pixel magnification factors m
- Skew s

That is when the projection matrix comes into play

2.2.2 What is projection matrix?

The projection matrix consists of the data that are mentioned in the introduction as well as the homography matrix. First, we have to understand what the intrinsic matrix (K) is,

The matrix K is define or structured as follows.

$$K = \begin{bmatrix} f & s & p_x \\ 0 & mf & p_y \\ 0 & 0 & 1 \end{bmatrix}$$

For us, the matrix K is given in the question, which is,

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

To match the given format, we have to take the transpose of K matrix given too us and store it as K only.

The fundamentals of projection or projecting a 3D object on a 2D plane can be better understood by the following Figure 19.

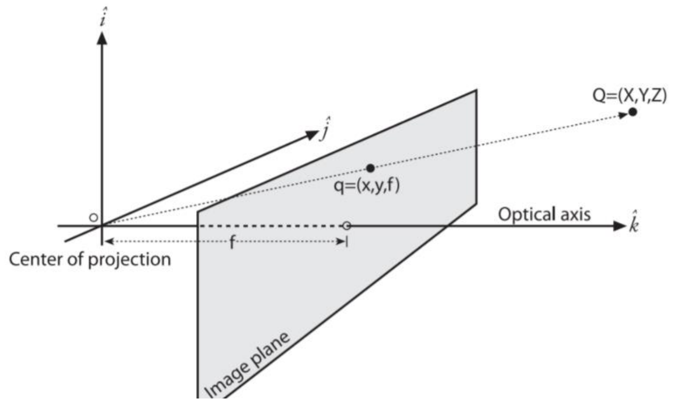


Figure 19. Projection

The 3D point has been projected on the 2D plane and the 2D plane is between the center of projection and that 3D point. This model is used to simplify the mathematics required for further calculations.

Following calculation will give us the projection matrix, note that the value of lambda (λ), the scaling parameter is calculated based on the homography matrix.

Assume all points lie in one plane with Z=0:

$$\boldsymbol{X} = (X, Y, 0, 1)$$

$$\begin{aligned} \boldsymbol{x} &= \boldsymbol{PX} \\ &= \boldsymbol{K}[\boldsymbol{r}_1\boldsymbol{r}_2\boldsymbol{r}_3\boldsymbol{t}]\begin{pmatrix} X \\ Y \\ 0 \\ 1 \end{pmatrix} \\ &= \boldsymbol{K}[\boldsymbol{r}_1\boldsymbol{r}_2\boldsymbol{t}]\begin{pmatrix} X \\ Y \\ 1 \end{pmatrix} \\ &= \boldsymbol{H}\begin{pmatrix} X \\ Y \\ 1 \end{pmatrix} \end{aligned}$$

$$\boldsymbol{H} = \lambda\boldsymbol{K}[\boldsymbol{r}_1\boldsymbol{r}_2\boldsymbol{t}]$$

$$\boldsymbol{K}^{-1}\boldsymbol{H} = \lambda[\boldsymbol{r}_1\boldsymbol{r}_2\boldsymbol{t}]$$

–r_1 and r_2 are unit vectors ⇨ find lambda
–Use this to compute t
–Rotation matrices are orthogonal ⇨ find r3

$$\boldsymbol{P} = \boldsymbol{K}\begin{bmatrix} \boldsymbol{r}_1 & \boldsymbol{r}_2 & (\boldsymbol{r}_1 \times \boldsymbol{r}_2) & \boldsymbol{t} \end{bmatrix}$$

Note that, the projection matrix is as well in the homogeneous format. The result we get from the multiplying a point of 3D space to projection matrix will give us the homogenous coordinates of 2D plane. With which we have to get the cartesian coordinates of that corresponding point. This can be done by using method shown in Problem 2, Part 1.

2.2.3 Pipeline

The first two steps here are similar to Problem 2, Part 1. The detection of the proper contour has been done and we have the coordinates of the corners of each contours.

The next step is to find the homography matrix as explained earlier, note that here is no need to calculate the orientation of the tag. The next step is to calculate the projection matrix. I have calculated the Projection matrix with self-defined function.

```
P = projection_matrix(H_cube, K)
```

Where the H_cube is homography matrix and K is transpose of given intrinsic matrix.

```
lambda_ * np.matmul(np.linalg.inv(K), h)
```

```
rotation = np.column_stack((r1, r2, r3, t))
```

```
projection_mat = np.matmul(K, rotation)
```

The r1, r2 and t are the column vectors of the matrix resulted from multiplication done above, while r3 is the cross product of r1 and r2.

Now, we have to multiply the projection matrix to each element in the 3D space (500x500x500) cube, to get the 2D coordinates in homogenous from. Then, use the method explained in previous part, convert them into cartesian format and draw the line using OpenCV function cv2.line().

The result will look like Figure 20.

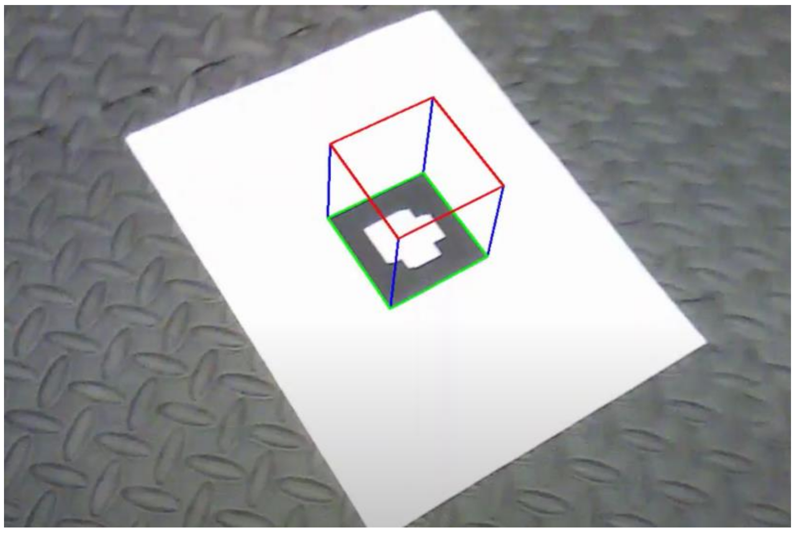


Figure 20. Result of Problem 2 Part 2

References

Edge Detection in Opencv 4.0, A 15 Minutes Tutorial | Sicara

Edge Detection in Opencv 4.0, A 15 Minutes Tutorial | Sicara. (2021). Retrieved 8 March 2021, from https://www.sicara.ai/blog/2019-03-12-edge-detection-in-opencv

Fourier series

Fourier series. (2021). Retrieved 8 March 2021, from https://en.wikipedia.org/wiki/Fourier_series#

Edge detection

Edge detection. (2021). Retrieved 8 March 2021, from https://en.wikipedia.org/wiki/Edge_detec

OpenCV: Contours Hierarchy

OpenCV: Contours Hierarchy. (2021). Retrieved 8 March 2021, from https://docs.opencv.org/master/d9/d8b/tutorial

Singular Value Decomposition (SVD) tutorial

Singular Value Decomposition (SVD) tutorial. (2021). Retrieved 15 February 2021, from https://web.mit.edu/be.400/www/SVD/Sing

Finding Homography Matrix using Singular-value Decomposition and RANSAC in OpenCV and Matlab - Robotics with ROS

Finding Homography Matrix using Singular-value Decomposition and RANSAC in OpenCV and Matlab - Robotics with ROS. (2017). Retrieved 15 February 2021 from http://ros-developer.com/2017/12/26/finding-homography-matrix-using-singular-value-decomposition-and-ransac-in-opencv-and-matlab/

Singular value decomposition

Singular value decomposition. (2021). Retrieved 15 February 2021, from https://en.wikipedia.org/wiki/Singular_value_decomposition

Neto, J.

Neto, J. (2021). Singular Value Decomposition. Retrieved 15 February 2021, from http://www.di.fc.ul.pt/~jpn/r/svd/svd.html

Installation Guide — Matplotlib 3.3.4 documentation

Installation Guide — Matplotlib 3.3.4 documentation. (2021). Retrieved 15 February 2021,
from https://matplotlib.org/stable/users/installing.htm

OpenCV: Introduction to OpenCV-Python Tutorials

OpenCV: Introduction to OpenCV-Python Tutorials. (2021). Retrieved 15 February 2021,
from https://docs.opencv.org/master/d0/de3/tutorial_py_intro.html

OpenCV: Install OpenCV-Python in Ubuntu
OpenCV: Install OpenCV-Python in Ubuntu. (2021). Retrieved 15 February 2021,
from https://docs.opencv.org/master/d2/de6/tutorial_py_setup_in_ubuntu.html

Homography (computer vision)
Homography (computer vision). (2021). Retrieved 15 February 2021,
from https://en.wikipedia.org/wiki/Homography