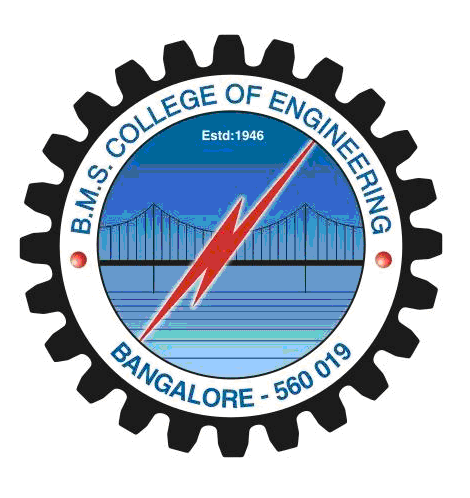
**B.M.S COLLEGE OF ENGINEERING**

**Department of Electronics and Communication Engineering**

Bull Temple Road, Basavanagudi, Bangalore - 560 019



DSP SELF STUDY REPORT

on

**“IMAGE PROCESSING USING RASPBERRY PI”**

Submitted by

|  |  |
| --- | --- |
| **Name of the Student** | **USN** |
| **DHEERAJ D KAMATH** | **1BM14EC027** |
| **DIGANTH P** | **1BM14EC028** |
| **G BALAJI** | **1BM14EC034** |
| **HARSHITH J** | **1BM14EC035** |

**Name of the Faculty In charge: S.LALITHA**

**Designation: Assistant professor, ECE Department**

Contents

[ABSTRACT 3](#_Toc467922415)

[INTRODUCTION 3](#_Toc467922416)

[THE RASPBERRY PI 4](#_Toc467922417)

[IMAGE PROCESSING 5](#_Toc467922418)

[THRESHOLDING 5](#_Toc467922419)

[CODE 5](#_Toc467922420)

[CONTOUR 6](#_Toc467922421)

[CONTOUR APPROXIMATION 7](#_Toc467922422)

[POSE ESTIMATION 8](#_Toc467922423)

[ARUCO MARKERS 10](#_Toc467922424)

[THE CODING TECHNIQUE 11](#_Toc467922425)

[PYTHON CODE 12](#_Toc467922426)

[CONCLUSION 20](#_Toc467922427)

[REFERENCES 21](#_Toc467922428)

# ABSTRACT

Aruco Markers find extensive use in Augmented Reality (AR). They are easy and a beginner’s path to enter the world of understanding how AR works.

Our work focusses mainly on

* We will see how we can distinguish between real world objects and the marker by identifying features unique to the marker.
* Detection, pose estimation and decoding of the marker.
* Understanding Image Processing library OpenCV
* Programming using Python IDE
* Working with the Raspberry Pi Model B+
* Understanding the Linux Environment

# INTRODUCTION

Pose estimation is of great importance in many computer vision applications: robot navigation, augmented reality, and many more. This process is based on finding correspondences between points in the real environment and their 2d image projection. This is usually a difficult step, and thus it is common the use of synthetic or fiducial markers to make it easier.

One of the most popular approach is the use of binary square fiducial markers. The main benefit of these markers is that a single marker provides enough correspondences (its four corners) to obtain the camera pose. Also, the inner binary codification makes them specially robust, allowing the possibility of applying error detection and correction techniques.The aruco module is based on the Aruco Library, a popular library for detection of square fiducial markers developed by Rafael Muñoz and Sergio Garrido.

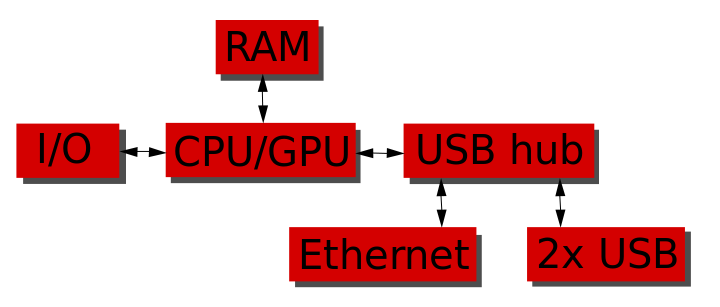
In upcoming pages we see how we can detect these markers from the real world objects and also decode the marker to get its id by performing several image processing and mathematical techniques.

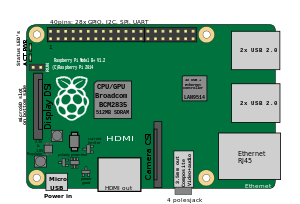
Raspberry Pi is one of the famous computer chips available which offers several capabilities for image processing. The project shows how image processing is realized using the Python IDE in the chip.

# THE RASPBERRY PI

All models feature a Broadcom system on a chip (SoC), which includes an ARM compatible central processing unit (CPU) and an on chip graphics processing unit (GPU, a VideoCore IV). CPU speed ranges from 700 MHz to 1.2 GHz for the Pi 3 and on board memory range from 256 MB to 1 GB RAM. Secure Digital (SD) cards are used to store the operating system and program memory in either the SDHC or MicroSDHC sizes. Most boards have between one and four USB slots, HDMI and composite video output, and a 3.5 mm phone jack for audio. Lower level output is provided by a number of GPIO pins which support common protocols like I²C. The B-models have an 8P8C Ethernet port and the Pi 3 has on board Wi-Fi 802.11n and Bluetooth.

The Foundation provides Raspbian, a Debian-based Linux distribution for download, as well as third party Ubuntu, Windows 10 IOT Core, RISC OS, and specialised media center distributions. It promotes Python and Scratch as the main programming language, with support for many other languages.





# IMAGE PROCESSING

## THRESHOLDING

Here, the matter is straight forward. If pixel value is greater than a threshold value, it is assigned one value (may be white), else it is assigned another value (may be black). The function used is **cv2.threshold**. First argument is the source image, which **should be a grayscale image**. Second argument is the threshold value which is used to classify the pixel values. Third argument is the maxVal which represents the value to be given if pixel value is more than (sometimes less than) the threshold value. OpenCV provides different styles of thresholding and it is decided by the fourth parameter of the function. Different types are:

* cv2.THRESH\_BINARY
* cv2.THRESH\_BINARY\_INV
* cv2.THRESH\_TRUNC
* cv2.THRESH\_TOZERO
* cv2.THRESH\_TOZERO\_INV

## CODE

**import** cv2

**import** numpy **as** np

**from** matplotlib **import** pyplot **as** plt

img **=** cv2**.**imread('gradient.png',0)

ret,thresh1 **=** cv2**.**threshold(img,127,255,cv2**.**THRESH\_BINARY)

ret,thresh2 **=** cv2**.**threshold(img,127,255,cv2**.**THRESH\_BINARY\_INV)

ret,thresh3 **=** cv2**.**threshold(img,127,255,cv2**.**THRESH\_TRUNC)

ret,thresh4 **=** cv2**.**threshold(img,127,255,cv2**.**THRESH\_TOZERO)

ret,thresh5 **=** cv2**.**threshold(img,127,255,cv2**.**THRESH\_TOZERO\_INV)

titles **=** ['Original Image','BINARY','BINARY\_INV','TRUNC','TOZERO','TOZERO\_INV']

images **=** [img, thresh1, thresh2, thresh3, thresh4, thresh5]

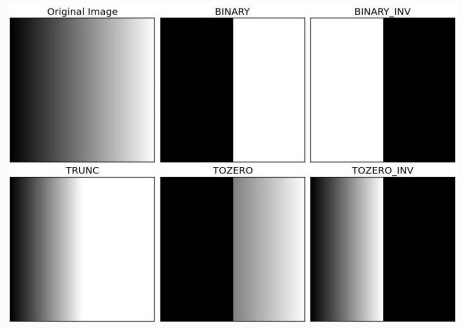
**for** i **in** xrange(6):

plt**.**subplot(2,3,i**+**1),plt**.**imshow(images[i],'gray')

plt**.**title(titles[i])

plt**.**xticks([]),plt**.**yticks([])

plt**.**show()



## CONTOUR

Contours can be explained simply as a curve joining all the continuous points (along the boundary), having same color or intensity. The contours are a useful tool for shape analysis and object detection and recognition.For better accuracy, use binary images. So before finding contours, apply threshold or canny edge detection.

findContours function modifies the source image. So if you want source image even after finding contours, already store it to some other variables.

In OpenCV, finding contours is like finding white object from black background. So remember, object to be found should be white and background should be black.

Let’s see how to find contours of a binary image:

**import** numpy **as** np

**import** cv2

im **=** cv2**.**imread('test.jpg')

imgray **=** cv2**.**cvtColor(im,cv2**.**COLOR\_BGR2GRAY)

ret,thresh **=** cv2**.**threshold(imgray,127,255,0)

image, contours, hierarchy **=** cv2**.**findContours(thresh,cv2**.**RETR\_TREE,cv2**.**CHAIN\_APPROX\_SIMPLE)

See, there are three arguments in **cv2.findContours()** function, first one is source image, second is contour retrieval mode, third is contour approximation method. And it outputs the image, contours and hierarchy. Contours is a Python list of all the contours in the image. Each individual contour is a Numpy array of (x,y) coordinates of boundary points of the object.

# CONTOUR APPROXIMATION

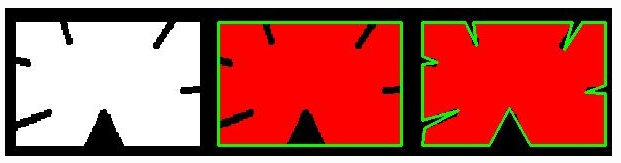
It approximates a contour shape to another shape with less number of vertices depending upon the precision we specify. It is an implementation of Douglas-Peucker algorithm. Check the Wikipedia page for algorithm and demonstration.

To understand this, suppose you are trying to find a square in an image, but due to some problems in the image, you didn’t get a perfect square, but a “bad shape” (As shown in first image below). Now you can use this function to approximate the shape. In this, second argument is called epsilon, which is maximum distance from contour to approximated contour. It is an accuracy parameter. A wise selection of epsilon is needed to get the correct output.

epsilon **=** 0.1**\***cv2**.**arcLength(cnt,True)

approx **=** cv2**.approxPolyDP**(cnt,epsilon,True)

Below, in second image, green line shows the approximated curve for epsilon = 10% of arc length. Third image shows the same for epsilon = 1% of the arc length. Third argument specifies whether curve is closed or not.



## POSE ESTIMATION

This is going to be a small section. During the last session on camera calibration, you have found the camera matrix, distortion coefficients etc. Given a pattern image, we can utilize the above information to calculate its pose, or how the object is situated in space, like how it is rotated, how it is displaced etc. For a planar object, we can assume Z=0, such that, the problem now becomes how camera is placed in space to see our pattern image. So, if we know how the object lies in the space, we can draw some 2D diagrams in it to simulate the 3D effect. Let’s see how to do it.

Our problem is, we want to draw our 3D coordinate axis (X, Y, Z axes) on our chessboard’s first corner. X axis in blue color, Y axis in green color and Z axis in red color. So in-effect, Z axis should feel like it is perpendicular to our chessboard plane.

First, let’s load the camera matrix and distortion coefficients from the previous calibration result.

let’s create a function, draw which takes the corners in the chessboard (obtained using cv2.findChessboardCorners()) and axis points to draw a 3D axis.

**import** cv2

**import** numpy **as** np

**import** glob

*# Load previously saved data*

**with** np**.**load('B.npz') **as** X:

mtx, dist, \_, \_ **=** [X[i] **for** i **in** ('mtx','dist','rvecs','tvecs')]

**def** **draw**(img, corners, imgpts):

corner **=** tuple(corners[0]**.**ravel())

img **=** cv2**.**line(img, corner, tuple(imgpts[0]**.**ravel()), (255,0,0),5)

img **=** cv2**.**line(img, corner, tuple(imgpts[1]**.**ravel()), (0,255,0),5)

img **=** cv2**.**line(img, corner, tuple(imgpts[2]**.**ravel()), (0,0,255),5)

**return** img

**for** fname **in** glob**.**glob('left\*.jpg'):

img **=** cv2**.**imread(fname)

gray **=** cv2**.**cvtColor(img,cv2**.**COLOR\_BGR2GRAY)

ret, corners **=** cv2**.**findChessboardCorners(gray, (7,6),None)

**if** ret **==** True:

corners2 **=** cv2**.**cornerSubPix(gray,corners,(11,11),(**-**1,**-**1),criteria)

*# Find the rotation and translation vectors.*

rvecs, tvecs, inliers **=** cv2**.**solvePnPRansac(objp, corners2,mtx, dist)

*# project 3D points to image plane*

imgpts, jac **=** cv2**.**projectPoints(axis, rvecs, tvecs, mtx, dist)

img **=** draw(img,corners2,imgpts)

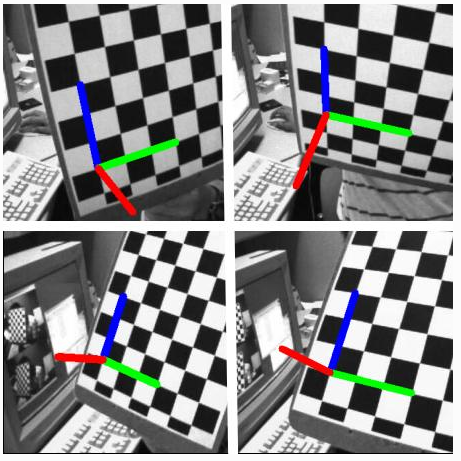
cv2**.**imshow('img',img)

k **=** cv2**.**waitKey(0) **&** 0xff

**if** k **==** 's':

cv2**.**imwrite(fname[:6]**+**'.png', img)

cv2**.**destroyAllWindows()

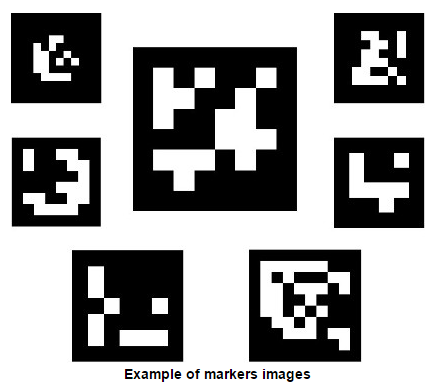


# ARUCO MARKERS

ArUco markers are binary square fiducial markers that can be used for camera pose estimation. Their main benefit is that their detection is robust, fast and simple.

An ArUco marker is a synthetic square marker composed by a wide black border and a inner binary matrix which determines its identifier (id). The black border facilitates its fast detection in the image and the binary codification allows its identification and the application of error detection and correction techniques. The marker size determines the size of the internal matrix. For instance, a marker size of 4x4 is composed by 16 bits.

Some examples of ArUco markers:

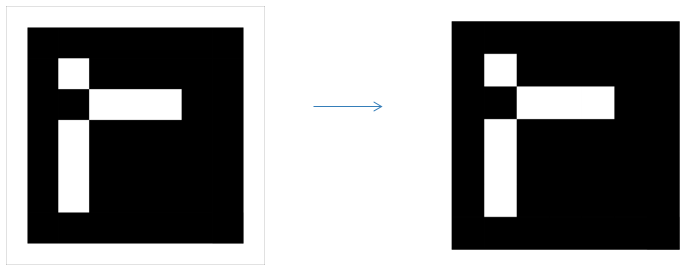


the first, third and fifth columns represent parity bits. The second and fourth columns represent the data bits.

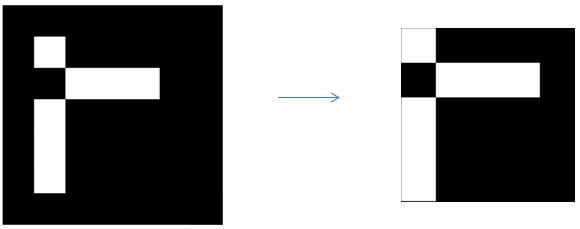
Hence, there are ten total data bits. So the maximum number of markers that can be encoded are 2^10 = 1024

## THE CODING TECHNIQUE

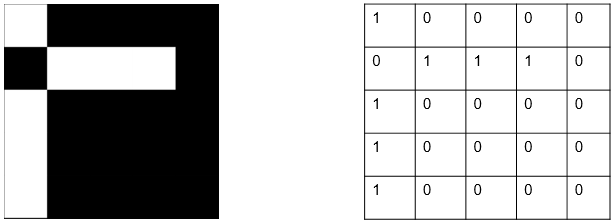
Step 1: - Extract the ArUco from the image



Step 2: - Remove the extra padding



STEP 3: - Divide the resulting image into a 5x5 grid and check the color in each cell of the second and fourth columns(in that order) in a top to bottom manner.



Step 4: If the color is white, write 1; else, write it 0.

Step 5: The resulting number will be in binary. Convert it into decimal

This is how the id of the marker is determined.

## PYTHON CODE

**Python code for decoding of aruco markers is as follows**

import cv2

#Define globals

MAX\_SIZE = 406

#Define helper functions

def extractAruco(img):

#Inverse threshold to get the inner contour

gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

ret, th = cv2.threshold(gray, 127,255,cv2.THRESH\_BINARY\_INV)

#Crop image

contours, hierarchy = cv2.findContours(th, cv2.RETR\_EXTERNAL,

cv2.CHAIN\_APPROX\_SIMPLE)

contours = contours[0]

epsilon = 0.1\*cv2.arcLength(contours,True)

approx = cv2.approxPolyDP(contours,epsilon,True)

image\_crop=gray[approx[0,0,1]:approx[2,0,1],approx[0,0,0]:approx[2,0,0]]

cv2.imshow("Crop", image\_crop)

#Return extracted image

return image\_crop

def findArucoID(inp\_img):

#Extract image

im = extractAruco(inp\_img)

#Resize it to smaller size for check

im = cv2.resize(im, (MAX\_SIZE,MAX\_SIZE))

#Remove padding

width = MAX\_SIZE / 7

im = im[width:width\*6,width:width\*6]

cv2.imshow("Padding", im)

#Calculate ID

ret\_val = 0

for y in range(5):

\_val1 = int(im[(y \* width) + (width / 2), width + width / 2]) #im[y, x] format

\_val2 = int(im[(y \* width) + (width / 2), 3 \* width + width / 2]) #im[y, x] format

if \_val1 == 255:

\_val1 = 1

if \_val2 == 255:

\_val2 = 1

ret\_val = ret\_val \* 2 + \_val1

ret\_val = ret\_val \* 2 + \_val2

return ret\_val

image = cv2.imread("aruco\_192.png")

ID = findArucoID(image)

cv2.waitKey(0)

cv2.destroyAllWindows()

print "Marker ID is", ID

**Detecting markers in everyday life**

#Imports

import cv2

import socket

import numpy as np

import math

#Global variables

MAX\_MARKERS = 4

mark\_detect = [0, 0, 0, 0]

objp = np.zeros((1\*4,3), np.float32)

objp[1,1] = 1

objp[2,0] = 1

objp[2,1] = 1

objp[3,0] = 1

objp = objp \* 105

#(700/9.605116)

count = 0

#Helper functions

def is\_aruco\_present(img1):

"""

\* Function Name: is\_aruco\_present

\* Input: Image captured

\* Output: Returns the set of corner points of the marker in the

image.

\* Logic: Finds the contour having four points and compares properties to that of a square

to find the marker

\* Example Call: is\_aruco\_present()

"""

i = 0

raw\_points = []

gray1 = cv2.cvtColor(img1, cv2.COLOR\_BGR2GRAY)

ret, thresh1 = cv2.threshold(gray1, 175,255, cv2.THRESH\_BINARY\_INV+ cv2.THRESH\_OTSU)

kernel = np.ones((5,5), np.uint8)

open1 = cv2.morphologyEx(thresh1, cv2.MORPH\_OPEN, kernel)

median1 = cv2.medianBlur(open1, 5)

c1, h1 = cv2.findContours(thresh1, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)

for c in c1:

k = cv2.isContourConvex(c) #Check for convexity, removes unwanted curved contours

if k == True:

i = i +1

continue

elif k == False:

e2 = 0.1\*cv2.arcLength(c, True)

a2 = cv2.approxPolyDP(c, e2, True)

print "a2 = ", a2, "\n", "e2 =", e2

if len(a2)== 4:

x = a2[0,0,1]-a2[2,0,1]

y = a2[2,0,0]-a2[0,0,0]

if x < 0:

x = x \* (-1) #Converting to unsigned int

if y < 0:

y = y \* (-1) #Converting to unsigned int

if 10 < e2 < 250: #10cm - 200cm efficiency

print "Contour Id : ",i,"length of array =", len(a2), "\n""\n", a2

print x-y

if -75 < x-y < 75:

raw\_points = a2

cv2.drawContours(img1, c1, i, (0,255,0), 2)

cv2.imshow('img',img1)

cv2.waitKey(500)

cv2.destroyAllWindows()

return raw\_points

i = i + 1

if raw\_points == []:

return '-1'

def get\_distance(x1,y1,x2,y2):

"""

\* Function Name: get\_distance

\* Input: Points

\* Output: Returns the distnce between two points

\* Logic: Uses algebra to find the distance

\* Example Call: get\_distance(x, y, x1, y1)

"""

distance = math.hypot(x2 - x1, y2 - y1)

return distance

#### To resolve the occurence of points randomly which may tilt image in Perspective ######

def refined\_points(approx):

"""

\* Function Name: refined\_points

\* Input: Points

\* Output: Corrects the randomness of points and the order

\* Logic: The distance between the aruco\_point and the point of the same type is least

\* Example Call: get\_distance(x, y, x1, y1)

"""

x1 = approx[0][0][0]

y1 = approx[0][0][1]

x2 = approx[1][0][0]

y2 = approx[1][0][1]

x3 = approx[3][0][0]

y3 = approx[3][0][1]

x4 = approx[2][0][0]

y4 = approx[2][0][1]

aruco\_points = [(x1,y1),(x2,y2),(x3,y3),(x4,y4)]

min\_dist = 10000

for i in aruco\_points:

x, y = i

dist = get\_distance(x,y,0,0)

if dist < min\_dist:

min\_dist = dist

X1, Y1 = x, y

min\_dist = 10000

for i in aruco\_points:

x,y = i

dist = get\_distance(x,y,0,480)

if dist < min\_dist:

min\_dist = dist

X2,Y2 = x,y

min\_dist = 10000

for i in aruco\_points:

x,y = i

dist = get\_distance(x,y,540,0)

if dist < min\_dist:

min\_dist = dist

X3,Y3 = x,y

min\_dist = 10000

for i in aruco\_points:

x,y = i

dist = get\_distance(x,y,540,480)

if dist < min\_dist:

min\_dist = dist

X4,Y4 = x,y

points = ((X1,Y1),(X2, Y2), (X3, Y3), (X4, Y4))

print "pts", points

return points

def Perspective(aruco\_points,img\_name):

"""

\* Function Name: Perspective

\* Input: A numpy array with four points and name of an image

\* Output: Returns the image after performing perspective transform

on it

\* Example Call: Perspective(points, "Marker.jpg")

"""

img = cv2.imread(img\_name)

a1 = aruco\_points

print aruco\_points

c1 = a1[0]

c2= a1[1]

c3 = a1[2]

c4 = a1[3]

d0 = c1[0]

d1 = c1[1]

d2 = c2[0]

d3 = c2[1]

d4 = c3[0]

d5 = c3[1]

d6 = c4[0]

d7 = c4[1]

print d0

#pts1 = np.float32([a1[0,0], a1[1,0], a1[2,0], a1[3,0]])

pts1 = np.float32([(d0,d1),(d2,d3), (d4,d5), (d6,d7)])

pts2 = np.float32([[0,0], [0,300], [300,300], [300,0]])

print pts1

print pts2

M = cv2.getPerspectiveTransform(pts1,pts2)

dst = cv2.warpPerspective(img,M,(300,300))

per\_img = dst.copy()

cv2.imshow("Perspective",dst)

cv2.waitKey(500)

cv2.destroyAllWindows()

resize = cv2.resize(per\_img, (399,399), interpolation = cv2.INTER\_CUBIC)

#Grid box

dx, dy = 57,57

# Custom (rgb) grid color

grid\_color = [0,0,255]

# Modify the image to include the grid

resize[:,::dy,:] = grid\_color

resize[::dx,:,:] = grid\_color

aruco = resize[57:342, 57:342]

cv2.imshow("grid",aruco)

cv2.waitKey(500)

cv2.destroyAllWindows()

ret, thresh= cv2.threshold(aruco, 127,255, cv2.THRESH\_BINARY)

g2 = cv2.cvtColor(thresh, cv2.COLOR\_BGR2GRAY)

return g2, pts1

def findArucoID(marker\_img):

"""

\* Function Name: findArucoID

\* Input: An image of Aruco marker.

\* Output: Returns an integer value that represents the ID of the

Aruco marker

\* Logic: The second and fourth columns from the Aruco marker are

analyzed. If the pixel value in the grid cell is 0

(black), it is taken as 0. If it is 255(white), it is

considered as 1. The binary number is generated by

reading in a top-bottom, left-right manner.

\* Example Call: findArucoID(img)

"""

height = 57

width = 57

ret\_val = 0

cv2.imshow("aruco", marker\_img)

cv2.waitKey(500)

cv2.destroyAllWindows()

for i in range(5):

for y in (1, 3):

px1 = marker\_img[height\*i + 29, y \* width + width/2]

if px1 == 255:

val = 1

#binary.append(val)

else:

val = 0

#binary.append(val)

ret\_val = 2 \* ret\_val + val

#print "Binary", binary, ret\_val

return ret\_val

def getProperties(points):

"""

\* Function Name: getProperties

\* Input: A set of four points of the corners of aruco markers.

These points can be obtained from is\_aruco\_present()

function.

\* Output: -

\* Logic: The Perspective-n-Point problem is solved by Ransac

algorithm. We obtain the translation and rotation

vectors through this function.

\* Example Call: getProperties(points)

"""

global objp

# Arrays to store object points and image points from all the images.

objpoints = objp

print "OBJP", objpoints

imgpoints = points

#imgpoints = np.array(imgpoints)

print "IMGP", imgpoints

mtx = np.load('matrix.npy')

dist = np.load('distortion.npy')

rvec, tvec, inliers = cv2.solvePnPRansac(objpoints, imgpoints, mtx, dist)

print "Rvec\n", rvec

print "\nTvec", tvec

x = tvec[0][0]

y = tvec[2][0]

dst, jacobian = cv2.Rodrigues(rvec)

print "Rot Matrix", dst

t = math.asin(-dst[0][2])

t1 = math.acos(dst[0][0])

return x, y, t, t1

def Video(True):

"""

\* Function Name: Video

\* Input: -

\* Output: Analyzes and detects markers till all markers are

detected.

\* Logic: It runs through a infinite loop checking

every frame for markers. If marker is obtained,

it increments a count, else, next frame is analyzed.

This process is continued till all the markers are

detected.

\* Example Call: Video(True)

"""

global count,s, mark\_detect

cap = cv2.VideoCapture(2)

while(True):

ret, frame = cap.read()

cv2.imshow("Video", frame)

if cv2.waitKey(1) == 32: # Ascii for spacebar

raw\_points = is\_aruco\_present(frame)

#cv2.imshow("Captured", frame)

if raw\_points != '-1':

img\_name = "Marker.jpg"

cv2.imwrite(img\_name, frame)

#count = count + 1

aruco\_points = refined\_points(raw\_points)

#p\_img, pts = Perspective(aruco\_points, img\_name)

#m\_id = findArucoID(p\_img)

#print m\_id

cap.release()

cv2.waitKey(0) # Escape sequence

cv2.destroyAllWindows()

Video(True)

s.close()

# CONCLUSION

Through the course of the project, we learnt

1) How we can detect markers from real world objects using Image Processing techniques

2) How we can decode it using simple coding methods

3) How to program in Linux

4) How Raspberry Pi is a cheap alternate for computing

5) How different image processing techniques work

# REFERENCES

* [http://opencv-python-tutroals.readthedocs.io/en/latest/py\_tutorials/py\_imgproc/py\_thresholding/py\_thresholding.html#thresholding](http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_thresholding/py_thresholding.html%23thresholding)
* <http://docs.opencv.org/3.1.0/d5/dae/tutorial_aruco_detection.html>
* <http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_calib3d/py_pose/py_pose.html?highlight=pose%20estimation>
* <http://iplimage.com/blog/create-markers-aruco/>

Evaluation:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Sl No. | Name of the Student | USN | Topic & Report  3 Marks | Presentation/Viva  2 Marks | Total  5 Marks |
| 1 |  |  |  |  |  |
| 2 |  |  |  |  |  |
| 3 |  |  |  |  |  |

Signature of the Faculty In charge