The steps used in the detection of parallelogram

1. Load the raw image and convert it into grayscale using the formula

*luminance = 0.30R + 0.59G + 0.11B*

*2. Plot the histogram to see what could be the threshold value for thresholding*

*3. Use canny edge detection to detect the edges*

*4. Threshold the image*

*5. Apply hough transform to get the accumulator and the points corresponding to*

*each accumulator array value.*

*6. Find every Two pair of parallel lines and then find two more pair of parallel lines*

*and find their intersection points.*

*7. Check if the intersection points are actually a part of the edge.*

*8. Using the intersection point found draw the line on the original image to show the*

*Parallelogram.*

*Project Structure:*

1. *Files*

* *Edgedetection.py-> This file contains functions related to edge detection and thresholding. The edge detection functions are generalised and can perform multiple edge detection depending on what parameters we pass to it.*
* *Region.py -> This file contains functions functions related to region segmentation, calculating area of regions and using a area filter. This is helpful if we want to apply area filter to the image to remove all the regions with very small area as they can be noise.*
* *HoughTransform.py -> This file contains functions to perform the houghtransform and find the points of the corners of the parallelogram.*

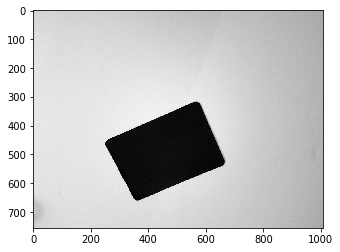
1. *Compiling and running*

* *Programming language used : Python*
* *To run the program change the name of the image in the houghtransform.py file. This change needs to be made at two lines:*

1. *image\_file\_name = "test1.raw"*
2. *rows,columns, c = cv2.imread("test1.jpg").shape*

* *The second line is just to get the size of the image. The images have to be in the same folder as the code.*
* *Finally run the python file houghtransform.py by typing the following command in the terminal( given that python and all the dependencies are already installed):* 
  + *python houghtransform.py*
* *The dependencies are the following python packages:*
  + *Bottleneck==1.2.1*
  + *numpy==1.13.3*
  + *opencv-python==3.3.0.10*
  + *scipy==1.0.0*

1. *Loading image and converting it to grayscale*

**

*Fig 1: The original color image(left) and the grayscale image(right)*

In this part we load the raw image and convert it into a grayscale image. The data format of the raw image is as follows:

The data arrangement for the interleaved raw image format is illustrated below, assuming an image of size M X N (rows X columns).

R(0,0) G(0,0) B(0,0) R(0,1) G(0,1) B(0,1) ……………………………………R(0,*N*-1) G(0,*N-1*) B(0,*N-1*)

R(1,0) G(1,0) B(1,0) R(1,1) G(1,1) B(1,1) ……………………………………R(1,*N*-1) G(1,*N*-1) B(1,*N*-1)

When we load the raw image it is thrice as wide as its grayscale image would be. To convert it to grayscale we loop over the raw image and multiply the R,G,B with their respective ration in luminance formula.

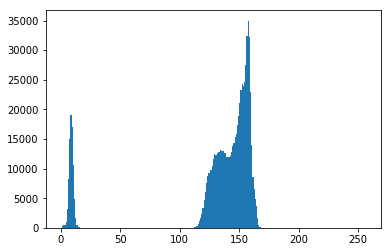
for i in range(0,rows):

for j in range(0,columns):

img\_gray\_man[i][j] = (0.3\*img\_color[i][j\*3] +0.59\*img\_color[i][j\*3+1] +0.11\*img\_color[i][j\*3+2])

1. *Plotting a histogram to get the threshold*

*The histogram of an image can be used to get an idea of what the threshold value should be. The point where we see a deep valley between two peaks in histogram can be a value of threshold. Instead of manual threshold automatic thresholding can also be used to get the threshold value but during this project I found that the automatic thresholding technique which uses peakiness to get the threshold value doesn’t perform well in all the cases. Peakiness depends on finding a valley between two peaks in histogram of an image but if the histogram is a convex shape, i.e. it has a single peak the peakiness method doesn’t perform well.*

**

***Fig 2: Histogram of image 1.***

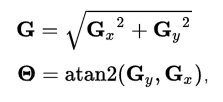
*In Fig 2 we see that the threshold value can be anything in the middle of these peaks.*

1. *canny edge detection to detect the edges*

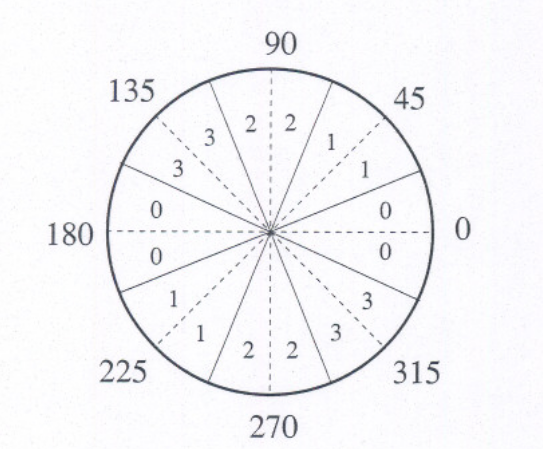
*In This part we perform the Canny edge detection to detect the edges in the image. The canny edge detection consists of 4 steps which are:*

* *Use gaussian smoothing to smooth the image. In this step we apply a gaussian filter to the image to remove noise.*
* *Apply the edge operator to the image. This will give back the image with thick edges.*
* *Quantize the gradient angle of the edges into three sectors.*
* *Apply non maxima suppression to the image with thick edges to get thin edges. This is done with the help of the gradient angles found in the previous step. We take the gradient of a pixel location and see the value of the gradient along the sector of its gradient angle. If it is less than any of the two pixels along that direction we set its value to zero.*

*The formula to calculate the gradient and the gradient angles are as follows:*



**Fig 3: Formula to calculate gradient value and gradient angle( Source:** [**Wiki**](https://en.wikipedia.org/wiki/Canny_edge_detector)**)**



**Fig 4: Value of Sectors depending on gradient angle**

def canny\_edge\_detection(image):

g = create\_guassian\_mask([3,3] , 0.0 , 1)

img\_after\_gaussian\_mask = (apply\_map(image , g , False) / sum(sum(g)))

#sobels operator returns the gradient magnitude array and the gradient direction array

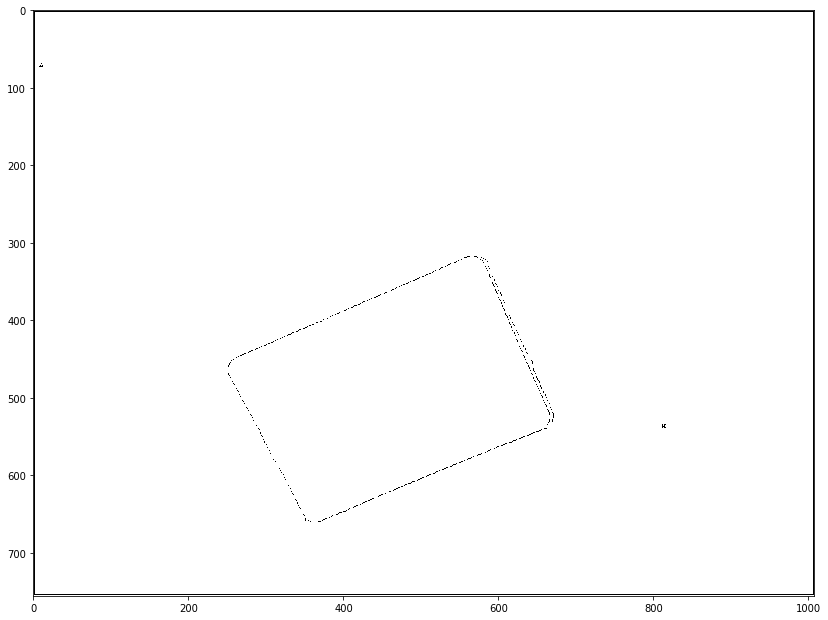
image\_after\_sobels\_operator , theta = edge\_operator(img\_after\_gaussian\_mask , "sobels")

quantized\_theta = quantize\_gradient\_angle\_to\_secotrs(theta)

image\_after\_non\_maxima\_suppression = non\_maxima\_suppression(image\_after\_sobels\_operator , quantized\_theta)

return image\_after\_non\_maxima\_suppression

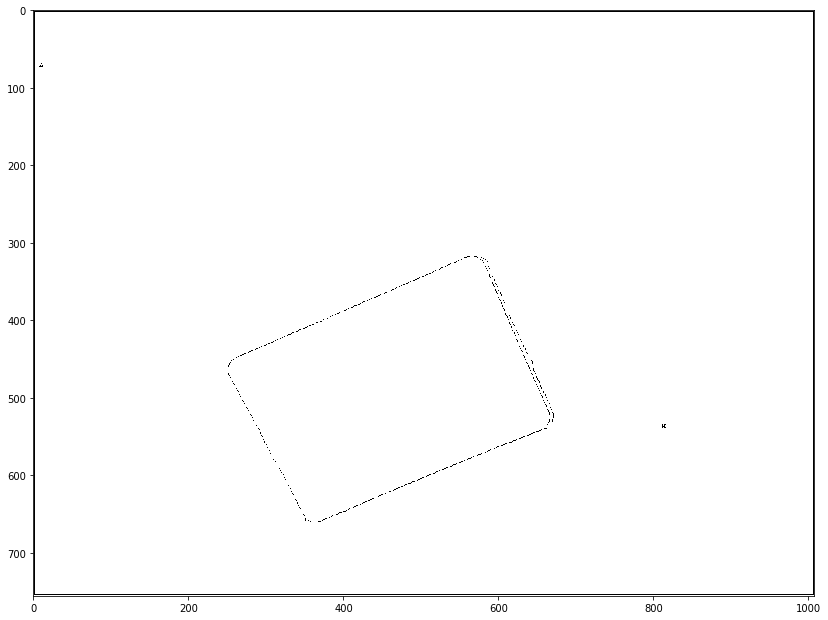
This function is in the edgedetction.py file and all the functions it calls are also in the same file. This files get called by the houghtransform.py file. It first creates a gaussian mask ‘g’ which is then applied to the image to smooth it and remove noise. Then we use the Sobels edge operator to get the gradient magnitudes and gradient angles. These gradient angles are then quantized by the function quantize\_gradient\_angle\_to\_secotrs.



**Fig 5: Image after canny edge detection**

1. *Threshold Image*

*In this part we threshold the image. We use the threshold value we found above to convert the grayscale image to binary image. This is based on simple comparison, if the value of the pixel is above threshold set it to 255 and if it is below set it to 0( this will be reversed depending on if you want the background to be black and foreground to be white or vice versa). After you threshold the image you will get a binary image on which we can perform the hough transform.*

******

***Fig 6: Image after thresholding***

1. *Apply hough transform*

*In this step we apply the* [*hough transform*](https://en.wikipedia.org/wiki/Hough_transform) *to get the accumulator and the positions of the pixels that correspond to incrementing a particular location in the accumulator.*

*The* ***Hough transform*** *is a* [*feature extraction*](https://en.wikipedia.org/wiki/Feature_extraction) *technique used in* [*image analysis*](https://en.wikipedia.org/wiki/Image_analysis)*,* [*computer vision*](https://en.wikipedia.org/wiki/Computer_vision)*, and* [*digital image processing*](https://en.wikipedia.org/wiki/Digital_image_processing)*.*[*[1]*](https://en.wikipedia.org/wiki/Hough_transform#cite_note-1) *The purpose of the technique is to find imperfect instances of objects within a certain class of shapes by a voting procedure. This voting procedure is carried out in a* [*parameter space*](https://en.wikipedia.org/wiki/Parameter_space)*, from which object candidates are obtained as local maxima in a so-called accumulator space that is explicitly constructed by the algorithm for computing the Hough transform.*

*The simplest case of Hough transform is detecting straight lines. In general, the straight line y = mx + b can be represented as a point (b, m) in the parameter space. However, vertical lines pose a problem. They would give rise to unbounded values of the slope parameter m.*

**

*Fig 7: Formula for the line in theta, r*

**

*Fig 8*

*Hough Transform Algorithm (for straight lines)*

1. *Quantize the parameter space appropriately*
2. *Assume that each cell in the parameter space is an accumulator. Initialize all cells to zero*
3. *For each point (x,y) in the image space increment by 1 each of the accumulators that satisfy the equation*
4. *Maxima in the accumulator array correspond to the parameters of model instances*

*Hough transform implementation:*

1. *First I create the theta array depending on the step size using the following code:*

*def get\_theta\_list(step\_size):*

*if 360 % step\_size != 0:*

*print("step\_size not a factor of 180")*

*no\_of\_steps = int(round(360/step\_size))*

*return np.linspace(0 , 360 ,no\_of\_steps )+(round(step\_size/2))*

1. *Then for all the points in the binary image that are foreground( 0 or 1 depending on how the foreground is represented) we increment the value of the accumulator array for all the values of theta in theta array that we got in previous step. We also store the location of the points that incremented a particular position in accumulator array in the dictionary accumulator\_positions The key of this dictionary is the p or r and the theta and the value is the list of points. The code to create the accumulator array is:*

*def increment\_accumulator(image , thetas , theta\_step\_size = step\_size\_theta, p\_step\_size = step\_size\_p):*

*accumulator\_positions = {}*

*n,m = image.shape*

*accumulator = create\_accumulator(len(thetas) , p\_step\_size)*

*accumulator\_rows , accumulator\_columns = accumulator.shape*

*# this line get the index of all the points that are non zero in the binary image*

*y\_idxs, x\_idxs = np.nonzero(image)*

*for z in range(len(y\_idxs)):*

*i = y\_idxs[z]*

*j = x\_idxs[z]*

*for theta in thetas:*

*theta\_position\_in\_accumulator = int(round((theta)/step\_size\_theta))-1*

*result = int(round((calculatep(theta , j , i)+diag\_len)/step\_size\_p))*

*accumulator[result][theta\_position\_in\_accumulator] += 1*

*#check if the point is already at this loc in dictionary*

*if((result,theta\_position\_in\_accumulator) in accumulator\_positions.keys()):*

*accumulator\_positions[result,theta\_position\_in\_accumulator].append((i,j))*

*else:*

*accumulator\_positions[result,theta\_position\_in\_accumulator] = [(i,j)]*

*return accumulator , accumulator\_positions*

1. *Once we get the accumulator array we threshold it to remove all the points that may not correspond to a straight line.*
2. *After thresholding we find the points of intersection of parallel lines to get the corners of the parallelogram. To do this first we find two pair of parallel lines from the accumulator\_pos dictionary and check if these parallel lines are some distance apart to prevent detecting very narrow parallelograms which may be of the same edge. Then we find two more parallel lines using the same dictionary and check if they are some distance apart. Now we have 4 lines which may form a parallelogram.*
3. *To check if the 4 lines actually form a parallelogram we use the equation of these four lines and find the four intersection points. Then we check if there are enough points in between these corner points to say if they actually for a parallelogram. To check this we can find the distance between the corner points and then check if the ratio of the distance and the number points between these corner point is above some threshold.*
4. *Once we have the corner points of the parallelogram we can draw it over the original image using the cv2.line function.*

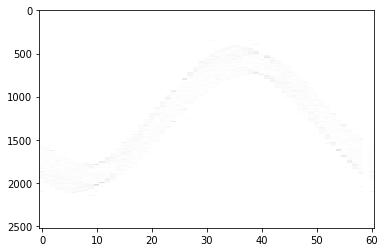


Fig 6: Image if the accumulator array(X-axis: theta , Y-axis:p)

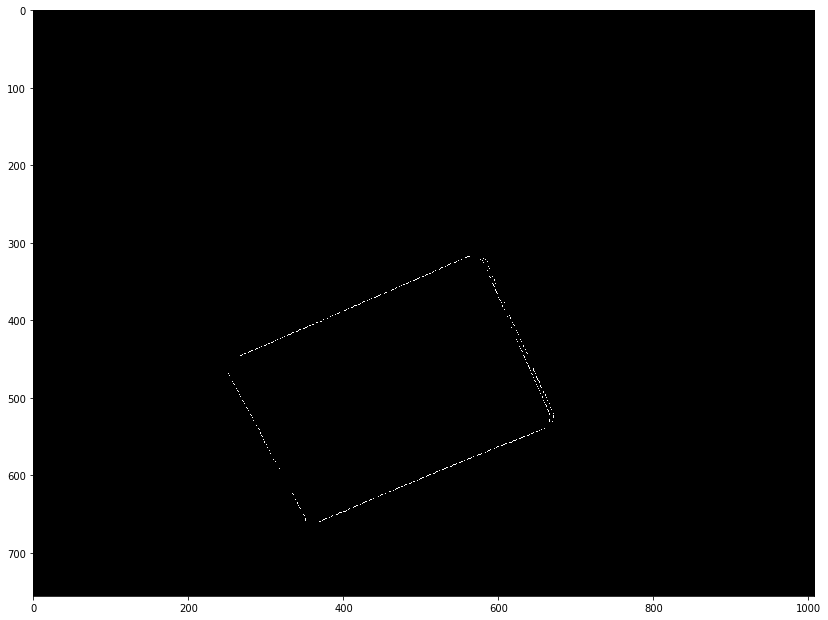
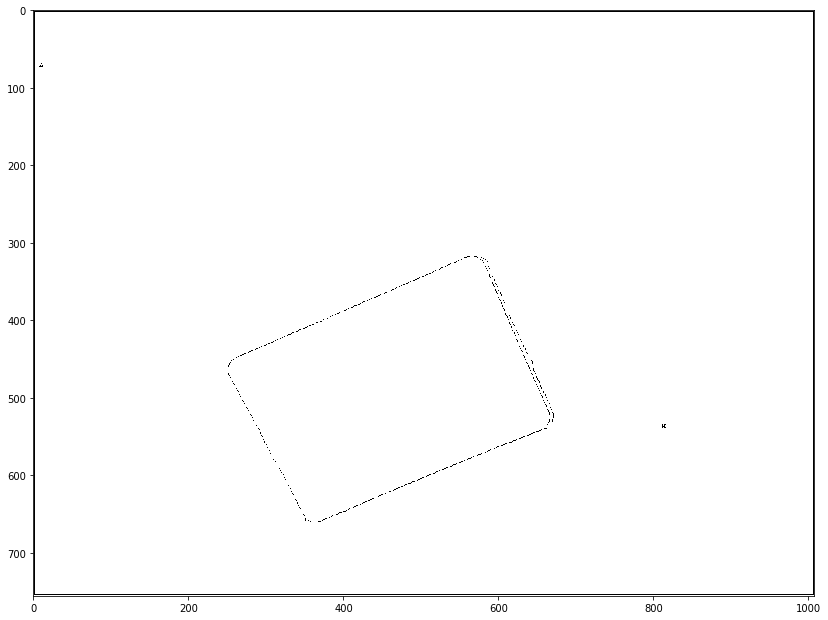
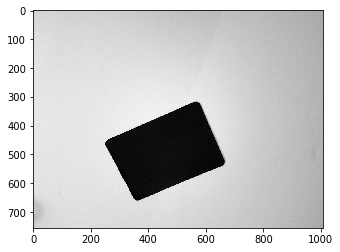


Fig 7: Image after thresholding of accumulator array

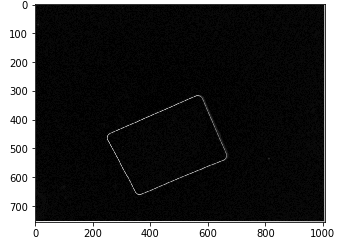
Image 1 :



Original Image



Grayscale Image Binary Image



Gradient image

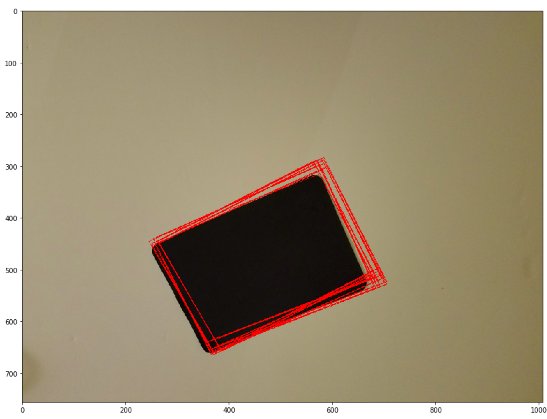


Image after drawing the parallelogram over it

Corner points:

[[(541, 659), (311, 570), (458, 239), (662, 343)], [(537, 669), (312, 569), (458, 239), (662, 343)], [(538, 667), (312, 567), (458, 239), (662, 343)], [(532, 681), (306, 581), (458, 239), (662, 343)], [(539, 665), (314, 564), (458, 239), (662, 343)], [(662, 343), (458, 239), (307, 580), (533, 680)], [(662, 343), (458, 239), (309, 575), (530, 687)], [(662, 343), (458, 239), (314, 563), (539, 664)], [(662, 343), (458, 239), (304, 585), (530, 686)], [(662, 343), (458, 239), (307, 579), (528, 691)], [(662, 343), (458, 239), (309, 574), (539, 662)], [(662, 343), (458, 239), (305, 584), (531, 685)], [(662, 343), (458, 239), (312, 568), (537, 668)]]

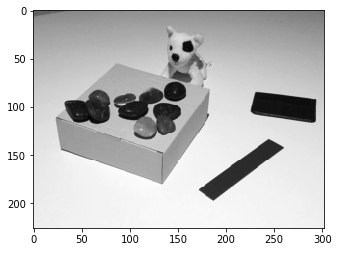
Threshold value : 40

Threshold of accumulator : 0.2\*max(accumulator)

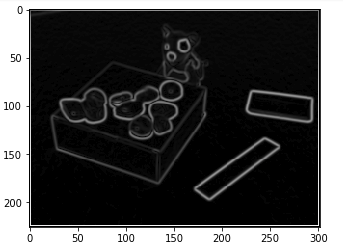
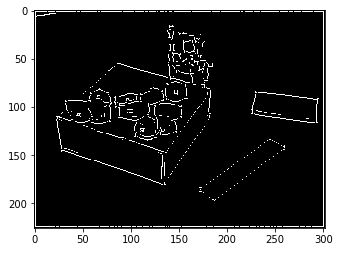
step\_size\_p = 1

step\_size\_theta = 3

Image2:



Original image grayscale image

 binary image Gradient Image

Threshold value : 40

Threshold of accumulator : 0.5\*max(accumulator)

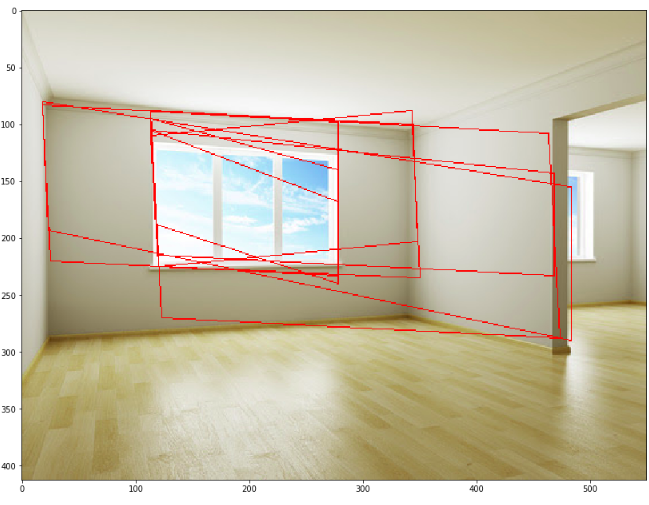
step\_size\_p = 2

step\_size\_theta = 3

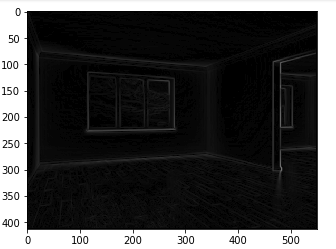
Image3:



Original Image Grayscale Image



Binary image Parallelograms found

Gradient Image

Threshold value : 40

Threshold of accumulator : 0.37\*max(accumulator)

step\_size\_p = 2

step\_size\_theta = 3

Corner points: [[(115, 189), (108, 49), (35, 45), (43, 185)], [(33, 7), (39, 111), (93, 111), (88, 10)], [(35, 45), (40, 137), (94, 140), (90, 48)], [(35, 45), (39, 111), (93, 111), (90, 48)], [(38, 45), (56, 111), (96, 111), (75, 47)], [(42, 45), (57, 187), (93, 187), (86, 48)], [(44, 45), (35, 137), (81, 139), (90, 48)], [(75, 47), (96, 111), (67, 111), (42, 45)], [(77, 9), (116, 193), (62, 193), (32, 7)]]