# Detecting the Spatial Layout of Indoor Scenes

**Final Report of**
**Computer Vison Project**

**by**

Haoyuan Zheng (hz463)
Minze Mo (mm2943)
Runqian Huang (rh627)
Ruoxuan Xu (rx65)

## Abstract

Our project aims to analysis the spatial layout of indoor scenes. To be concretely, we will develop a suitable algorithm to detect the indoor structure automatically, including the intersection points and lines between the ceiling, walls and floor. Such topology information can provide service to many areas, such as indoor robots and so on. The main steps include pre-processing (noise removal via LPF, suitable thresholding method selection (adaptive thresholding), morphological filter (opening, dilation and erosion), edge detection(Sobel operator, Canny operator)), Hough transform( straight lines detection), post-processing(desired intersection lines and intersection points selection),experiment(accuracy measurement and comparison with baseline). The result is that our prototype can detect the desired intersection points and lines and the room's layout can be estimated accurately. Some challenges we meet include shadow interference, a comprehensive post-processing to handle various layout.

## Introduction

### 1.1background

Consider the purpose of our project: automatic layout detection. For a regular room, the layout is composed of intersection lines between ceiling and wall, floor and wall. The vertices are the intersection points of three intersection lines. To simplify the problem, we temporally only take an empty room without any clusters into accounts. It is natural to think that we can find out all straight lines in the image and select the desired lines from them based on some standards. Finally, we connect all desired lines together and acquire the desired layout.
In order to figure out its layout accurately, Hough transform is applied to detect the straight lines in the image. Straight line Hough transform takes edge information as input. Thus before applying Hough transform, pre-processing is necessary in order to preserve as much as important information in the image. Because Hough transform is a common algorithm in computer vision, there is Hough transform function in openCV. In this way, it is easy to detect the straight lines in the result of edge detection. After Hough transform, a post-processing is needed to remove the undesired lines and make the result as accurate as possible. Lines selection in post-processing can base on some geometric properties, like angle relationship and distance relationship.

There are different problems should be addressed in every step.

In the pre-processing step, thresholding selection is significant. Under normal indoor environment, even the light sources distribute uniformly, it is avoidable to generate shadows in the intersection areas of walls and ceiling. The existence of shadows makes a huge negative difference on the final result. Because the pixels in the shadow areas have strongly different pixel values with those of walls. During the thresholding process, these undesired pixels are still be preserved as foreground pixels. In the edge detection step, the edge operator will think the boundary of the shadow area as an edge rather than the intersection lines between ceiling and wall. If we don't remove the shadows in the very beginning, it will leave a cascading effect on

the all following steps, including edge detection, Hough transform and post-processing and so on. One of the main issue in pre-processing step is to remove the shadow and make the edge detection as clear as possible.
Solution can be selecting suitable thresholding method. The sources of generating shadows are the variance of illumination. Adaptive thresholding can handle such problem. It divides the image into several local areas and thresholds each region with different threshold. In this way, it can effectively solve the problem caused by illumination variance.

In post-processing step, we should remove the undesired lines. Because after Hough transform, it will detect many straight lines it can but we only want several of them. If we emit this step, lots of lines generate lots of intersection points, it is impossible to distinguish which lines are what we need.
For a bunch of lines, if they pass a same point or the distance between them is really small, we combine them together to generate a single line; For some discontinuous lines, we measure the appearance of background pixels. If background pixels appear frequently along this line, we think that it is a discontinuous line and it must not be the intersection line we want. In these ways, we can significantly reduce the amount of undesired lines.

## 1.2 Previous Work

We research different papers and find out several reasonable methods to achieve our goal.
In the first one, it firstly applies Hough transform to detect the straight lines in the room, then these straight lines are extended to find out the vanishing points, which some intersection points of walls and ground are hidden by the furniture(1).
Besides, another paper illustrates a method based on cascaded Hough transform to detect vanishing points. This method emphasizes on determining the direction of each line and find out the intersection point of three-orthogonal-direction lines(2) In this way, we can acquire several possible layouts of this room by connecting these points together. Then it applies edge-based image features to analyze possible layouts and find the most possible one. Another paper supposes a method to detect the vanishing point directly form the result of Hough transform. We can do a calculation in polar plane to find out the vanishing points. In this way, less computation is needed because we take the information from last step directly.

We apply Manhattan distance to our evaluation function. We are inspired by several paper which illustrates an effective edge-based methods based on Manhattan distance(3)
Manhattan distance is the distance between two points in a grid based on a strictly horizontal and/or vertical path. It is easy to calculate so that it is suitable for implementing our evaluation function. In this paper, it points out that It is often argued that it is best to base estimation on all pixels in the image(4) By extending this point, we take all the coordinates of characteristic points, both in ground truth and program output, into considerations in order to develop a feasible and reliable evaluation function to measure the performance of the prototype(5)

In order to extract desired information from the lines acquired through Hough transform, we also explore how to post-processing these straight lines. A paper focus on graph cut (6)

. It applies order-preserving labeling to graph-cut segmentation. It is a novel to segment the structure of a room in a graphic way. Another paper firstly estimates the room edge maps. use thick lines to form the maps, and apply Gaussian blur to further smooth the boundaries of edges and non-edge regions. Then there is a network to take the room image as input and the room edge map as output(7) The first part of the network is feature matching with several convolutional layers and the second part is map generation with several deconvolution layers. We can gain concrete steps via these two papers. Finding all possible vanishing points and generating the candidate layouts are obvious. We can apply Hough transform to acquire straight lines and feature extraction to generate possible layouts. Finally, we should design a set of criteria to rank these candidates and the one with highest marks as the result.

## 1.3 Proof of concept example

Before developing the comprehensive approach, we try on a simple approach to prove the feasibility of our design. In the first step, a Canny edge detector is applied to pre-processing the origin image. Then we apply Hough transform and probability Hough transform to detect the straight line. Probability Hough transform is different from Hough transform. It randomly takes twenty percent of points to do Hough transform then remove these points. It also combines the lines which are close to each other. In the result shown below, even though the results are not accurate, it also proves that our designed approach is feasible and still many refinements can be made.

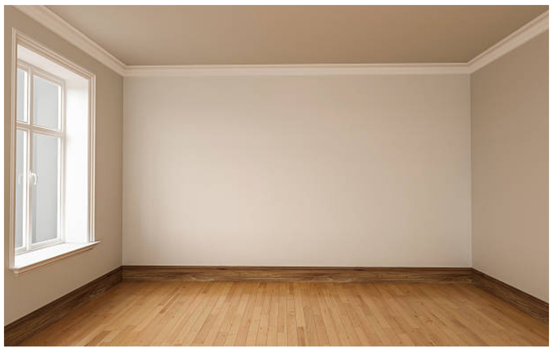


Figure 1 Original Image

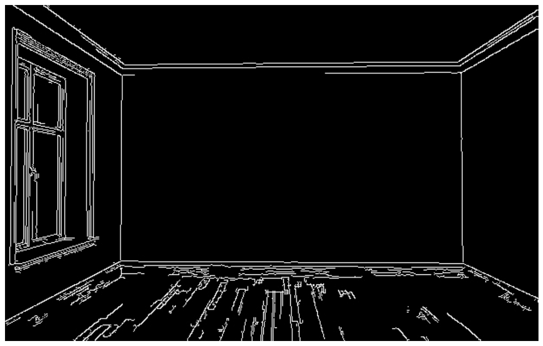


Figure 2 Canny Edge Detection

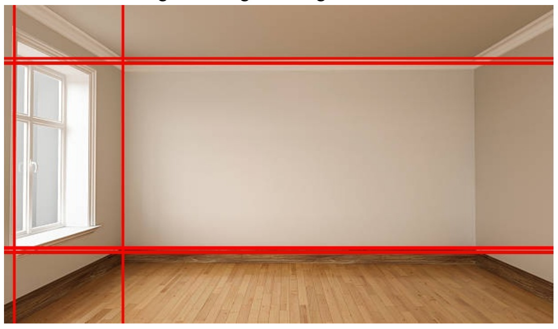


Figure 3 Original Hough Transform

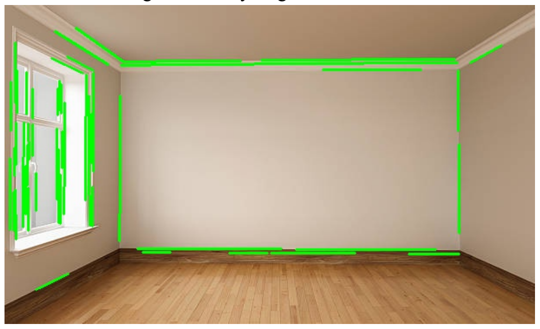


Figure4 Probability Hough Transform

## 1.4 Overview of algorithm

We choose the algorithm of Hough Transform as the foundation and develope our own methods based on the output of it. Our main algorithm can be set into 4 portions, which contains preprocessing, Hough transform, lines filtering and finally feature detection. The outcome of our algorithms for each input image is the coordinates of the key points from which the layout lines of the space are determined.

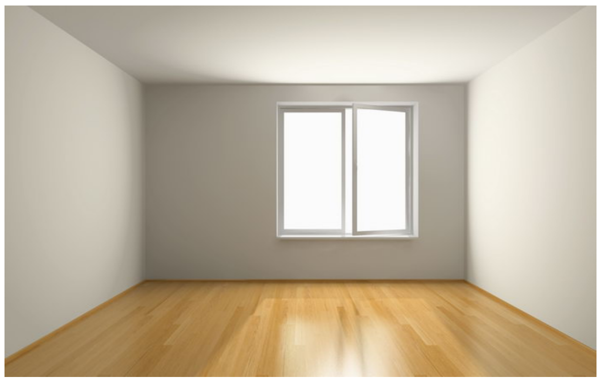
The dataset we use includes 8 images of intercrossing walls with no clutter and 2 images where some obstruction to the walls are added. The source of images is *Google Image.* With the key points we have marked manually, we compute the distances between the man-marked key points and the key points generated from our algorithms with a designed formula. As a result, the error rate of this picture is drawn and accordingly we get the evaluation outcome.

The programming environment applied here is Pycharm, by which we create python programs with opencv2 library. The reason why we missed such a chance to explore VisionX is that we constantly met with segmentation faults in performing the Hough Transform Functions on 2-channel images with VisionX. Indicated by the instructor's answer on Piazza, we realized the Hough Transform on VisionX may have some restrictions on image size which are hard for us to ensure, we had to swift to another software in this project.

## Algorithm

### 2.1 Preprocessing.

An robust preprocessing algorithm have been developed as the first step. As can be seen in the baseline, Canny edge filter miss lots of important lines even in simple image due to illumination. The main problem for wall images is that the light condition varies much due to the reflection of white surfaces. Around the intersection points of walls and floors, there are some shadows which seriously reduces the performance of following edge detection. Without the removal of shadows, program detects the boundaries of the shadows as the edge rather than the intersect lines. Therefore, adaptive thresholding is employed to remove the quantization noise to get a binary image. Besides, the concrete steps of preprocessing are listed as follows:


The original image

1. Convert the RGB image to grey scale image and blur it via Gaussian filter
2. Apply adaptive thresholding to threshold the grey scale image. It partitions image into local regions, determines a threshold for each region and interpolates a threshold for each pixel. The most significant point is that adaptive thresholding is suitable for dealing images with variant illumination.
3. Adaptive thresholding will inverse foreground and background. The pixels we need are labeled as background in the processed image. The purpose of this step is to switch foreground and background so that the boundaries we want become foreground pixels.
Compared with binary thresholding, it is obviously to see the removal of the shadows in the upper-left and upper-right corners. Besides, more details of the right intersection line will be preserved via adaptive thresholding.
4. Apply opening to remove the scattered pixels. The kernel is a circle with radius of 2.

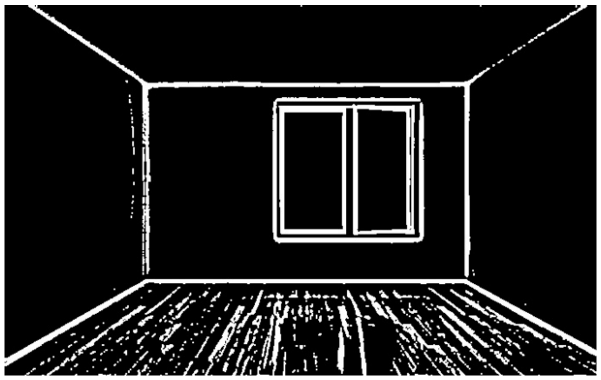
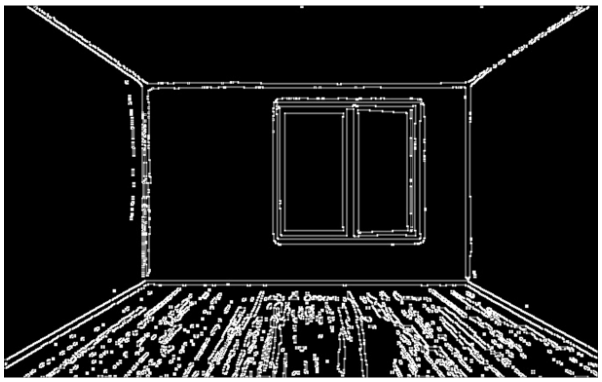


Image after adaptive thresholding

5. Apply Sobel edge operator to do the first edge detection. This step is the preparatory work for the removal of the redundant pixels on the floor.


The sobel edge

6. Apply dilation and erosion to remove the undesired pixels on the floor.

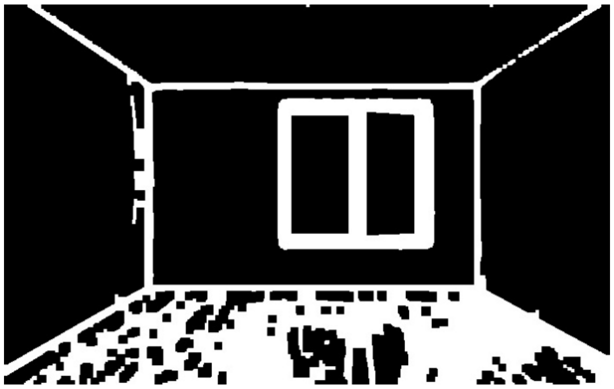
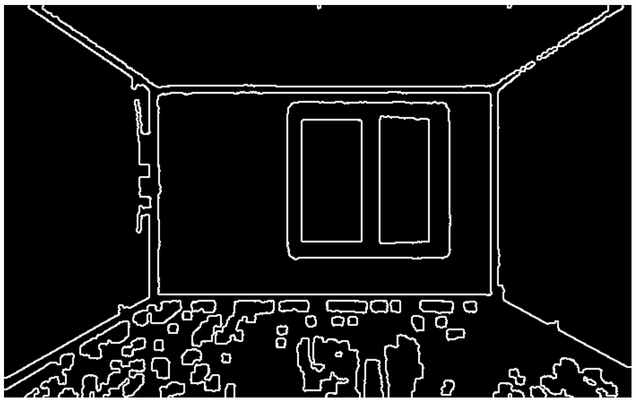


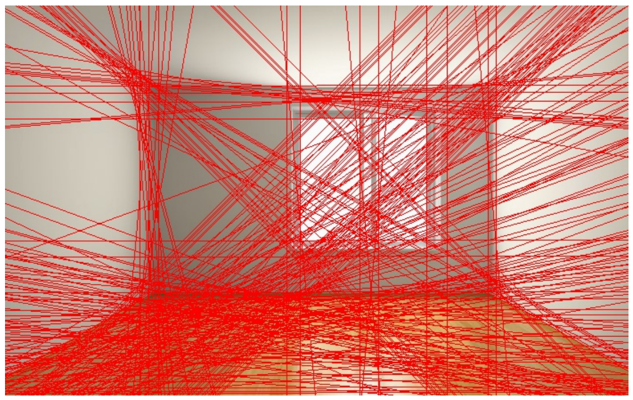
Image after morphological process

7. Apply Canny edge operator.


Final edge image

### 2.2 Hough transform

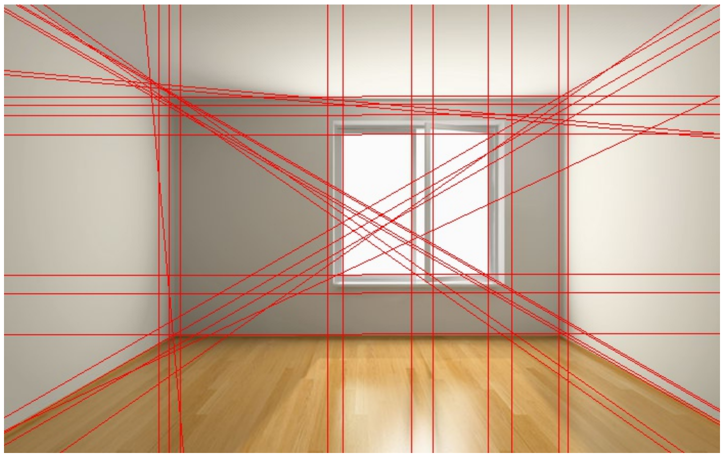
In this part, we use Hough transform with low threshold to retain more information of straight line. In baseline approach, the straight line is based on the strength of counting number of threshold. This criterion is not suitable for layout detection case. Therefore, in this step, we choose to retain as much as as possible and leave the criterion of choosing straight line to the next step.In more details,we choose to set the sampling interval to be the smallest, which is 1°, and the threshold of weight in Hough space to be low, which is 30.



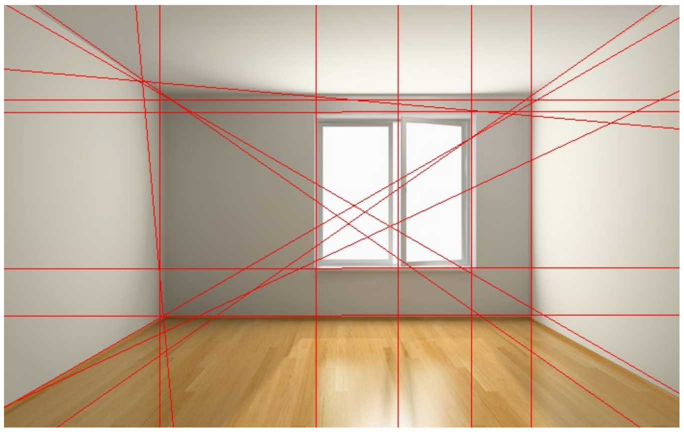
Result of Straight Line Hough Transform

## 2.3 Lines filtering

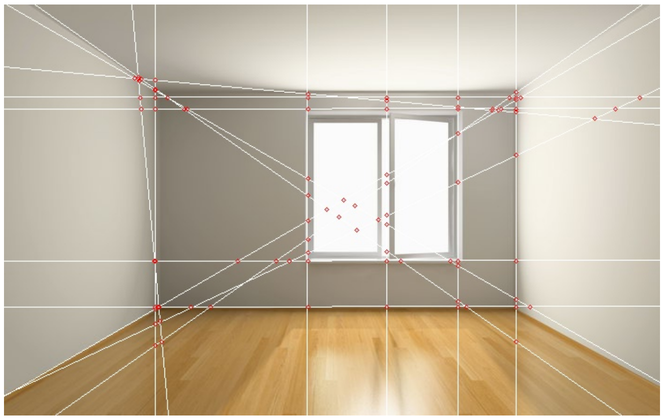
In this part, we filter the unqualified lines out and find the intersection points of remaining lines for the future detection of walls' layout. There are two parts of line filtering: similar line merge and discontinuity reduction.
The first step is to implement a 'sliding kernel' to detect the continuousness of the lines detected by Hough transform. As in Hough space, parameters voted over a relative lower threshold are all taken in, we should rule out the 'lines' not successive in the previous image to deal with it. To be more specific, for every line detected by Hough Transform, we search through the corresponding neighborhood (defined by the kernel sliding along the line) on the binary image once serving as the input of Hough function. If a corresponding point with the gray level of 255 is got right after the former one with 0, which means this one is the beginning of a visible part of the line, we start a counter and increment one for every 255 - gray value pixel coming in a row. We would switch to another counter as soon as the next beginning of a visible line segment is found. After the iteration of the whole line, we check if there is a counter exceeding a certain value, and when it does, we reserve that line from line list.


Lines after 'sliding kernel'

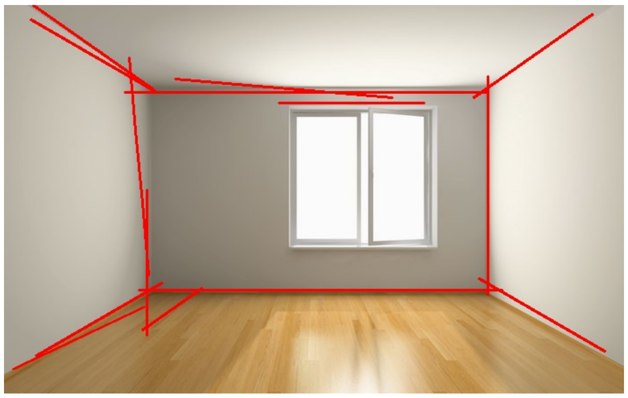
After the 'sliding kernel', we take the second step of substituting one pair of parallel lines which are very close to each other with a newly made one with the same slope, lying in the middle of them. The problem we try to solve here is that around the wall edges there are more shadow than a clear line after the preprocessing of the image, so after Hough transform the edges of the shadow, instead of the wall edges itself, are detected. Therefore, we just take all lines returned from Hough transforms into groups by the value of and check the differences of the value of among the lines in one group. When the difference between a pair is lower than a threshold we set, it is eliminated from the line list and a new one will be added into it.


Lines after merging parallel lines

There are post line filtering work following. We need to find the intersection points of the filtered lines so that we can utilize the feature of them to extract the wall edges. Here we are not trying to find all the intersections for lines, but those likely be formed by the wall edges. We realize it by classifying the lines into 4 groups according to their values, and only computing the intersection points of lines from different groups. Until now we get all the filtered lines in one list as well as all the intersection lines in another list.


Result of finding intersection points

And as we can see in the results, the lines are all passing through the whole image, that is not what we want. The lines should fit the exact layout. Here we use a kernel to search along each of the lines. If the kernel continuously finds edges within it, and the length is larger than our threshold, then we think that this line segment should be part of the actual edges. If the kernel continuously finds that no edge is within it, and the length is larger than our threshold, then it means that the line segment should end here and we have to start to find a new line segment. Finally, we can get many line segments that fit the original layout.


Final line segments

### 2.4 feature detection

We have concluded some features shared by most of the indoor wall images, and they are listed as follows:

1. the corner lines between 2 walls, without ceilings or floors involved, are usually vertical or almost vertical in the image
2. Among all the vertical lines in the image, only real line corners have two lines in different slopes varying much interacted with itself on nearly the same point.

Based on such 2 features we deducted from the wall images, we set the program first to sort out the vertical lines in the image and to detect all the points it intersected with other lines in a largely variant direction. Also, we compute the intersection distance and find if there are 3 points are within a small distance from each other. If so, this vertical line is taken as the corner lines formed by wall intersections. Thus we determined the corner lines and the points it has with the ceiling or the floor. Then we compute the intersection of the ceiling edges with the border of image to get the feature point of this edge.

## Experiment

### 3.1 Hypothesis and evaluation function

We design a simple but effective evaluation function to measure the performance of our prototype. We manually pick up the coordinates of all characteristic points (vertices of the room) as the ground truth. We also record the coordinates of these characteristic points estimated by program. For a single point, we calculate the Manhattan distance between the ground truth and the program output. Distance value is divided by the number of characteristic points. Then this temporary result is divided by the sum of the width and height of this image. The result is considered as the error rate. We measure the performance of our prototype in a quantized way. It is obvious that the lower the error rate is, the greater accuracy of detection is. The formula is listed below, where ε denotes the error rate.

$$\varepsilon = \frac{\frac{1}{n}\,\Sigma\,|p_i(x,y) - p'_i(x,y)|}{img.width + img.length}$$

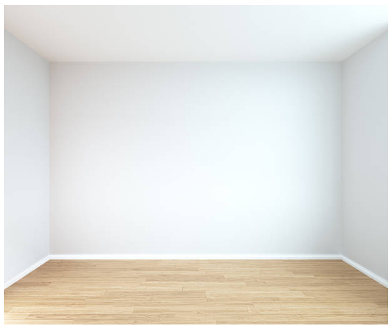
The reason why the Manhattan distance is divided by the number of characteristic point and sum of image length and width is that we should eliminate the variance of the image sizes given the truth that the larger the image is, the same distance of pixels counts less. As the order of the distance is 1, we choose the length value of the image shape rather than the area value to calculate.

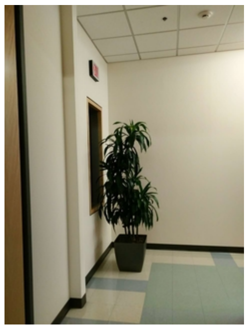
For cases where we cannot detect the same number of points as the ground truth, we still count the distance between those key points of two group, only for the smaller number of points no matter it is from the program generated points or by human vision. As we have sorted the two lists of points, generated by program or recognized by Human visual system, we are able to compute the ε through points in these lists until one list reaches the end. Then the ε is multiplied by the larger number of points ratio the smaller number, such serving as a method to represent the increasing error for the points the program misses.

Our hypothesis for the evaluation result is to lower the error rate down to 20%. Considering our performance feature detection methods rely much on the situations of the images, the error rate lower than 40% is a challenging goal for us.

### 3.2 Documented Data Set

Our data set is partly from google image and partly taken by ourselves, and the amount of the images is 10. In the data set most images are indoor scene with wall intersection and light reflection. 4 of them are only door with no obstructions or interference like clutters or windows, while 4 of them contain some doors or windows inside. Both these two kinds are taken from different perspective and light reflections. There are 2 more images that are cluttered with plants and in a darker condition. example are listed as figures below.





We mark the vertices of walls and the intersecting lines for wall edges with the borders of image so that the layout of the space can be specifically determined. The coordinates of the marked points are taken as ground truth.

### 3.3 Experiment procedure

We read the images in the dataset and go through the four parts describes in the program.
In part 1, the main parameters are the thresholding parameters for adaptive thresholding, and the kernel size for the morphological processing, based on observation of the outcome of that process of image.

The second part we determined the parameters for Hough transform.

In the part of three we refine the algorithms of lines filter, change the slide kernel size and the intersection methods.

In part four we have tried several ways to detect the key points, including vertices detecting function in Opencv library, the color-spliting function to recognize part of the key points and so on, to detect the feature points as many as possible. Based on the key points we detected and evaluated, we finally determined to use the algorithm of vertical lines searching to fulfill the goal in this stage.

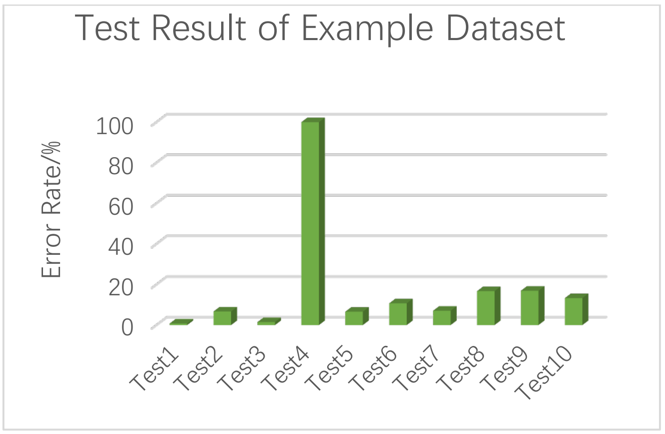
## Result

The result of our experiment taken on 10 tests are presented below. The error rate ε is presented as the first row in the table, and the number of feature detected by our algorithm and defined by human are showed in the following tuples. Except for the image in test 4, we can find the feature points can almost be detected by the program, and the error rate met our hypothesis.

| *ImageNo.* | *Test1* | *Test2* | *Test3* | *Test4* | *Test5* | *Test6* | *Test7* | *Test8* | *Test9* | *Test10* |
|---|---|---|---|---|---|---|---|---|---|---|
| *Error Rate(%)* | 0.769 | 6.696 | 1.474 | 100 | 6.589 | 10.700 | 7.067 | 16.710 | 16.924 | 13.300 |
| *Number of Feature* | 8 | 8 | 6 | 6 | 6 | 8 | 6 | 8 | 8 | 6 |
| *Detected Feature* | 8 | 8 | 6 | 0 | 6 | 8 | 6 | 8 | 8 | 6 |

Table: result of test case

The test result is analyzed with this diagram, and the mean error rate computed from all the tests are 16.352 %, which succeeds in being lower than the assumed error rate we make in the hypothesis part.



result of test case

## Discussion

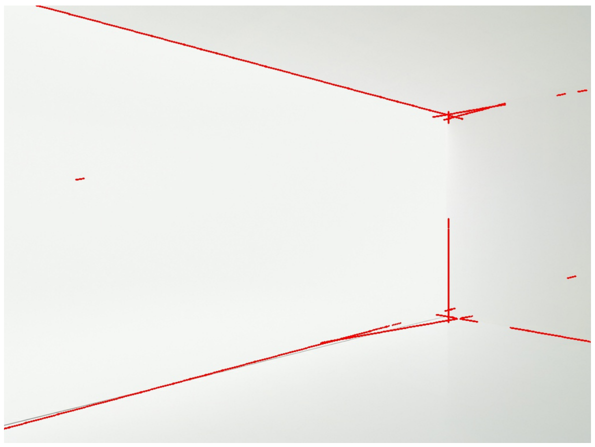
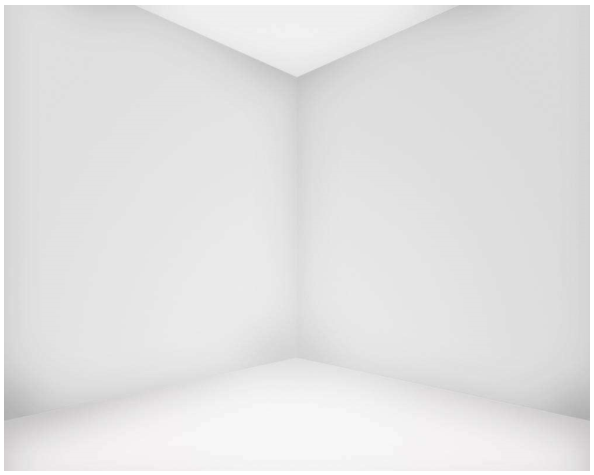
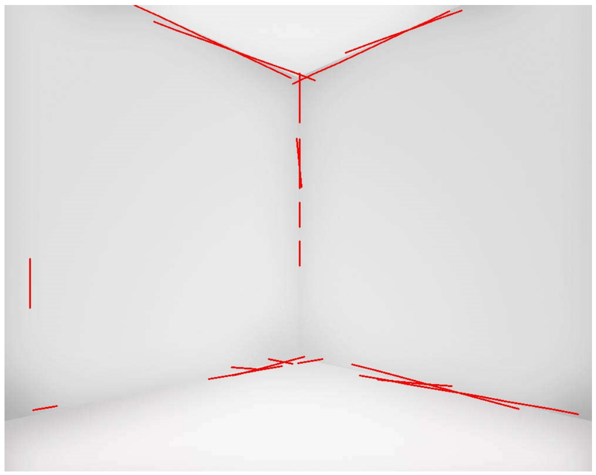
As can be seen in the result, our strategy have gained quite promising result on the spatial layout detection. The overall error rate of our algorithm is 16.8%. As can be observed in the graph, in most situations the detection rate is under 10%. In simple and clean environment, our algorithm works quite well on the layout. In the best result, the error rate is as low as 0.76% as following.

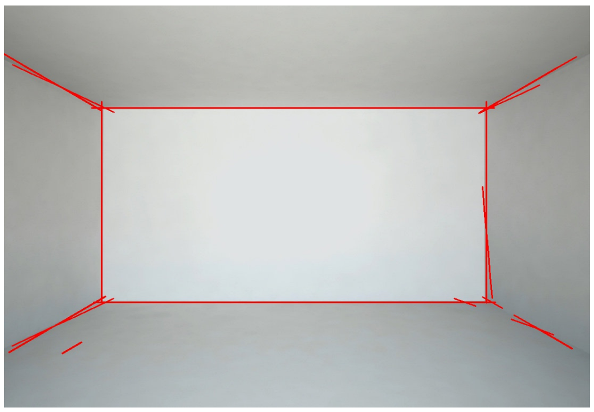
In most test situation, our algorithm have succeed in finding the right number of major point, with a slightly error distance from human visual system. However, in some example, our result fails on finding any topology features and return no answer, which significantly increase our overall error rate. This is due to the failure in identifying all straight line in preprocessing steps.
Some of object with straight line in image could affect the result of our detection, even it is not cover the line of wall, it increase the complexity of our algorithm and increase the error rate typically.

In the following we will show the results when we apply our algorithm in different conditions and environments.

1. Pure white room

As for the pure white room which the layout is complete, our algorithm can find the layout well. However, we can see in the results that some of the lines are incomplete, that is because the when the walls have similar color and similar shadow, the boundary between them just becomes invisible even for humans. So the edge detector could not detect the edge. If the length of this missing edge is within our threshold, then our algorithm could still draw the complete layout line, however if it is too long, then our line will be incomplete too. And we can also notice that there are some line segments that are not the layout line, that is because of the shade or the changes of the white color, we can’t remove the effect of these factors completely, therefore, after edge detection, the shade can also generate some edges, which will generate some redundant lines. In general, our algorithm can handle this kind of condition well, the images of the walls can be taken from any angles, and our algorithm could still be able to find their layout.

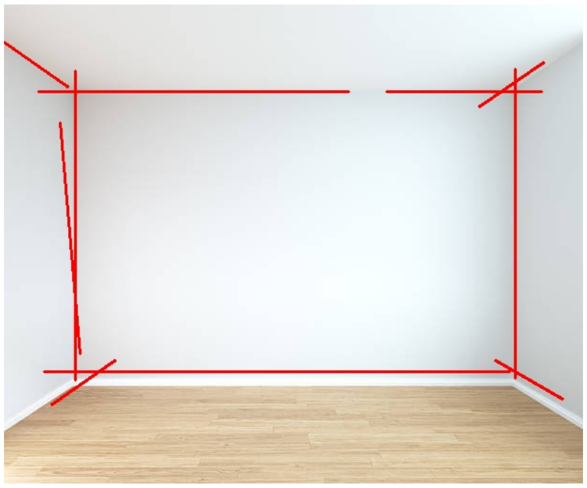
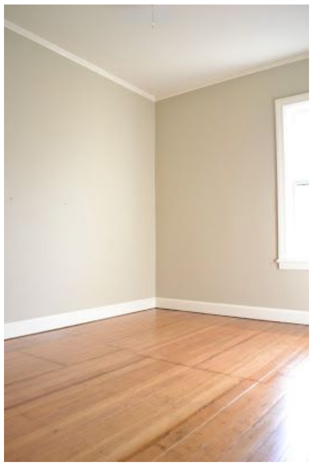
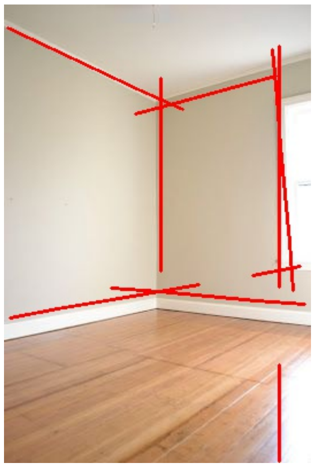






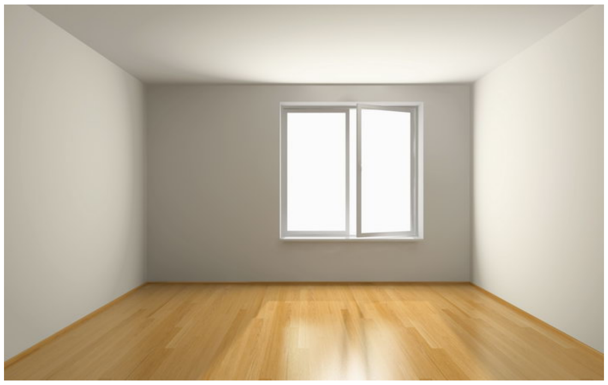
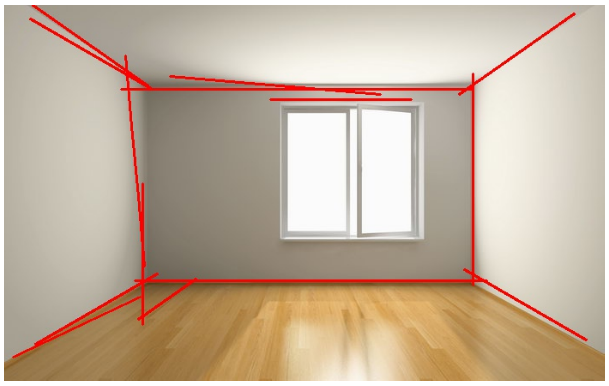
2. Pure white walls with color ground

As for the room with pure white walls and color ground, this will be more complex than the previous condition. The color ground always have texture, this would bring noises to the image.

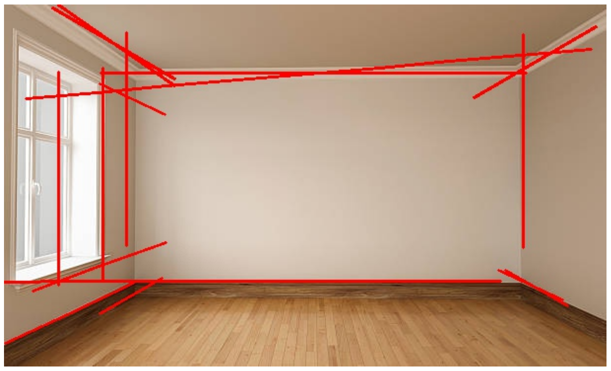
As texture itself would have some edges, therefore, Hough transform would generate straight lines for them too. However, as we can see from the results, our algorithm get rid of most of the effects of the ground texture.

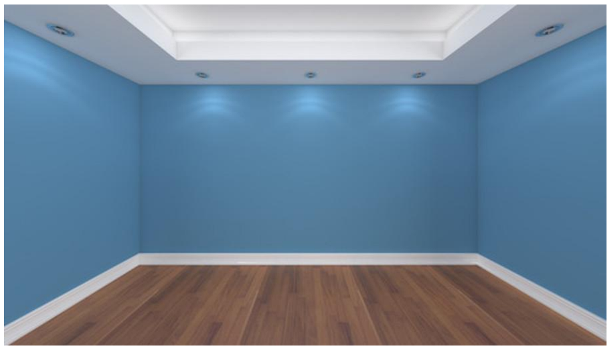
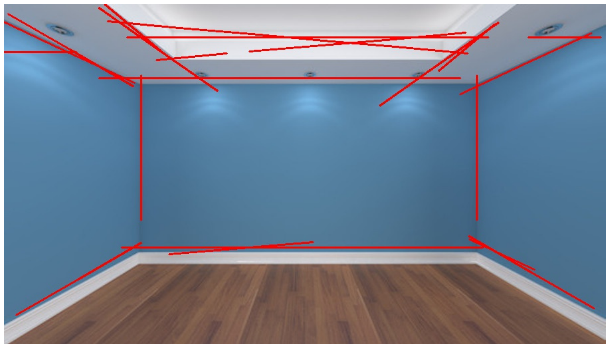
3. White walls with window and color ground
The walls of the previous conditions are always pure, what if we have something on it. Therefore, we test with the images which have windows on their walls. This would bring much challenge to us, because windows are always composed of straight lines, and distinguish them

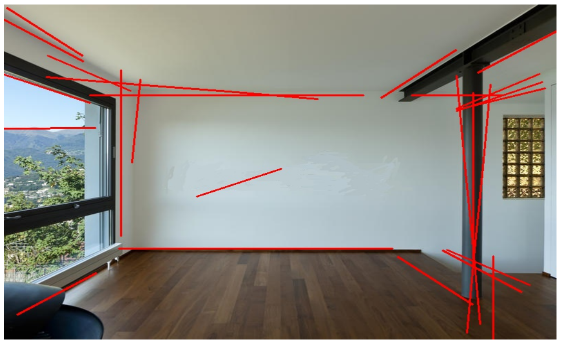
with the layout is not very easy. Our algorithm distinguishes them according to their length, and as shown in the results, we can get rid of some of the effect of the windows.

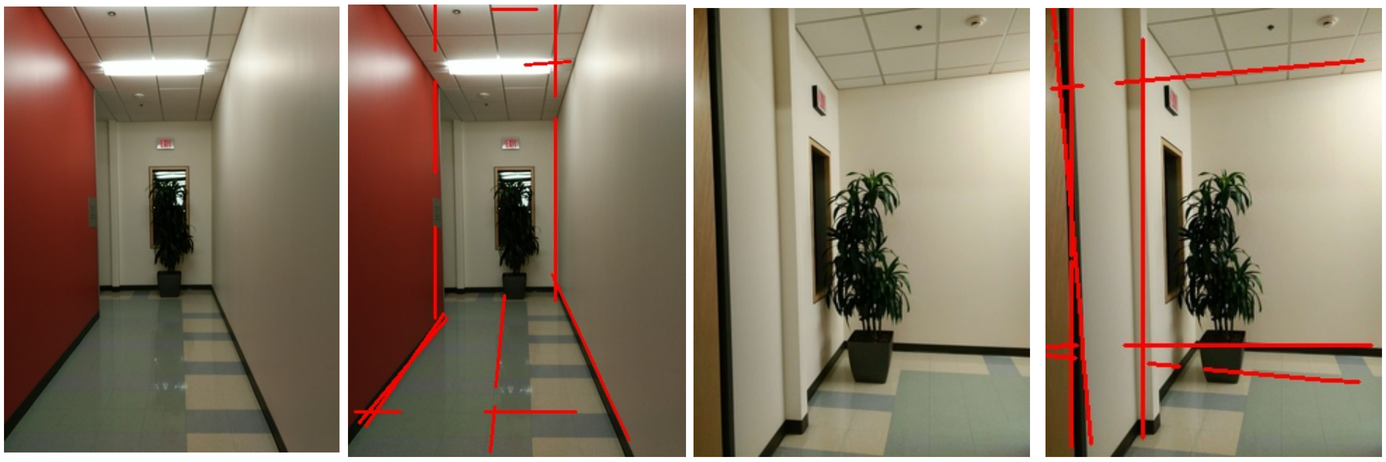


4. Color walls with color ground
All our cases above have the white walls, what if the walls and the ground all have color. As the color walls and color ground all contains noises, it becomes even harder for us to find the layout well. However, in our result, we can see that though the lines become more messy, they can still describe the layout well.

5. More complex conditions
Now the images we test are all about empty rooms. However, to make our algorithm much more general, it should also work for regular rooms. In the regular rooms, there may be textures on the walls or ground, and there may be some objects hang on the walls or placed on the ground, there may even be some objects exactly hide the layout. According to our results, our algorithm can figure out most of the layout well, but they are much more messy and incomplete than all the previous results.





From the results of all above, we can see that our algorithm is robust in many conditions. However, when the condition becomes more and more complex, the results become worse. If there are many objects containing straight lines or blocking the layout, or the color change and the shade are very complex, our algorithm will work poorly.

## Conclusion

We develop computer vision algorithm to find the spatial layout in the simple and clean indoor room environment or with a little cluster, the average error rate we obtained  is 16.8%. Our algorithm have proven its robustness and high accurracy in simple and clean room environment. In our algorithm, we have restrict our room into quite clean room while in reality it is not always that case. Also, there are some assumptions of the topology of the feature which may not always hold. In general, we have designed carefully to decrease the error rate of specific situation but the generalization of this algorithm is reduced. A more robust and general way to find the layout of room should take the information of the object in the room to promote the prediction of layout

## References

[1] Matessi A, Lombardi L. Vanishing point detection in the hough transform space[J]. Euro-Par'99 Parallel Processing, 1999: 987-994.
[2] Lutton E, Maitre H, Lopez-Krahe J. Contribution to the determination of vanishing points using Hough transform[J]. IEEE transactions on pattern analysis and machine intelligence, 1994, 16(4): 430-438.
[3]Denis P, Elder J H, Estrada F J. Efficient edge-based methods for estimating manhattan frames in urban imagery[C]//European conference on computer vision. Springer, Berlin, Heidelberg, 2008: 197-210.
[4] Coughlan J M, Yuille A L. Manhattan world: Orientation and outlier detection by bayesian inference[J]. Neural Computation, 2003, 15(5): 1063-1088.
[5] Ren Z, Sudderth E B. Three-dimensional object detection and layout prediction using clouds of oriented gradients[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2016: 1525-1533.

[6] Liu X, Veksler O, Samarabandu J. Graph cut with ordering constraints on labels and its applications[C]//Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on. IEEE, 2008: 1-8.
[7]Zhang W, Zhang W, Liu K, et al. Learning to Predict High-Quality Edge Maps for Room Layout Estimation[J]. IEEE Transactions on Multimedia, 2017, 19(5): 935-943.
[8] Explaination on official website of Opencv for Hough Transform in opencv2, including the meaning of the parameters and the return value for ithttps://docs.opencv.org/3.0-beta/doc/py_tutorials/py_imgproc/py_houghlines/py_houghlines.html
[9] An example on how to use kmeans function in opencv2 to classify a single columes of values into a determined number of groups as well as labeling each value with the group number. https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_ml/py_kmeans/py_kmeans_opencv/py_kmeans_opencv.html

## Program Documentation

### 7.1 Adaptive threshold

For this section, we first read in the original image as img, and make several copies of it in order to protect the original image from our following process. Then, we transfer the color image into gray image using opencv package cv.COLOR_RGB2GRAY. In order to remove the noises, we implement Gaussian filtering on the img. Finally, we apply adaptive threshold, using filtered image(img) and passing it to function adaptiveThreshold() to do adaptive thresholding. The mode we select is Adaptive_Threshold_Mean which cooperates with a mean filter and get the binary image thresh1.

After adaptive thresholding, the processed image is foreground-background inverse. It means that the pixels needed in edge detection become background pixels after thresholding. We switch the background pixels to foreground pixels and switch foreground pixels to background pixels and the result image will be passed down to edge detection. But there are still some noise on the thresholded image. If we apply filter to remove these noise, we will lose some details of the boundary either. To solve this problem, we apply opening(morphological filtering) to remove these small pixels but preserve the important edge information at the same time. Opening can be used to remove the object which is smaller than kernel. The kernel we choose is a circle with radius of 2 so that these small and scattered pixels can be removed.

### 7.2 Edge detection

For this section, we use the binary image we get in adaptive threshold. We first apply Sobel edge detection to thresh1 using Sobel() from opencv and get the local maximum on the edges in x-direction and y-direction, where the input of Sobel() are our image thresh1, the method cv.CV_16S, and 1 ,0 or 0, 1, which means we are using the first derivatives. Then, we use the root of the sum of x-square and y-square to form the sobel edge and we get the sobel edge image img2. After Sobel edge detection, we can find that the img2 becomes gray level image again, therefore, we make a threshold on img2 to get a binary image of edge as thresh.

However, the edges in the thresh are often incomplete, and there are still some noises. Therefore, we apply dilation and erosion on the thresh, where the input of dilation and erosion are our image thresh, our kernel with size 3*3, and the number of iterations, to make the edges connected and merge the noises, and get a new version of edge image img3. In img3 we can see that the edges are thick enough, but the merged noises generate many blocks which can affect our following detection. Therefore, we apply Canny edge detection on img3 again, where the input of Canny() are our image img3, the low threshold, the high threshold, and the aperture size. Then we get our final edge image edges.

### 7.3 Straight Line Hough Transform

When we get the final edge image, we start to find the straight line in the image. Here we apply the cv.HoughLines() from opencv, where the input parameters are our image edges, the distance search step, the angle search step, and the threshold of the line criteria (how many points form a line). Then we get the lines which are composed of their orientation and their distance from the origin.

### 7.4 Delete incorrect lines

We can get lots of lines from Hough Transform, and lots of them are formed because of the noises. Therefore, we need to delete those lines and remain the lines that are close enough to the layout of the room. Here we write a function findLen(), the input parameters are the lines and the edge image, and the output parameter is the new lines. We also write two functions to support the function findLen(), which are isbound() and islabel(). The input parameters of isbound() are the edge image and the x-coordinate, y-coordinate. This function is to find out whether a point (x, y) is within the bound of edge image. The input parameters of islabel() are the edge image and the x-coordinate, y-coordinate. This function is to find out whether there is a pixel within the bound of edge image and with value 255 in a kernel whose center is (x, y). In the findLen(), for each line, we know the orientation theta and the distance from the origin rho, then we can set the start point as (rho * cos(theta), tho * sin(theta)), and search the continuous overlap between the edge and the line using islabel(), the search step is 2 * (sin(theta), cos(theta)). If we find an overlap point, we add 1 to the count. Once we reach a point not overlapping, which means the point is far away from the edge, the count is reset to 0. Therefore, we can find a maximal count of a line, and we select the lines with the maximal count larger than the threshold we set to be the lines that fit the layout of the room, and output them as newline.

### 7.5 Combining parallel lines

In the newline we get from above, we can find out that there will be many parallel lines which are all close enough to the layout of the room, however, we only need one for each. Therefore, we have to merge the parallel lines to one line. We first sort the newline by their orientation theta. And for each theta, we sort the lines by their distance from origin. We set the lines with the shortest distance as the tag line of this region, and if the distance from the tag line is larger than the threshold we set, the line should belong to another region. Therefore, we get many regions which composed of the parallel lines close to each other. Finally, for each region, we add the distances together and divide them by the number of lines in this region to get the average distance ave of this region. Then then merged line will have the same orientation of lines in this region and have the distance of ave from the origin. The output here is called newlines.

## 7.6 Get the intersection points of lines

In the python file of intersecting_point.py, we define 3 functions to draw the intersection point coordinates by lines in 4 different slopes. The first function, segment_by_angle_kmeans(lines, k=4, **kwargs) is designed to classify the lines according to their values of θ into k (4) groups by the k means algorithm and label the same number on the lines which belong to the same group. By the value of labels this function can create a list with multiple aggregations of lines as item in it. Then the program only computes the intersection of lines in different groups. By literalizing every pair of groups and calculate the intersections between lines in that pair, we can get all the intersection points made by lines with different direction.

## 7.7. Get the line segments

All the lines we get from above pass through the whole image. In order to form the right result, we need to cut the lines into line segments and fit them with the layout. Here we write a function cutLen(), whose input parameters are the kernel size, the lines, the edge image, xhi, yhi, threshold1 and threshold2. And the output is a set of points. To support cutLen(), we write a function islabel1(), whose input parameters are the kernel size, the edge image, x-coordinate, y-coordinate. This function is to find out whether there is a pixel within the bound of edge image and with value 255 in a kernel whose center is (x, y) and size is k * k. In the cutLen(), for each line, we set the start point as (rho * cos(theta), tho * sin(theta)), and search the continuous overlap between the edge and the line using islabel1(), the search step is 2 * (sin(theta), cos(theta)). If islabel1() returns true, we set ncount as 0 which counts the continuous points not overlapping, and if the startflag is 0 which means the start point is not found yet, we set the current x and y as the coordinates of the start point, and set the startflag as 1. Then we add 1 to lcount which counts the continuous points overlapping. If islabel1() returns false, if startflag is 1 and endflag is 0, which means the start point is found but the end point is not found, we set the current x and y as the coordinates of the end point, but we do not set the endflag as 1 because this is not the true end point. If lcount is smaller than threshold2, which means the line segment is not long enough, we set startflag and lcount to 0. Then we add 1 to ncount. If ncount is larger than threshold1 and startflag is 1, which means the line segment should end, then we set the endflag to 1. Finally, if the startflag is 1 and endflag is 0, and lcount is larger than threshold2, which means the line segment is long enough but it reach the edge of the image, then we set the current x and y as the coordinates of the end point and set endflag as 1. If the startflag and the endflag are both 1, which means the start point and the end point have been found, then we output the points and set ncount, lcount, startflag and endflag back to 0. Finally, we use the start points and the end points to draw line segments on the image.

### 7. 8 Getting the layout

Using the lines we selected from former step, we have category the line by its geometric characteristics. Here, we find the intersection point by of lines and select desired points. The first step in function is to slide all the vertical or approximately vertical lines, and by calculating the intersection line with this vertical line, we could choose those with desired angle and with small distance. One or two vertical lines may be chosen as the primary line of this step. With this line, we would return 6 points in order if finding one wall line or 8 points if finding 2 wall lines.

# Appendix :Program Code

## main.py

```
# -*- coding: utf-8 -*-
import cv2 as cv
import numpy as np
import intersecting_point as intersect
import findPoint



def evaluation (input_image, output_points, groundtruth_points):
    s = 0.0
    if output_points == None:
        return 1;
    for i in range(len(output_points)):
            s += abs(output_points[i][0] - groundtruth_points[i][0]) + 
abs(output_points[i][1] - groundtruth_points[i][1]);
    error_rate = s / 
(input_image.shape[0]+input_image.shape[1])/len(groundtruth_points)
    print "the error rate is ", error_rate
    return error_rate

def isbound(edge, ix, iy):
    if ix >= 0 and ix < edge.shape[1] and iy >= 0 and iy < edge.shape[0]:
        return True;
    return False;


def islabel(edge, x, y):
    kernelsize = 1;
    ix = int(x);
```

```python
    iy = int(y);
    for i in range(-1, 2):
        for j in range(-1, 2):
            if isbound(edge, ix + i, iy + j) and edge[iy + j][ix + i] == 255:
                return True;

    return False;


def findLen(lines, edge):
    threshold = 30;
    newline = [];
    for i in range(len(lines)):
        for rho, theta in lines[i]:
            #     # draw a line
            a = np.cos(theta);
            b = np.sin(theta);
            x0 = a * rho;
            y0 = b * rho;
            #print(x0, y0);
            #print(b);
            #print(a);

            x = x0;
            y = y0;
            count = 0;
            maxnum = 0;
            num = 0;

            while num < 1000:
                if islabel(edge, x, y) == True:
                    count += 1;
                    maxnum = max(maxnum, count);
                else:
                    count = 0;
                x = x + 2 * (-b);
                y = y + 2 * a;
                num += 1;
            if maxnum > threshold:
                newline.append([[rho, theta]]);
                continue;

            x = x0;
            y = y0;
            count = 0;
            maxnum = 0;
            num = 0;

            while num < 1000:
                if islabel(edge, x, y) == True:
                    count += 1;
                    maxnum = max(maxnum, count);
                else:
                    count = 0;
                x = x - 2 * (-b);
                y = y - 2 * a;
                num += 1;
            if maxnum > threshold:
                newline.append([[rho, theta]]);
    return newline
```

```python
img=cv.imread("test10.jpg")
print("preprocessing...")
result1 = img.copy()

img = cv.cvtColor(img, cv.COLOR_RGB2GRAY)
img = cv.GaussianBlur(img, (3, 3), 0)

thresh1 = cv.adaptiveThreshold(img,255,cv.ADAPTIVE_THRESH_MEAN_C,\
            cv.THRESH_BINARY,11,2)

temp=thresh1.shape

for i in range(temp[0]):
    for j in range(temp[1]):
        if (thresh1[i][j]==255):
            thresh1[i][j]=0
        else:
            thresh1[i][j]=255

kernel1 = np.ones((2,2),np.uint8)
thresh1 = cv.morphologyEx(thresh1, cv.MORPH_OPEN, kernel1)
#cv.imwrite("test.jpg", thresh1)

x = cv.Sobel(thresh1, cv.CV_16S, 1, 0)
y = cv.Sobel(thresh1, cv.CV_16S, 0, 1)
absX = cv.convertScaleAbs(x)  # uint8
absY = cv.convertScaleAbs(y)
edges = cv.addWeighted(absX, 0.5, absY, 0.5, 0)
cv.imwrite("SR.jpg", edges)# sobel detection

img2=cv.imread("SR.jpg")
#img2=cv.imread("sobelresult.jpg")
ret,thresh = cv.threshold(img2,10,255,cv.THRESH_BINARY)
kernel = np.ones((5,5),np.uint8)
dilation = cv.dilate(thresh,kernel,iterations = 2)
erosion = cv.erode(dilation,kernel,iterations = 2)
#edges = cv.Canny(thresh, 50, 150, apertureSize=3)
#dst=morphology.remove_small_objects(edges,min_size=10000,connectivity=2)
cv.imwrite("edge.jpg", erosion)#

img3=cv.imread("edge.jpg")
edges = cv.Canny(img3, 50, 150, apertureSize=3)
kernel = np.ones((2,2),np.uint8)
edges = cv.dilate(edges,kernel,iterations = 1)
print("Hough transform...")
lines = cv.HoughLines(edges, 1, np.pi / 36, 75)

print("Deleting lines...")
newline = findLen(lines,edges)
#print(lines)
# combine lines
print("Combining lines...")
threshold = 35
theta = []
num = 0
number = 0
sum_num = []
rho_new = []
theta_new = []
for line in newline:
    if line[0][1] not in theta:
        theta.append(line[0][1])
        num += 1
for i in range(0,num):
    distance = []
    dnum = 0
    for line in newline:
        if line[0][1]==theta[i]:
            distance.append(line[0][0])
            dnum += 1
    #print(distance)
    for j in range(0,dnum):
        for k in range(j+1,dnum):
            if distance[j]>distance[k]:
                temp = distance[j]
                distance[j]=distance[k]
                distance[k]=temp
    #print(distance)
    tag = []
    tag.append(distance[0])
    tagnum = 1
    for j in range(0,dnum):
        if abs(distance[j]-tag[tagnum-1])>threshold:
            tag.append(distance[j])
            tagnum += 1
    #print(tag)
    for j in range(0,tagnum):
        temp = 0
        sum = 0
        for line in newline:
            if line[0][1]==theta[i]:
                if abs(line[0][0]-tag[j])<=threshold:
                    sum += line[0][0]
                    temp += 1
        avg = sum/temp
        theta_new.append(theta[i])
        rho_new.append(avg)
        sum_num.append(temp)
        number += 1
newlines = []
for i in range(0,number):
    newlines.append([[rho_new[i],theta_new[i]]])
print("Searching intersect points...")
segmented = intersect.segment_by_angle_kmeans(newlines)
intersections = intersect.segmented_intersections(segmented)
generate_point = findPoint.findTargetPoint(newlines,intersections,result1);
result = evaluation(result1,generate_point,ground_truth);
file = open('testresult.txt','a');
file.write(str(result) + '\n');
file.close();

cv.imshow("Result", result1);
cv.waitKey(0);
cv.destroyAllWindows();
```

## findPoint.py

```
# -*- coding: utf-8 -*-
import cv2 as cv
import numpy as np
import intersecting_point as intersect
import math;


'''the function of find Target Point is to find the intersection point with topology
features, there would be lots
lots of details. Readers could skip some part of this'''
def drawlin(line,result1):
   rho = line[0][0];
   theta = line[0][1];
   a = np.cos(theta)
   b = np.sin(theta)
   x0 = a*rho
   y0 = b*rho
   x1 = int(x0 + 1000*(-b))
   y1 = int(y0 + 1000*(a))
   x2 = int(x0 - 1000*(-b))
   y2 = int(y0 - 1000*(a))
   cv.line(result1,(x1,y1),(x2,y2),(255,0,0),1);

def isbound(edge, ix, iy):
    if ix >= 0 and ix < edge.shape[1] and iy >= 0 and iy < edge.shape[0]:
        return True;
    return False;

def findBoundaryPoint(lines,mode,result1):
   if mode == 0:
      #find the intersection with the left up boundary
      line = lines[0];

      rho = line[0][0];
      theta = line[0][1];
      a = np.cos(theta)
      b = np.sin(theta)
      x0 = a*rho;
      y0 = b*rho;
      num = 0;
      x = x0;
      y = y0;
      while num < 1000:
         x = x + (-b);
         y = y + a;
         if abs(x) < 1.5 and isbound(result1,int(x),int(y)):
            return (int(x),int(y));
         if abs(y) < 1.5  and isbound(result1,int(x),int(y)):
            return (int(x),int(y));
         num += 1;
      num = 0;
      x = x0;
      y = y0;
      while num < 1000:
         x = x - (-b);
         y = y - a;
         if abs(x) < 1.5 and isbound(result1,int(x),int(y)):
            return (int(x),int(y));
         if abs(y) < 1.5  and isbound(result1,int(x),int(y)):
            return (int(x),int(y));
         num += 1;
      return (-1,-1);

   if mode == 1:
      #find the intersection with the left down boundary
      print("mode 1")
      line = lines[1];

      rho = line[0][0];
      theta = line[0][1];
      a = np.cos(theta)
      b = np.sin(theta)
      x0 = a*rho;
      y0 = b*rho;
      num = 0;
      x = x0;
      y = y0;
      while num < 1000:
         x = x + (-b);
         y = y + a;
         if abs(x) < 1.5 and isbound(result1,int(x),int(y)):
            return (int(x),int(y));
         if y > result1.shape[0] - 1.5  and isbound(result1,int(x),int(y)):
            return (int(x),int(y));
         num += 1;
      num = 0;
      x = x0;
      y = y0;
      while num < 1000:
         x = x - (-b);
         y = y - a;
         if abs(x) < 1.5 and isbound(result1,int(x),int(y)):
            return (int(x),int(y));
         if y > result1.shape[0] - 1.5 and isbound(result1,int(x),int(y)):
            return (int(x),int(y));
         num += 1;
      return (-1,-1);

   if mode == 2:
      #find the intersection with the right up  boundary;
      line = lines[1];

      rho = line[0][0];
      theta = line[0][1];
      a = np.cos(theta)
      b = np.sin(theta)
      x0 = a*rho;
      y0 = b*rho;
      num = 0;
      x = x0;
      y = y0;
      while num < 1000:
         x = x - (-b);
         y = y - a;
         if abs(x) > result1.shape[1] - 1.5 and isbound(result1,int(x),int(y)):
            return (int(x),int(y));
         if abs(y) < 1.5 and isbound(result1,int(x),int(y)):
            return (int(x),int(y));
         num += 1;
      return (-1,-1);

      num = 0;
      x = x0;
      y = y0;
      while num < 1000:
         x = x + (-b);
         y = y + a;
         if abs(x) > result1.shape[1] - 1.5 and isbound(result1,int(x),int(y)):
            return (int(x),int(y));
         if abs(y) < 1.5 and isbound(result1,int(x),int(y)):
            return (int(x),int(y));
         num += 1;


   if mode == 3:
      #find the intersection with the right down  boundary;
      line = lines[0];

      rho = line[0][0];
      theta = line[0][1];
      a = np.cos(theta)
      b = np.sin(theta)
      x0 = a*rho;
      y0 = b*rho;
      num = 0;
      x = x0;
      y = y0;
      while num < 1000:
         x = x + (-b);
         y = y + a;
         if abs(x) > result1.shape[1] - 1.5 and isbound(result1,int(x),int(y)):
            return (int(x),int(y));
         if abs(y) > result1.shape[0] - 1.5 and isbound(result1,int(x),int(y)):
            return (int(x),int(y));
         num += 1;
      num = 0;
      x = x0;
      y = y0;
      while num < 1000:
         x = x - (-b);
         y = y - a;
         if abs(x) > result1.shape[1] - 1.5 and isbound(result1,int(x),int(y)):
            return (int(x),int(y));
         if abs(y) > result1.shape[0] - 1.5 and isbound(result1,int(x),int(y)):
            return (int(x),int(y));
         num += 1;
      return (-1,-1);

def drawline(line,result1):
   print("ready to draw");
   for i in range(len(line)):
      for rho,theta in line[i]:
         a = np.cos(theta)
         b = np.sin(theta)
         x0 = a*rho
         y0 = b*rho
         x1 = int(x0 + 1000*(-b))
         y1 = int(y0 + 1000*(a))
         x2 = int(x0 - 1000*(-b))
         y2 = int(y0 - 1000*(a))
      cv.line(result1,(x1,y1),(x2,y2),(255,255,255),1);
```

```
def distance(point1,point2):
   return math.sqrt((point1[0] - point2[0])*(point1[0] - point2[0]) + (point1[1] -
point2[1])*(point1[1] - point2[1]));

def findTargetPoint(lines,intersection,result1):
   #first find all potential lines,using the condition of the angle of the wall line

   threshold = 15;

   targetlinevertical = [];
   for i in range(len(lines)):
      for rho,theta in lines[i]:
         if abs(theta) < 0.01:
            targetlinevertical.append([[rho,theta]]);



   targetlinehorizontal = [];
   for i in range(len(lines)):
      for rho,theta in lines[i]:
         if abs(theta - np.pi/2) < 0.02:
            targetlinehorizontal.append([[rho,theta]]);

   drawline(targetlinehorizontal,result1);

   targetlineangle1 = []
   for i in range(len(lines)):
      for rho,theta in lines[i]:
         if theta > np.pi/2  and theta < np.pi  - np.pi/9:
            targetlineangle1.append([[rho,theta]]);


   drawline(targetlineangle1,result1);



   targetlineangle2 = []
   for i in range(len(lines)):
      for rho,theta in lines[i]:
         if theta > np.pi/9 and theta < np.pi/2 -0.01:
            targetlineangle2.append([[rho,theta]]);

   drawline(targetlineangle2,result1);

   if len(targetlinevertical) == 0:
      print("sorry I can't find line");
      return;



   #the first topology
   primarylinelist = [];

   primarypointlist = []
   primarypointlist2 = [];
   print("Ready to find first topology")
   for i in range(len(targetlinevertical)):
      primaryline = False;
      highpoint = 10000;
      lowpoint = -1;
      for j in range(len(targetlineangle1)):
         for k in range(len(targetlineangle2)):
            point1 = intersect.intersection(targetlineangle1[j],targetlinevertical[i]);
            point2 = intersect.intersection(targetlineangle2[k],targetlinevertical[i]);
            if distance(point1,point2) < threshold:
               highpoint = min(point1[1],point2[1],highpoint);
               lowpoint = max(point1[1],point2[1],lowpoint);
               remainj = j;
               remaink = k;
               primaryline = True;
      if primaryline == True:
         primarylinelist.append(targetlinevertical[i]);
         primarypointlist.append(highpoint);
         primarypointlist2.append(lowpoint);

   primarylinelist2 = [];


   for i in range(len(primarylinelist)):
      if primarypointlist2[i] - primarypointlist[i] > 50:
         primarylinelist2.append(primarylinelist[i]);

   if len(primarylinelist2) == 0 and len(targetlinevertical) == 1:
      print("I can some line")
      primarypointlist2.append(targetlinevertical[0]);

   if len(primarylinelist2) != 0:
      #we just find the primaryline with min highpoint value
      drawline(primarylinelist2,result1);
      primarypointlist = []
      primarypointlist2 = [];
      for i in range(len(primarylinelist2)):
         primaryline = False;
         highpoint = 10000;
         lowpoint = -1;
         for j in range(len(targetlineangle1)):
            for k in range(len(targetlineangle2)):
               point1 = 
intersect.intersection(targetlineangle1[j],primarylinelist2[i]);
               point2 = 
intersect.intersection(targetlineangle2[k],primarylinelist2[i]);
               if distance(point1,point2) < threshold:
                  highpoint = min(point1[1],point2[1],highpoint);
                  lowpoint = max(point1[1],point2[1],lowpoint);
                  remainj = j;
                  remaink = k;
                  primaryline = True;
         if primaryline == True:
            primarylinelist.append(primarylinelist2[i]);
            primarypointlist.append(highpoint);
            primarypointlist2.append(lowpoint);
      targeti = -1;
      Min = 10000;
      for i in range(len(primarypointlist2)):
         if primarypointlist[i] < Min:
            Min = primarypointlist[i];
            targeti = i;

      #get the target line for result
      print("find the target primary line")
      targetline = primarylinelist2[i];
      drawlin(targetline,result1);
      #given this line, get the two result in the threshold
      Min = 10000;
      Max = -1;
      count = 0;
      for i in range(len(targetlineangle1)):
         for j in range(len(targetlineangle2)):
            point1 = intersect.intersection(targetlineangle1[i],targetline);
            point2 = intersect.intersection(targetlineangle2[j],targetline);
            if distance(point1,point2) < threshold * 3:
               if (point1[1] + point2[1])/2 < Min:
                  Min = (point1[1] + point2[1])/2;
                  targetpointmin = ((point1[0] + point2[0])/2,(point1[1] +
point2[1])/2);
                  Minline = (targetlineangle1[i],targetlineangle2[j]);
               if (point1[1] + point2[1])/2 > Max:
                  Max = (point1[1] + point2[1])/2;
                  targetpointmax = ((point1[0] + point2[0])/2,(point1[1] +
point2[1])/2);
                  Maxline = (targetlineangle1[i],targetlineangle2[j]);

      targetpoint1 = targetpointmin;
      targetpoint2 = targetpointmax;
      #after getting the maxline and min line, find the boundary point by finding the
boundary
      print("find the target primary point");
      print(targetpoint1);
      print(targetpoint2);


      targetpoint3 = findBoundaryPoint(Minline,0,result1);
      print(targetpoint3);
      #drawline(targetlineangle1,result1);
      targetpoint4 = findBoundaryPoint(Maxline,1,result1);
      print(targetpoint4);
      targetpoint5 = findBoundaryPoint(Minline,2,result1);
      print(targetpoint5);
      targetpoint6 = findBoundaryPoint(Maxline,3,result1);
      print(targetpoint6);
      targetpoint =
(targetpoint1,targetpoint2,targetpoint3,targetpoint4,targetpoint5,targetpoint6);
      for point in targetpoint:
         cv.circle(result1,point,2,(0,0,255),thickness=1);
      return targetpoint;




   else:
      #there is no primary line in the list, so check the second situation
      leftlinelist = []
      leftpointlist = [];
      for i in range(len(targetlinevertical)):
         leftline = False;
         leftpoint = 1000;
         for j in range(len(targetlineangle1)):
            for k in range(len(targetlinehorizontal)):
               point1 =
intersect.intersection(targetlineangle1[j],targetlinevertical[i]);
               point2 =
intersect.intersection(targetlinehorizontal[k],targetlinevertical[i]);
               if distance(point1,point2) < threshold:
                  leftline = True;
                  leftpoint = point1[0];
         if leftline == True:
            leftlinelist.append(targetlinevertical[i]);
            leftpointlist.append(point1[0]);


      if len(leftlinelist) == 0:

         print("sorry,no left line, I have to return")
         return;
      targeti = -1;
      Min = 10000;
      for i in range(len(leftlinelist)):
         if leftpointlist[i] < Min:
            Min = leftpointlist[i];
            targeti = i;

      targetleftline = leftlinelist[targeti];
      rightlinelist = []
      rightpointlist = [];
      for i in range(len(targetlinevertical)):
         rightline = False;
         rightpoint = -1;
         for j in range(len(targetlineangle2)):
            for k in range(len(targetlinehorizontal)):
               point1 =
intersect.intersection(targetlineangle2[j],targetlinevertical[i]);
               point2 =
intersect.intersection(targetlinehorizontal[k],targetlinevertical[i]);
               if distance(point1,point2) < threshold:
                  rightpoint = point1[0];
                  rightline = True;
         if rightline == True:
            rightlinelist.append(targetlinevertical[i]);
            rightpointlist.append(rightpoint);


      targeti = -1;
      Max = -1;
      if len(rightlinelist) == 0:
         print("sorry,no right line, I have to return")
         return;
      for i in range(len(rightlinelist)):

         if rightpointlist[i] > Max:
            Max = rightpointlist[i];
            targeti = i;
      targetrightline = rightlinelist[targeti];


      #get point similarily from primary line
      Min = 10000;
      Max = -1;
      count = 0;
      for i in range(len(targetlineangle1)):
         for j in range(len(targetlinehorizontal)):
            point1 = intersect.intersection(targetlineangle1[i],targetleftline);
            point2 = intersect.intersection(targetlinehorizontal[j],targetleftline);
            if distance(point1,point2) < threshold:
               if (point1[1] + point2[1])/2 < Min:
                  Min = (point1[1] + point2[1])/2;
                  targetpointmin = ((point1[0] + point2[0])/2,(point1[1] +
point2[1])/2);
                  Minline = (targetlineangle1[i],targetlinehorizontal[j]);

      for i in range(len(targetlineangle2)):
         for j in range(len(targetlinehorizontal)):
            point1 = intersect.intersection(targetlineangle2[i],targetleftline);
            point2 = intersect.intersection(targetlinehorizontal[j],targetleftline);
            if distance(point1,point2) < threshold * 2:
               if (point1[1] + point2[1])/2 > Max:
                  Max = (point1[1] + point2[1])/2;
                  targetpointmax = ((point1[0] + point2[0])/2,(point1[1] +
point2[1])/2);
                  Maxline = (targetlinehorizontal[j],targetlineangle2[i]);


      targetpoint1 = targetpointmin;
      targetpoint2 = targetpointmax;
      targetpoint5 = findBoundaryPoint(Minline,0,result1);
      targetpoint6 = findBoundaryPoint(Maxline,1,result1);
      print(targetpoint1);
      print(targetpoint2);
      print(targetpoint5);
      print(targetpoint6);

      Min = 10000;
      Max = -1;
      count = 0;

      for i in range(len(targetlineangle1)):
         for j in range(len(targetlinehorizontal)):
            point1 = intersect.intersection(targetlineangle1[i],targetrightline);
            point2 = intersect.intersection(targetlinehorizontal[j],targetrightline);
            if distance(point1,point2) < 2 * threshold:
               if (point1[1] + point2[1])/2 > Max:
                  Max = (point1[1] + point2[1])/2;
                  targetpointmax = ((point1[0] + point2[0])/2,(point1[1] +
point2[1])/2);
                  Maxline = (targetlineangle1[i],targetlinehorizontal[j]);

      for i in range(len(targetlineangle2)):
         for j in range(len(targetlinehorizontal)):
            point1 = intersect.intersection(targetlineangle2[i],targetrightline);
            point2 = intersect.intersection(targetlinehorizontal[j],targetrightline);
            if distance(point1,point2) < threshold * 2:
               if (point1[1] + point2[1])/2 < Min:
                  Min = (point1[1] + point2[1])/2;
                  targetpointmin = ((point1[0] + point2[0])/2,(point1[1] +
point2[1])/2);
                  Minline = (targetlinehorizontal[j],targetlineangle2[i]);

      targetpoint3 = targetpointmax;
      targetpoint4 = targetpointmin;
      targetpoint7 = findBoundaryPoint(Maxline,3,result1);
      targetpoint8 = findBoundaryPoint(Minline,2,result1);
      print(targetpoint3);
      print(targetpoint4);
      print(targetpoint7);
      print(targetpoint8);
      targetpoint = 
(targetpoint1,targetpoint2,targetpoint3,targetpoint4,targetpoint5,targetpoint6,targetp
oint7,targetpoint8);
      for point in targetpoint:
         cv.circle(result1,point,2,(0,0,255),thickness=1);
      return targetpoint;
```

## Intersection_point.py

```
from collections import defaultdict
import cv2
import numpy as np

def segment_by_angle_kmeans(lines, k=2, **kwargs):
    """Groups lines based on angle with k-means.

    Uses k-means on the coordinates of the angle on the unit circle
    to segment `k` angles inside `lines`.
    """

    # Define criteria = (type, max_iter, epsilon)
    default_criteria_type = cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER
    criteria = kwargs.get('criteria', (default_criteria_type, 10, 1.0))
    flags = kwargs.get('flags', cv2.KMEANS_RANDOM_CENTERS)
    attempts = kwargs.get('attempts', 10)

    # returns angles in [0, pi] in radians
    angles = np.array([line[0][1] for line in lines])
    # multiply the angles by two and find coordinates of that angle
    pts = np.array([[np.cos(2*angle), np.sin(2*angle)]
                    for angle in angles], dtype=np.float32)

    # run kmeans on the coords
    labels, centers = cv2.kmeans(pts, k, None, criteria, attempts, flags)[1:]
    labels = labels.reshape(-1)  # transpose to row vec

    # segment lines based on their kmeans label
    segmented = defaultdict(list)
    for i, line in zip(range(len(lines)), lines):
        segmented[labels[i]].append(line)
    segmented = list(segmented.values())
    return segmented

def intersection(line1, line2):
    """Finds the intersection of two lines given in Hesse normal form.

    Returns closest integer pixel locations.
    See https://stackoverflow.com/a/383527/5087436
    """
    rho1, theta1 = line1[0]
    rho2, theta2 = line2[0]
    A = np.array([
        [np.cos(theta1), np.sin(theta1)],
        [np.cos(theta2), np.sin(theta2)]
    ])
    b = np.array([[rho1], [rho2]])
    x0, y0 = np.linalg.solve(A, b)
    x0, y0 = int(np.round(x0)), int(np.round(y0))
    return (x0, y0)
```

```
def segmented_intersections(lines):
    """Finds the intersections between groups of lines."""

    intersections = []
    for i, group in enumerate(lines[:-1]):
        for next_group in lines[i+1:]:
            for line1 in group:
                for line2 in next_group:
                    intersections.append(intersection(line1, line2))

    return intersections
```

## cut.py

```
import cv2 as cv
import numpy as np
import intersecting_point as intersect
from skimage import morphology

def isbound(edge, ix, iy): # determine whether the point (ix, iy) is within the bound of edge
  if ix >= 0 and ix < edge.shape[1] and iy >= 0 and iy < edge.shape[0]:
    return True;
  return False;

def islabel(edge, x, y): # determine whether there is pixel of edge that have the value 255 within the kernel whose
  kernelsize = 2;          # center is (x, y) and kernelsize is 2*2, if within, it means the point is close enough to the edge
  ix = int(x);
  iy = int(y);
  for i in range(-1, 2):
    for j in range(-1, 2):
      if isbound(edge, ix + i, iy + j) and edge[iy + j][ix + i] == 255:
        return True;

  return False;

def islabel1(k,edge, x, y): # determine whether there is pixel of edge that have the value 255 within the kernel whose
  ix = int(x);                # center is (x, y) and kernelsize is k*k, if within, it means the point is close enough to the edge
  iy = int(y);
  for i in range(-k, k+1):
    for j in range(-k, k+1):
      if isbound(edge, ix + i, iy + j) and edge[iy + j][ix + i] == 255:
        return True

  return False;

def findLen(lines, edge): # find lines that are close enough to the edge
  threshold = 30; # the minimum of number of points that could form a line
  newline = [];
  for i in range(len(lines)):
    for rho, theta in lines[i]:
      a = np.cos(theta);
      b = np.sin(theta);
      x0 = a * rho; # start point
      y0 = b * rho;
      #print(x0, y0);
      #print(b);
      #print(a);

      x = x0; # find the longest continuous line in one direction
      y = y0;
      count = 0; # count the number of continuous points
      maxnum = 0; # the largest number of continuous points
      num = 0; # how many points on a line to search

      while num < 1000:
        if islabel(edge, x, y) == True:
            count += 1;
            maxnum = max(maxnum, count); # record the maximum of count
        else:
            count = 0; # once the line is not continuous, reset count
        x = x + 2 * (-b); # move the search kernel in one direction
        y = y + 2 * a;
        num += 1;
      if maxnum > threshold: # if the max length of continuous line is larger than threshold, the line should remain
        newline.append([[rho, theta]]);
        continue;

      x = x0; # find the longest continuous line in the other direction
      y = y0;
      count = 0;
      maxnum = 0;
      num = 0;

      while num < 1000:
        if islabel(edge, x, y) == True:
            count += 1;
            maxnum = max(maxnum, count);
        else:
            count = 0;
        x = x - 2 * (-b);
        y = y - 2 * a;
        num += 1;
      if maxnum > threshold:
        newline.append([[rho, theta]]);
  return newline

def cutLen(k, lines, edge, xhi, yhi, threshold1, threshold2): #use k*k kernel to search along the lines within the image
  point = [];  # which size is xhi*yhi, remain the lines longer than threshold2, if the blank is longer than threshold1, a
  for i in range(len(lines)): # new line segment should be started
    for rho, theta in lines[i]:
      a = np.cos(theta);
```

```
b = np.sin(theta);
x0 = a * rho; # start point
y0 = b * rho;
if x0<0:
    x0=0
    y0=rho/b
if y0<0:
    y0=0
    x0=rho/a
#print(x0, y0);
#print(b);
#print(a);

x = x0; # find the line segments in one direction
y = y0;
ncount = 0; # the number of continuous blank points
lcount = 0; # the number of continuous edge points close enough to the line (within the kernel)
xstart = ystart = xend = yend = 0; # coordinate of start point and end point of a line segment
startflag = endflag = 0 # whether the start point/ end point has been found

while x>=0 and x<=xhi and y>=0 and y<=yhi: # within the image
    if islabel1(k,edge, x, y) == True: # if find edge point within kernel
        ncount = 0; #reset ncount
        if startflag == 0: # if start point not found, set this point as start point and set startflag to 1
            xstart = x
            ystart = y
            startflag = 1
        lcount += 1; # update lcount
    else: # if no edge point in kernel
        if startflag == 1 and endflag == 0: # if start point is found but end point is not found
            xend = x  # set this point as temporary end point
            yend = y
        if lcount < threshold2: # if the continuous line is not long enough, delete it and reset lcount
            startflag = 0
            lcount = 0
        ncount += 1; # update ncount
        if ncount > threshold1 and startflag == 1: # if the continuous blank is long enough, end the line segment
            endflag = 1
    x = x + 2 * (-b); # move to the next point in one direction
    y = y + 2 * a;
    if startflag == 1 and endflag == 0: # if the line segment is long enough but still can't find the end point
        if lcount>threshold2:                # set the current point as end point
            xend = x
            yend = y
            endflag=1
    if startflag == 1 and endflag == 1: # if start point and end point are both found, output them and reset
        point.append((int(xstart), int(ystart)));
        point.append((int(xend), int(yend)));
        ncount = 0
        lcount = 0
        startflag = endflag = 0
        x = x0; # find the line segment in another direction
        y = y0;
        ncount = 0;
        lcount = 0;
        xstart = ystart = xend = yend = 0;
        startflag = endflag = 0

        while x>=0 and x<=xhi and y>=0 and y<=yhi:
            if islabel1(k,edge, x, y) == True:
                ncount = 0;
                if startflag == 0:
                    xstart = x
                    ystart = y
                    startflag = 1
                lcount += 1;
            else:
                if startflag == 1 and endflag == 0:
                    xend = x
                    yend = y
                if lcount < threshold2:
                    startflag = 0
                    lcount = 0
                ncount += 1;
                if ncount > threshold1 and startflag == 1:
                    endflag = 1
            x = x - 2 * (-b);
            y = y - 2 * a;
            if startflag == 1 and endflag == 0:
                if lcount>threshold2:
                    xend = x
                    yend = y
                    endflag=1
            if startflag == 1 and endflag == 1:
                point.append((int(xstart), int(ystart)));
                point.append((int(xend), int(yend)));
                ncount = 0
                lcount = 0
                startflag = endflag = 0

    return point

input = "room.jpg"
output = "room-result.jpg"

img=cv.imread(input) # read in the image
print("Preprocessing...")
result1 = img.copy()
result2 = img.copy()
result3 = img.copy()
result4 = img.copy()
result5 = img.copy()

img = cv.cvtColor(img, cv.COLOR_RGB2GRAY) # transform to gray level image
img = cv.GaussianBlur(img, (3, 3), 0) # Gaussian filter, remove the noise

thresh1 = cv.adaptiveThreshold(img,255,cv.ADAPTIVE_THRESH_MEAN_C,\
          cv.THRESH_BINARY,11,2) # adaptive threshold

size=thresh1.shape

for i in range(size[0]): # inverse the image
   for j in range(size[1]):
       if (thresh1[i][j]==255):
          thresh1[i][j]=0
       else:
          thresh1[i][j]=255

kernel1 = np.ones((2,2),np.uint8) # 2*2 kernel
thresh1 = cv.morphologyEx(thresh1, cv.MORPH_OPEN, kernel1) # opening
cv.imwrite("room-adaptive_threshold.jpg", thresh1)

x = cv.Sobel(thresh1, cv.CV_16S, 1, 0) # sobel edge detection, use first derivative in x-direction
y = cv.Sobel(thresh1, cv.CV_16S, 0, 1) # sobel edge detection, use first derivative in y-direcation
absX = cv.convertScaleAbs(x)  # transform back into uint8
absY = cv.convertScaleAbs(y)
edges = cv.addWeighted(absX, 0.5, absY, 0.5, 0) # compute the edges
cv.imwrite("room-sobel.jpg", edges)# sobel edge

img2=cv.imread("room-sobel.jpg")
#img2=cv.imread("sobelresult.jpg")
ret,thresh = cv.threshold(img2,10,255,cv.THRESH_BINARY) # tranform the edge image into binary image
kernel = np.ones((5,5),np.uint8) # 5*5 kernel
dilation = cv.dilate(thresh,kernel,iterations = 2) # morphological process, make the edge more complete and thicker
erosion = cv.erode(dilation,kernel,iterations = 2) # and merge the remaining noises
#edges = cv.Canny(thresh, 50, 150, apertureSize=3)
#dst=morphology.remove_small_objects(edges,min_size=10000,connectivity=2)
cv.imwrite("room-edge.jpg", erosion)

img3=cv.imread("room-edge.jpg")
B, G, R = cv.split(img3)
edges = cv.Canny(img3, 50, 150, apertureSize=3)# canny edge detection
kernel = np.ones((2,2),np.uint8) # 2*2 kernel
edges = cv.dilate(edges,kernel,iterations = 1) # dilation, make the edge thicker and connected
cv.imwrite("room-canny.jpg", edges)
print("Hough transform...")
lines = cv.HoughLines(edges, 1, np.pi / 36, 75)  # straight line hough transform, (image, distance step, angle step,
for i in range(len(lines)):                                   # threshold)
       for rho,theta in lines[i]:
          a = np.cos(theta)
          b = np.sin(theta)
          x0 = a*rho
          y0 = b*rho
          x1 = int(x0 + 2000*(-b))
          y1 = int(y0 + 2000*(a))
```

```
      x2 = int(x0 - 2000*(-b))
      y2 = int(y0 - 2000*(a))

    cv.line(result1,(x1,y1),(x2,y2),(0,0,255),1) # draw the lines on result1
cv.imwrite("room-hough.jpg",result1)
print("Deleting lines...")
newline = findLen(lines,edges) # delete incorrect lines
for i in range(len(newline)):
    for rho,theta in newline[i]:
      a = np.cos(theta)
      b = np.sin(theta)
      x0 = a*rho
      y0 = b*rho
      x1 = int(x0 + 2000*(-b))
      y1 = int(y0 + 2000*(a))
      x2 = int(x0 - 2000*(-b))
      y2 = int(y0 - 2000*(a))

    cv.line(result2,(x1,y1),(x2,y2),(0,0,255),1) # draw the new line on result2
cv.imwrite("room-deleteline.jpg",result2)
#print(lines)
# merging the parallel lines
print("Combining lines...")
threshold = 35 # the distance between different regions of lines
theta = [] # angles of lines
num = 0 # number of different angles
number = 0 # number of result lines
sum_num = [] # sum of distances of lines from origin in each region
rho_new = [] # distances of new lines from origin
theta_new = [] # orientations of new lines
for line in newline: # record different theta
  if line[0][1] not in theta:
    theta.append(line[0][1])
    num += 1
for i in range(0,num): # for each theta
  distance = []
  dnum = 0
  for line in newline:
    if line[0][1]==theta[i]:
      distance.append(line[0][0])
      dnum += 1
  #print(distance)
  for j in range(0,dnum): # sort the lines of this theta by distance from origin
    for k in range(j+1,dnum):
      if distance[j]>distance[k]:
        temp = distance[j]
        distance[j]=distance[k]
        distance[k]=temp
  #print(distance)
  tag = [] # tag line of each region
  tag.append(distance[0])
  tagnum = 1
```

```
    for j in range(0,dnum):
        if abs(distance[j]-tag[tagnum-1])>threshold: # if distance out of range, start a new region
            tag.append(distance[j])
            tagnum += 1
    #print(tag)
    for j in range(0,tagnum): # for each region, compute the average distance
        temp = 0
        sum = 0
        for line in newline:
            if line[0][1]==theta[i]:
                if abs(line[0][0]-tag[j])<=threshold:
                    sum += line[0][0]
                    temp += 1
        avg = sum/temp
        theta_new.append(theta[i])
        rho_new.append(avg)
        sum_num.append(temp)
        number += 1
newlines = []
for i in range(0,number): # record new lines
    newlines.append([[rho_new[i],theta_new[i]]])
for i in range(len(newlines)):
        for rho,theta in newlines[i]:
            a = np.cos(theta)
            b = np.sin(theta)
            x0 = a*rho
            y0 = b*rho
            x1 = int(x0 + 2000*(-b))
            y1 = int(y0 + 2000*(a))
            x2 = int(x0 - 2000*(-b))
            y2 = int(y0 - 2000*(a))

        cv.line(result3,(x1,y1),(x2,y2),(0,0,255),1) # draw new lines on result3
cv.imwrite("room-combineline.jpg",result3)
print("Searching intersect points...")
segmented = intersect.segment_by_angle_kmeans(newlines)
intersections = intersect.segmented_intersections(segmented) # find intersections
for i in range(len(newlines)):
        for rho,theta in newlines[i]:
            a = np.cos(theta)
            b = np.sin(theta)
            x0 = a*rho
            y0 = b*rho
            x1 = int(x0 + 2000*(-b))
            y1 = int(y0 + 2000*(a))
            x2 = int(x0 - 2000*(-b))
            y2 = int(y0 - 2000*(a))

        cv.line(result4,(x1,y1),(x2,y2),(255,255,255),1)
for center in intersections:
    cv.circle(result4, center, 2, (0, 0, 255), 1) # draw new lines and intersections on result4
cv.imwrite("room-intersect.jpg",result4)
print("Cutting lines...")
```

```
points = cutLen(6, newlines, edges, size[1], size[0], 0, 20) # cut lines into line segments, (kernel size, lines, image,
numoflen=0                                                    # xhi, yhi, threshold1, threshold2)
start = [] # start points of line segments
end = [] # end points of line segments
for i,p in enumerate(points):
  if i%2==0:
      start.append(p)
  else:
      end.append(p)
      numoflen += 1
for i in range(0,numoflen):
  cv.line(result5, start[i], end[i], (0, 0, 255), 2) # draw line segments on result5
  #cv.line(edges, start[i], end[i], (122), 1)
#for center in intersections:
#     cv.circle(result1, center, 2, (0, 0, 255), 1)
for i in range(len(newlines)):
      for rho,theta in newlines[i]:
          a = np.cos(theta)
          b = np.sin(theta)
          x0 = a*rho
          y0 = b*rho
          x1 = int(x0 + 2000*(-b))
          y1 = int(y0 + 2000*(a))
          x2 = int(x0 - 2000*(-b))
          y2 = int(y0 - 2000*(a))

      cv.line(edges,(x1,y1),(x2,y2),(122,122,122),1) # draw new lines on edge image
#cv.imshow("Result", result1)
cv.imshow('Edge',edges)
#cv.imwrite("finaledge.jpg",edges)
#cv.imwrite(output, result5)
cv.waitKey(0)
cv.destroyAllWindows()
```