Vanishing Point

Image Processing - OpenCV, Python & C++

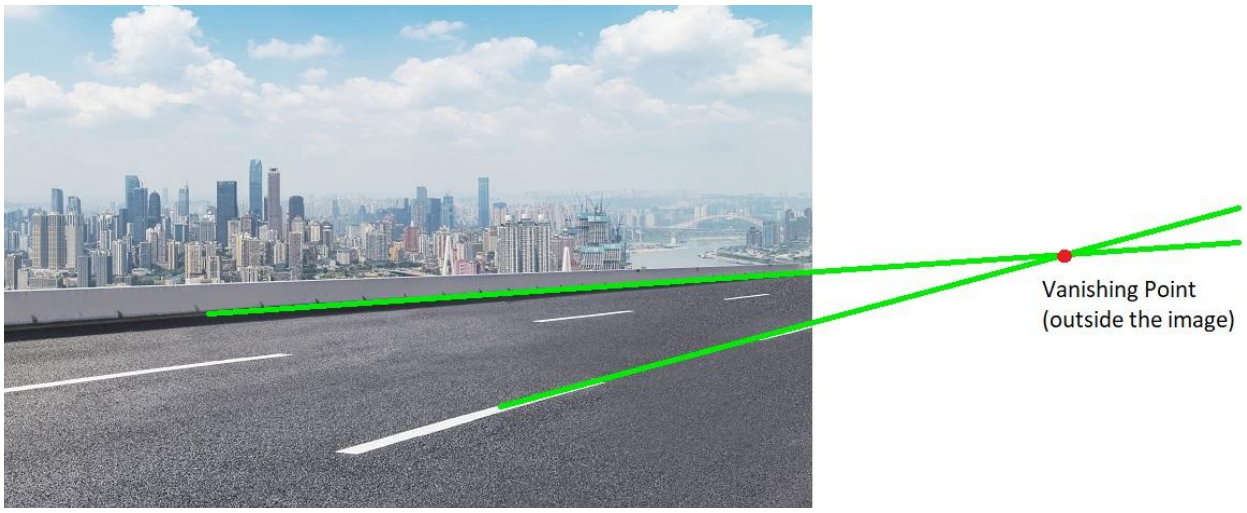
Table of Contents

Overview

“A **vanishing point** is a point on the image plane of a perspective drawing where the two-dimensional perspective projections (or drawings) of mutually parallel lines in three-dimensional space appear to converge.”

This is a typical definition of the vanishing point from Wikipedia. Basically, it says that everything in an image appears to converge at a point, this point is known as the vanishing point. As shown in the above image, the “red dot” at the center of the right image is the vanishing point of the image.

Also, note that it is not necessary that the vanishing point is always inside the image. The objects of the image may appear to converge at a point outside the image as shown below. Also, there can be more than one vanishing point in an image but here, we are only concerned with an image with one vanishing point.



Problem Statement

Given an image with the vanishing point inside it, the aim of this project is to build up a generalized code that can find the vanishing point in the image.

Note that this code can also be implemented for images with the vanishing point outside the frame but here we will only take the images with vanishing point inside the frame as it is easier to visualize. Also, this code is built only for images with one vanishing point.

Approach

To solve this problem, we will implement the following easy steps:

- First, we will find all the lines present in the image. These lines should be at least a few pixels long.
- Secondly, we will filter these found lines. Filtering will be done wrt the line's angle wrt the horizontal and their length. The explanation is provided in the respective section.
- Finally, we will find the vanishing point of the image with the help of the lines found in the above 2 steps. It is to note that the vanishing point is approximately the point of intersection of these lines.

Setting up initial code

First of all, we will have to make a basic code that imports all the required libraries and read the input image(s), and then pass each image for processing. In this section, we will see how to do it.

First of all, create a new directory for this project. I am naming it as **“VanishingPoint”** and then create a file in it that will contain our source code. I am naming this file as **“main.py”**. Also, add a folder in this directory containing all of our input images in which we will have to find the vanishing point. I have added the input images in the folder named **“InputImages”**.

Now in the main.py file, let’s write the code that will read the input images and pass them one by one for further processing.

(I personally recommend you to learn this code and save it for further use. This code is generalized and can be used in almost all of the projects where you will have to work with many images.)

Let us first start by importing the libraries required.

```
import os
import cv2
import math
import numpy as np
```

Here I have imported the **os**, **cv2**, **numpy**, and **math** libraries. The **os** library will be used in the code section which reads the images. The **math** library will be used for implementing simple mathematical operations while finding the vanishing point. The **cv2** and **numpy** libraries will be used throughout the project.

Now let’s build up the code that will read the input images from the folder and pass them for further processing.

The following code contains a function that will take the path of the image or the folder containing the input images and return a list of images and a list of names of the images read in the same order. The names of the images can be used when you have to store the final output images to avoid confusion.

The code is explained by the comments.

```
def ReadImage(InputImagePath):
    Images = []             # Input Images will be stored in this list.
    ImageNames = []         # Names of input images

    # Checking if the path is of file or folder.
    if os.path.isfile(InputImagePath):                  # If path is of file.
        InputImage = cv2.imread(InputImagePath)         # Reading the image.

        # Checking if image is read.
        if InputImage is None:
            print("Image not read. Provide a correct path")
            exit()

        # Storing the image and its name. Name of the image is same as it
        # is stored in the folder
        Images.append(InputImage)
        ImageNames.append(os.path.basename(InputImagePath))

    # If the path is of a folder containing images.
    # NOTE: The folder must contain only images.
    elif os.path.isdir(InputImagePath):
        # Getting all image's name present inside the folder.
        for ImageName in os.listdir(InputImagePath):
            # Reading images one by one.
            InputImage = cv2.imread(InputImagePath + "/" + ImageName)

            # Storing the image and its name.
            Images.append(InputImage)
            ImageNames.append(ImageName)

    # If it is neither file nor folder(Invalid Path).
    else:
        print("\nEnter valid Image Path.\n")
        exit()

    return Images, ImageNames
```

Now let's call this function and loop over the images and pass them for finding the vanishing point.

```
if __name__ == "__main__":
    # Reading all input images
    Images, ImageNames = ReadImage("InputImages")

    # Iterating over the input images and passing them for further processing
    for i in range(len(Images)):
        Image = Images[i]            # Reading the current image

        # Call other functions for processing here in this loop one by one
```

Now with the help of the above code, we are ready to build up the code for finding the vanishing point in the image. Let's see how it is done.

Get Lines in the image

In this section, we will discuss how can we find the straight lines present in the image that on extending will approximately merge at the vanishing point.

Here we will find the lines using the **Probabilistic Hough Lines Transform** technique (function: **HoughLinesP, OpenCV** library).

Let us see how to do it.

First, we will have to find the edge image of the colored input image.

To do so, we will convert the colored input image to grayscale using the function **cvtColor** of the OpenCV library.

Then we will blur the image using the function **GaussianBlur** of the OpenCV library to remove unnoticeable noise in the image which leads to irregular edges that are not required and may lead to false positives. Meaning that these noises may give us unuseful lines.

Then we will find the edge image using the function **Canny** of the OpenCV library. This edge image will be used to find the lines later.

```
# Converting to grayscale
GrayImage = cv2.cvtColor(Image, cv2.COLOR_BGR2GRAY)
# Blurring image to reduce noise.
BlurGrayImage = cv2.GaussianBlur(GrayImage, (5, 5), 1)
# Generating Edge image
EdgeImage = cv2.Canny(BlurGrayImage, 40, 255)
```

Below is demonstrated the transformation of the input image to the edge image using the code discussed above.

Input Coloured Image

Grayscale Image of the Input Image

Blurred Gray Image

Edge Image

Now we will find the lines in the edge image using the function **HoughLinesP** of the OpenCV library and exit if no lines are found.

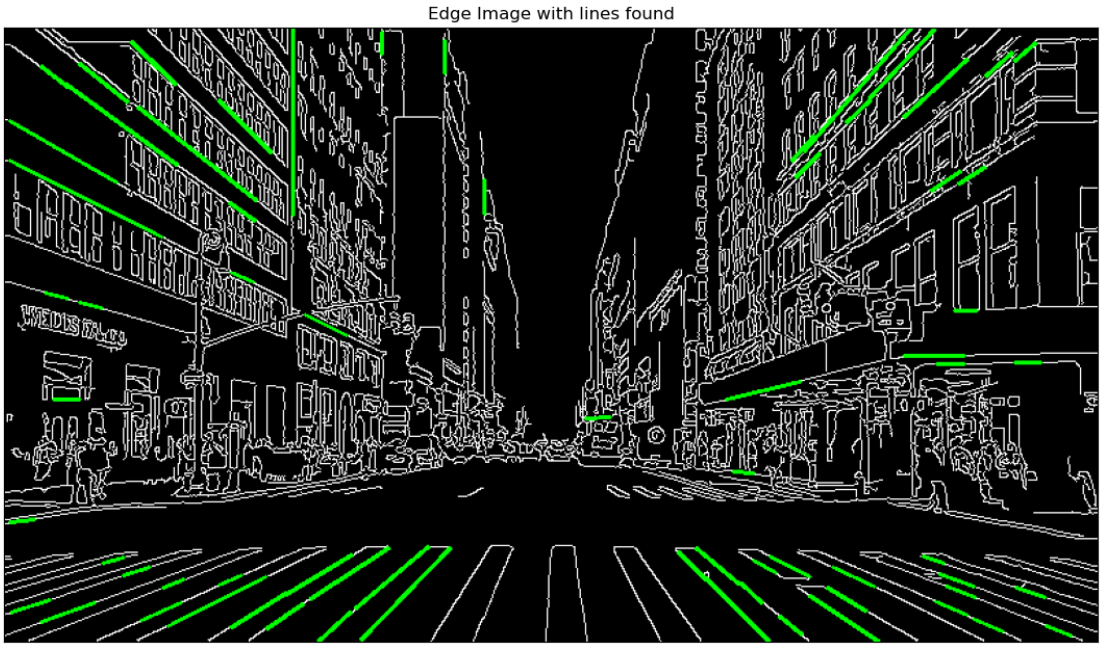
```
# Finding Lines in the image
Lines = cv2.HoughLinesP(EdgeImage, 1, np.pi / 180, 50, 10, 15)

# Exit if no lines are found
if Lines is None:
    print("Not enough lines found in the image for Vanishing Point
           detection.")
    exit(0)
```

As you can see here, we are passing various parameters to the function for finding the lines, where, the first parameter is the edge image. The parameters passed are explained briefly below.

- **Edge Image** - This is the edge image found earlier. It should be a binary image(1 channel).
- **rho** - The resolution parameter ***r*** in pixels. We use 1 pixel.
- **theta** - The resolution parameter theta in radians. We use 1 degree(np.pi/180).
- **threshold** - The minimum number of intersections to detect a line.
- **minLinLength** - The minimum number of points that can form a line. Lines with less than this number of points are disregarded.

- **maxLineGap** - The maximum gap between two points to be considered in the same line.

Below is the edge image with the lines found using the **HoughLinesP** shown in green.



As you can see in the above image, few of the lines in the image are found most of which on extending will possibly converge at the vanishing point of the image. Note that on playing with the parameters passed to the function, you can get different results with more or less line however the current values of the parameters are obtained by hit and trial method and works just fine for almost all the images.

Now we are concerned to filter out these lines to get the most appropriate lines for finding the vanishing point. Let us see how can we approach this problem in the next section.

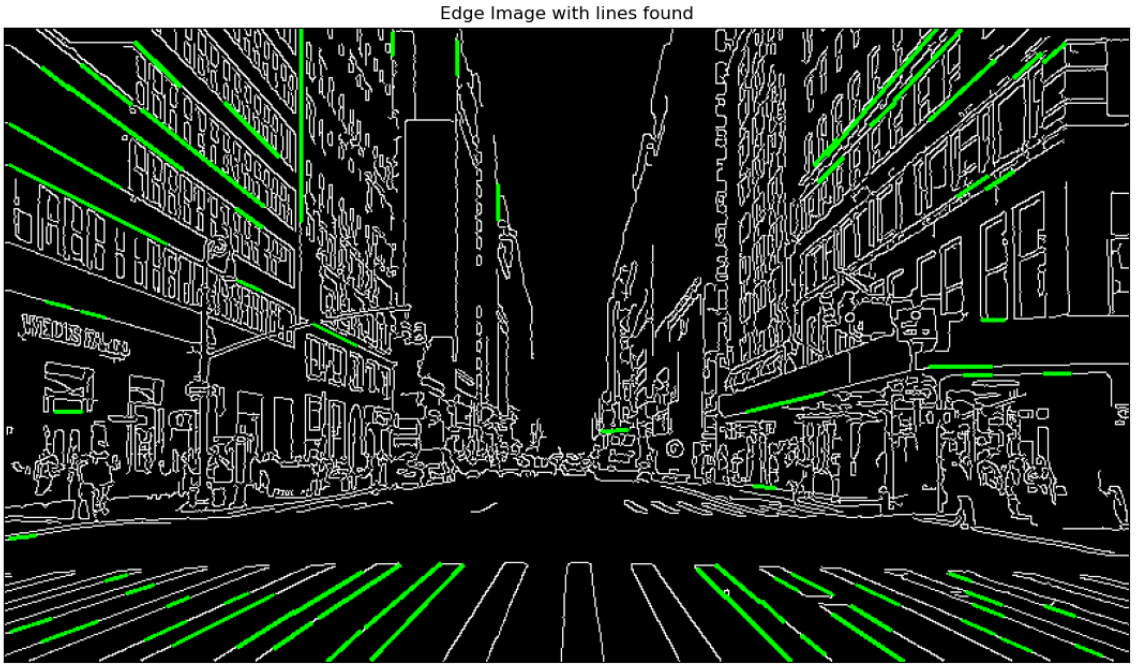
Filter out Lines

In this section, we will see how can we filter the lines found earlier to get the most appropriate lines for the detection of the vanishing point.

There are many possible approaches you can think of but here we will filter these lines wrt their angle and length. Let's see how can this be done.

Filter lines wrt their angle

Below is our edge image found earlier and the lines found drawn on it with green.



As you can notice on the upper of this image, there are a few vertical lines detected on the edges of the buildings. These vertical lines on extending will not cross from near the vanishing point of the image which is near the center. These lines are obtained generally because of the objects kept in the scene or any vertical thing just like buildings in this case. Thus, we will have to remove such vertical lines from our calculation.

Also, there will be cases when we will get horizontal lines because of similar reasons. We will have to remove them also. Luckily, in this case, the HoughLinesP function didn't give us any horizontal lines.

To remove these horizontal lines, we can first calculate the equation of each of these lines in the format **y = mx + c**. Then using the slope **m**, we can find the angle of these lines wrt the horizontal, and using this we can reject the horizontal and vertical lines.

Let's have a look at the piece of code which will do this for us.

Before moving ahead, it is important to note that the **HoughLinesP** function returns the coordinates of the endpoints of the lines in the format:

[[[x11, y11, x12, y12]],
[[x21, y21, x22, y22]],
[[x31, y31, x32, y32]],
…….]

Where, (xi1, yi1), and (xi2, yi2) are the x and y coordinates of the endpoints of the i^{th} line.

```
REJECT_DEGREE_TH = 4.0
FilteredLines = []                  # Filtered lines will be stored here
for Line in Lines:                  # Iterating over each line
    [[x1, y1, x2, y2]] = Line       # Getting the coordinates of the line

    # Calculating equation of the line: y = mx + c
    # if x1 != x2, slope can be found using regular equation
    if x1 != x2:
        m = (y2 - y1) / (x2 - x1)
    # if x1 = x2, slope is infinity, thus a large value
    else:
        m = 100000000
    c = y2 - m*x2         # c = y - mx from the equation of line
    # theta will contain values between -90 -> +90.
    theta = math.degrees(math.atan(m))

    # Storing lines of slope not near to 0 degree or +-90 degree
    if REJECT_DEGREE_TH <= abs(theta) <= (90 - REJECT_DEGREE_TH):
        # length of the line
        l = math.sqrt( (y2 - y1)**2 + (x2 - x1)**2 )
        FilteredLines.append([x1, y1, x2, y2, m, c, l])
```

Here, after finding the constant **m** and **c**, we are finding the angle **theta** of the line wrt the horizontal using the **atan** (inverse of tan) function of the **math** library. This function returns the value of the angle in radians so we are converting radians to degrees using the function **degrees** of the **math** library. This will return the value of **theta** in degrees between **[-90, +90]**.

Now we are checking the angle **theta** and storing only the lines that are not vertical or horizontal i.e., theta is not close to 0 degrees or +- 90 degrees. Also, we have introduced a threshold value **REJECT_DEGREE_TH** for the value of **theta**. This is taken into consideration because, in image processing, coordinates of a point can only have discrete values thus we can probably have the value of theta close to 0 for a horizontal line. Thus, the values of theta for the acceptable lines can be in the ranges **[-86, -4] and [4, 86]**, where **REJECT_DEGREE_TH = 4.0**.

NOTE: We are storing the data of filtered lines at the end in the format:

[[x1, y1, x2, y2, m, c, l], ...]

Where, **(x1, y1, x2, y2)** have usual meaning, **(m, c)** are the constants slope and c of the equation of the line **y = mx + c**, and **l** is the length of the line in pixels between the two endpoints. This is found by the usual **Euclidean Distance** formula. These values are stored in such a way because they will be used in the later stages of the project.

Now let us see the second approach on how to filter the lines.

Filter lines wrt their length

In Image Processing, it is always advised to not take small lines into consideration of processing as even if they seem to point towards the correct direction, on extending, they might cross from near the required point. So here, we will reject the small lines from our further processing stages.

Now a question might come into your mind that what should be the minimum length of a line to be considered it as a correct line for our further computation. This is a valid question and the answer to this depends upon the problem statement and the input at hand however in this project, we will solve this problem with another approach due to reasons explained below.

The first reason being that by eliminating the lines that are of less than a certain length, it is a possibility that we may lose all of our current lines detected as the size of the input image itself may also vary. Whereas, it is a necessity for us to have atleast a few lines for computation of the vanishing point.

The second reason for our approach to this problem is that there is also a possibility that we may have hundreds of lines after filtering of lines (wrt minimum length logic). These hundreds of lines are not necessary as the vanishing point can be found accurately using a few lines only. These extra lines will only contribute towards the computational expense of our code and may make our code slow.

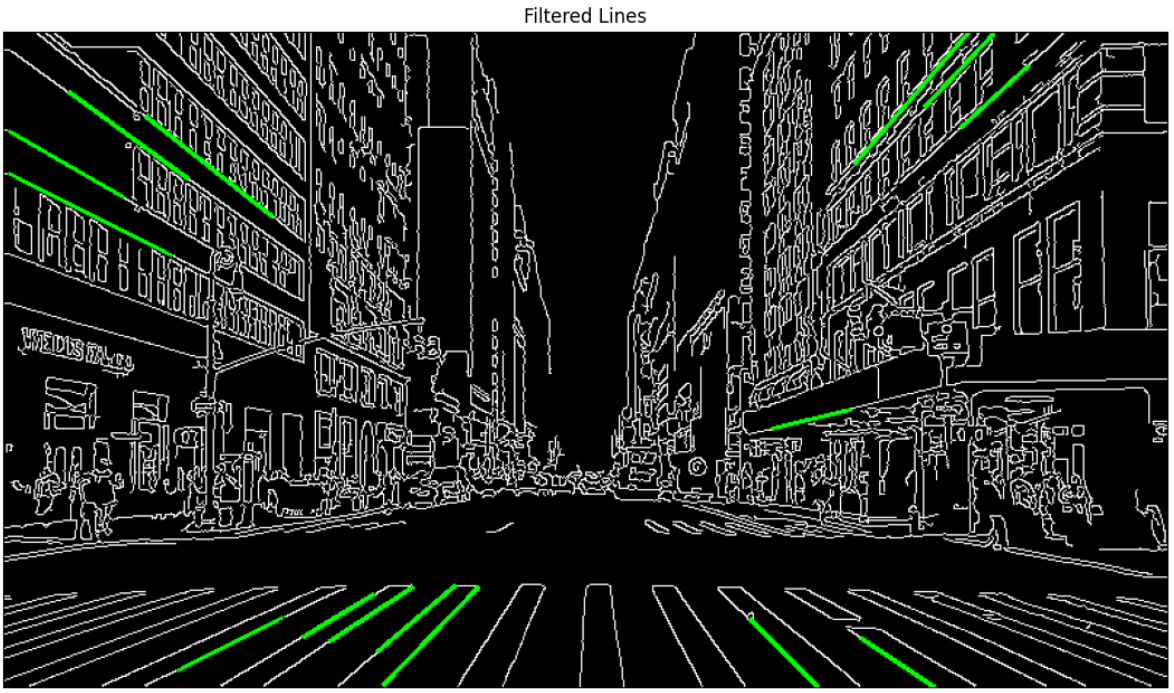
So, in our approach, we will take only the **15 longest lines** from all the lines obtained after filtering them in the previous step (only if we have more than 15 lines, else if less than 15 lines, we will take them all). The number **"15"** is just a parameter that I consider correct for our computation of vanishing point however you can change it according to your wish.

For obtaining the longest lines, we are first sorting the lines wrt their lengths using the **sorted** function, and then we are slicing out the first 15 lines (longest 15 lines).

Let's see how this can be done.

```
if len(FilteredLines) > 15:    # if more than 15 lines are there
    # Sorting lines wrt their length in decreasing order
    FilteredLines = sorted(FilteredLines, key=lambda x: x[-1], reverse=True)
    # Taking only the longest 15 lines
    FilteredLines = FilteredLines[:15]
```

The image below shows the best 15 lines filtered using the above filtering procedures.



Now let's have a look at the final section of our project where we will calculate the vanishing point using these lines.

Calculate Vanishing Point

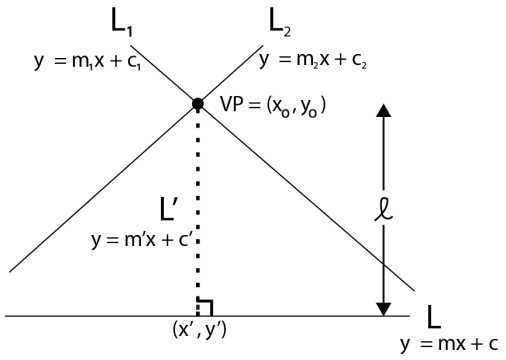
Kudos!!! You have reached the final section of this project where we will calculate and find the vanishing point of the image using the lines found earlier.

This section involves basic mathematics related to lines (their intersection, and their distance from a point) and a small logic. Let's get started.

What we will do here is that we will take a combination of 2 lines and then find their intersection point. This intersection point can be the vanishing point as it is expected that the lines converge near the vanishing point. But a problem here is that there are many lines (max 15 for us), so the number of such intersections of two lines can result in many distinct points. So how do we find the point among these points that is possibly the closest to the actual vanishing point?

To do so, we introduce a factor of **"error"** in our logic. The error is basically the **sq root of the sum of squares of the distance of this point from each line**. The point corresponding to the minimum error value will possibly be the closest to the vanishing point and hence our **Vanishing Point**.

Below is the image demonstrating the calculation behind the intersection of 2 lines and that points error value. Here we are taking only 3 lines as an example.



$$\ell = \sqrt{(y'-y_o)^2 + (x'-x_o)^2}$$

$$\text{Error} = \sqrt{\ell^2 + \_\_\_}$$

$$x_o = \left\{ \frac{c_1 - c_2}{m_2 - m_1} \right\}$$

$$y_o = m_1 x_o + c_1$$

$$m' = \frac{-1}{m} \quad \text{[L \& L' are perpendicular]}$$

$$c' = y_o - m' x_o \quad \text{[As } (x_o, y_o) \text{ is a point on L']}$$

$$x' = \left\{ \frac{c - c'}{m' - m} \right\}$$

$$y' = m' x' + c'$$

Here we are taking lines **L1** and **L2** whose intersection point is **(x0, y0)**. The equation of line **L1** is **y = m1*x + c1** and the equation of line **L2** is **y = m2*x +c2**. We will now find the distance of this point from the line **L** whose equation is **y = m*x + c**. To do so, we make a line **L_ (L dash)** (equation: **y = m_*x + c_**) perpendicular to **L** and passing through the point

(x0, y0). The point of intersection of line **L_** and **L** is **(x_, y_)**, thus, the distance of the point **(x0, y0)** from line **L** is **l = distance between (x0, y0) and (x_, y_)**.

The value of **Error** for such points wrt the line **L** is the root of the sum of squares of such **l** as shown in the image above.

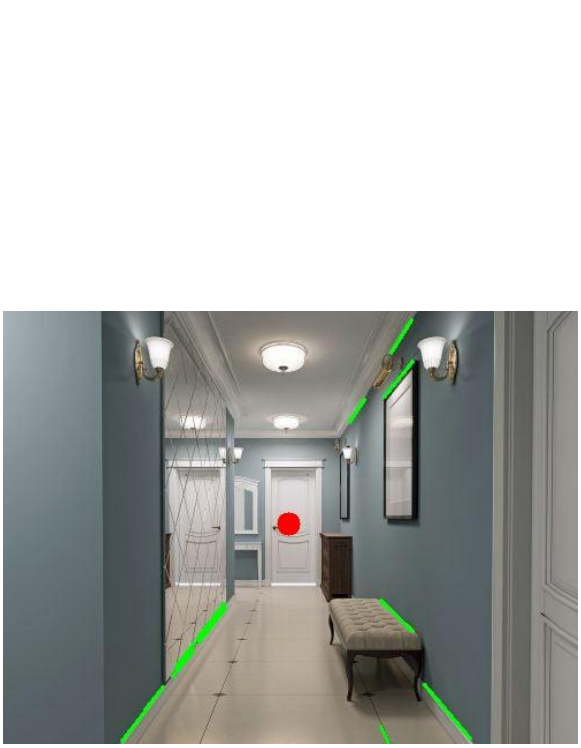
Let's have a look at the code now. The code is explained with comments.

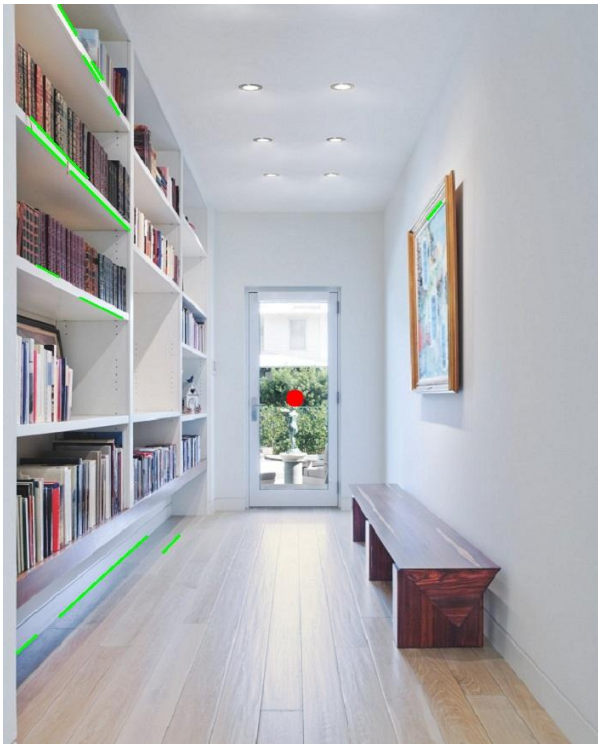
```
# Initializing variables
VanishingPoint = None       # Coordinates of the vanishing point
MinError = 100000000000     # Minimum error found (initially large value)

for i in range(len(Lines)):    # Iterating over lines and taking 2 at once
    for j in range(i+1, len(Lines)):
        # Reading m and c values of the line
        m1, c1 = Lines[i][4], Lines[i][5]
        # Reading m and c values of the line
        m2, c2 = Lines[j][4], Lines[j][5]

        # If lines are not parallel
        if m1 != m2:
            # Finding (x0, y0)
            x0 = (c1 - c2) / (m2 - m1)
            y0 = m1 * x0 + c1

            err = 0            # Error of this point
            # Iterating over all lines for error calculation
            for k in range(len(Lines)):
                # Reading m and c value of the line L
                m, c = Lines[k][4], Lines[k][5]
                # Calculation m and c of L_
                m_ = (-1 / m)
                c_ = y0 - m_ * x0

                # Calculating (x_, y_) - point of intersection of L and L_
                x_ = (c - c_) / (m_ - m)
                y_ = m_ * x_ + c_

                # Calculation distance between (x0, y0) and (x_, y_)
                l = math.sqrt((y_ - y0)**2 + (x_ - x0)**2)

                err += l**2          # Adding to error value
```

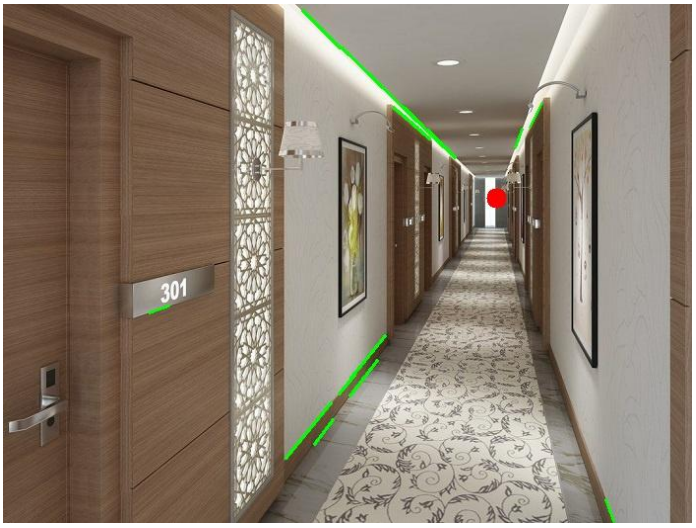
```
        err = math.sqrt(err)      # Finally taking sq root of error

        # Comparing with minimum error value till now
        # If present error value is lesser, updating minimum error value
        # and vanishing point coordinates.
        if MinError > err:
            MinError = err
            VanishingPoint = [x0, y0]
```

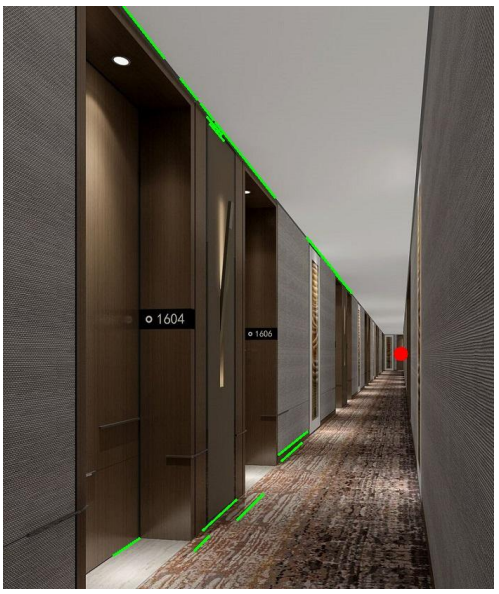
Finally, we will get the value of the coordinate of the Vanishing Point, and thus, we have achieved our goal!!!!!!

Let’s have a look at the output of this code on different images.






301


1604
1606