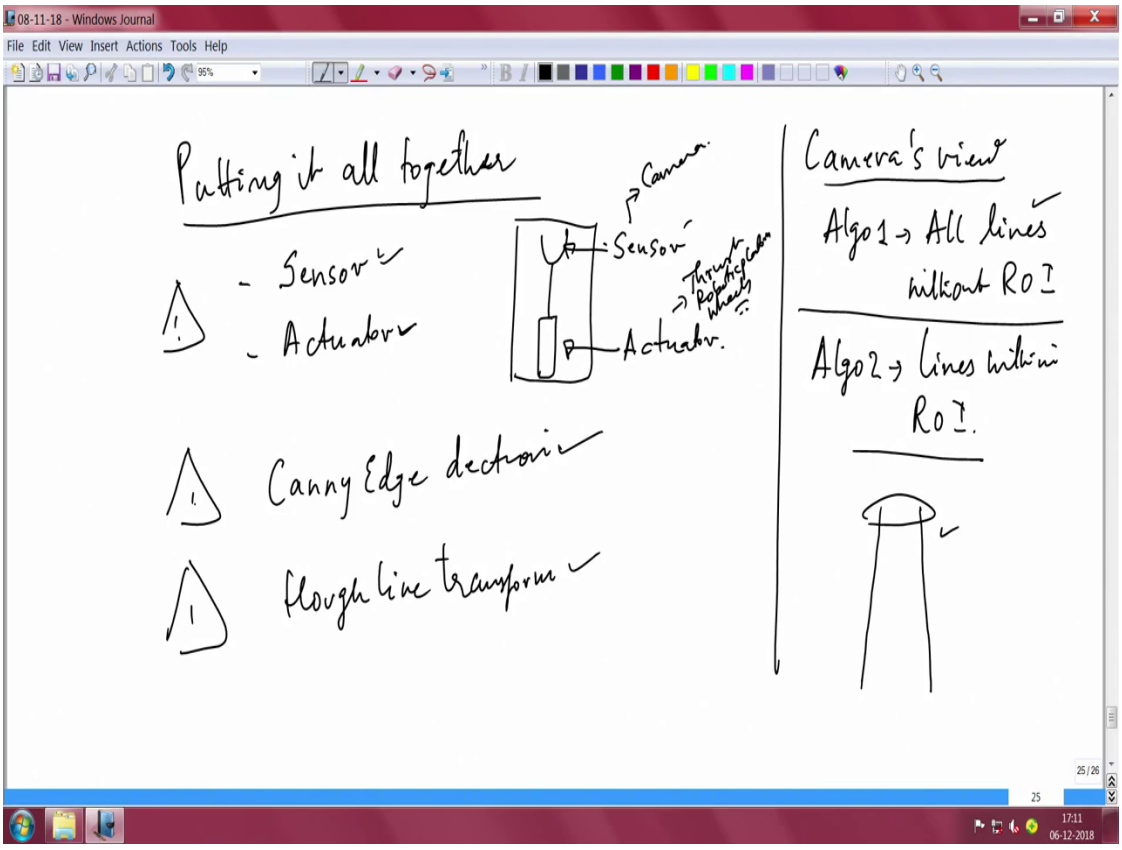
Advanced IOT Applications
Dr. T V Prabhakar
Department of Electronic Systems Engineering
Indian Institute of Science, Bangalore

Lecture – 14
Code walkthrough of computer vision algorithm

Let us see first, this whole course, we talked about Braitenberg experiment, computer vision and all of that right? Let me show you this picture, here we are trying to put everything together.

(Refer Slide Time: 00:44)



In the demonstration you have seen a sensor and the actuator. The Braitenberg experiment actually spoke about sensor and actuator in its simplest form how they are coupled. That is exactly what the Kobuki system actually had.

In our case the sensor was the camera and actuator was the thrust we provided to the robotic platform particularly the wheels right, the thrust is given to the wheels. And, that thrust we will look in little more detail. What all you are supposed to do? You suppose to

have a sensor, you are supposed to have an actuator, you are expected to do canny edge detection, and then you are supposed to do Hough line transform. And all of this should be in a possible demonstrable form which is exactly what that basic demonstration was doing.

By no means this is anywhere close to what reality is, but never the less it allows you to think deeply about the possibilities and how the algorithms can actually work.

For that we will first tell you how complex this whole system can be; to give you a view of that complexity, it is important to know what is the cameras view and what exactly is it seeing right. So, let us look at 2 algorithms; first algorithm is about several lines where I do not do any preprocessing of the image with respect to you know cutting of the region of interest, I do not do anything. I just say I will call Hough line, Houghline P I will call where I will give a certain specification I will come to that as we show you the source code. Then it will return a set of lines, it's business is to return a set of lines, which adhere to a certain length and to a certain width; all lines in the image it is suppose to return, all possible lines, if you specify what you want in terms of length and width of the line it will display everything.

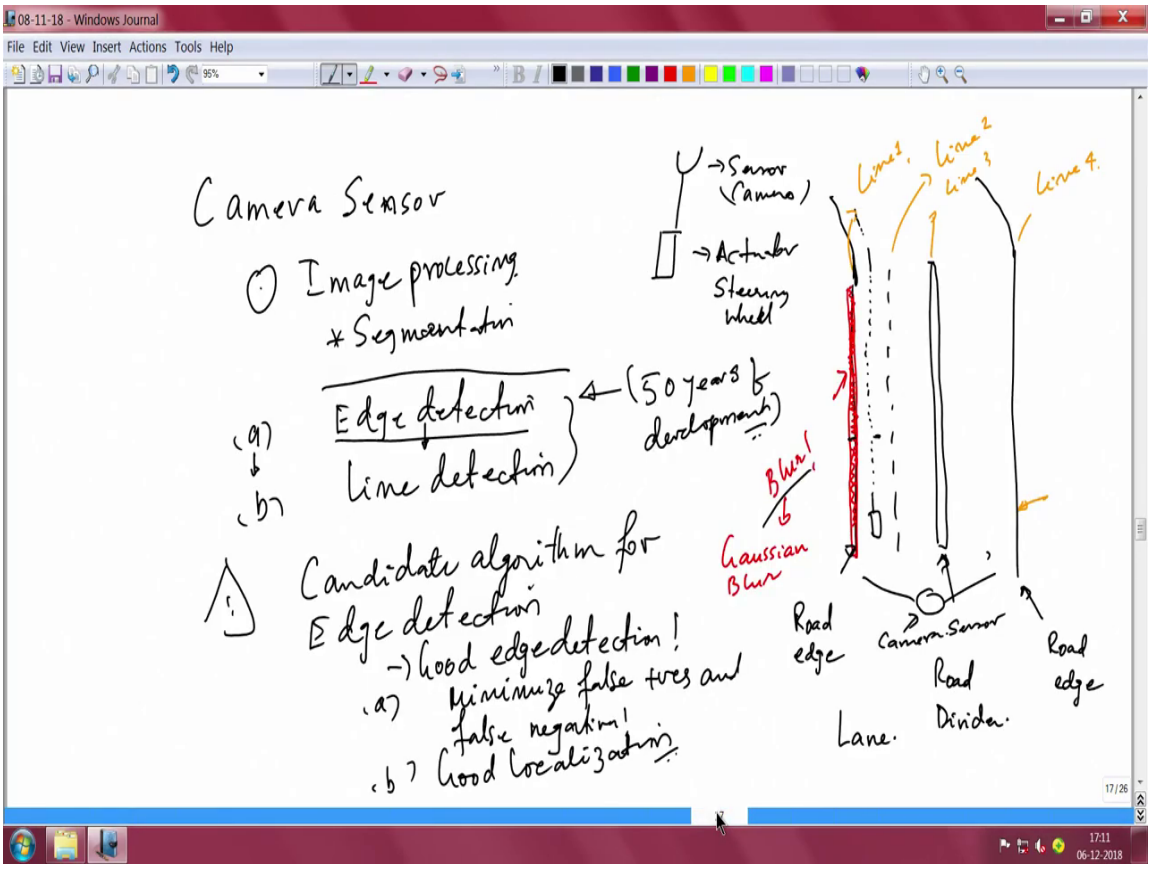
It is now your algorithm, which you will have to take care to ensure that the platform is moving exactly on those 2 lines of interest. So, this is the most important thing. So, we will see what is the cameras view. The second thing is the algorithm 2 where you do some processing on the image, you take care of region of interest, and you just then say that only these 2 lines which are within the region of interest 2 or more; you do not even know that by the way. It is just that you expect that if you are looking at a region of interest and you expect that there are 2 prominent lines, which you want to sort of look as edges and you want to use the Hough line transform in order to make your system to move across it .

So, now why is this an important thing? there are many nice things, which you can exploit in a camera kind of a sensor. The following holds true, if a camera sees a line in front of it, it continues to see the line tapering as it goes along. So, almost it starts of by being straight here and as the view of the camera extends the same 2 lines appear it to be a little bit tilted towards each other. It is also true for sometimes even for our own eyes,

if you are seeing a long distance you will see the 2 lines are converging at some point. So, this also the same view that the camera is looking at.

You can think in a simplistic manner, because most of our demonstrations in part of this module are very simple just to excite you. So, we will take these simple things and then try to exploit around it to make a nice working demonstration that was the real goal. So, I will exploit this conditions. So, this is a very important point. I will show you how we can build on this particular aspect, but before we go and actually see anything about the cameras view it was not far long back that I had shown you the below picture in right.

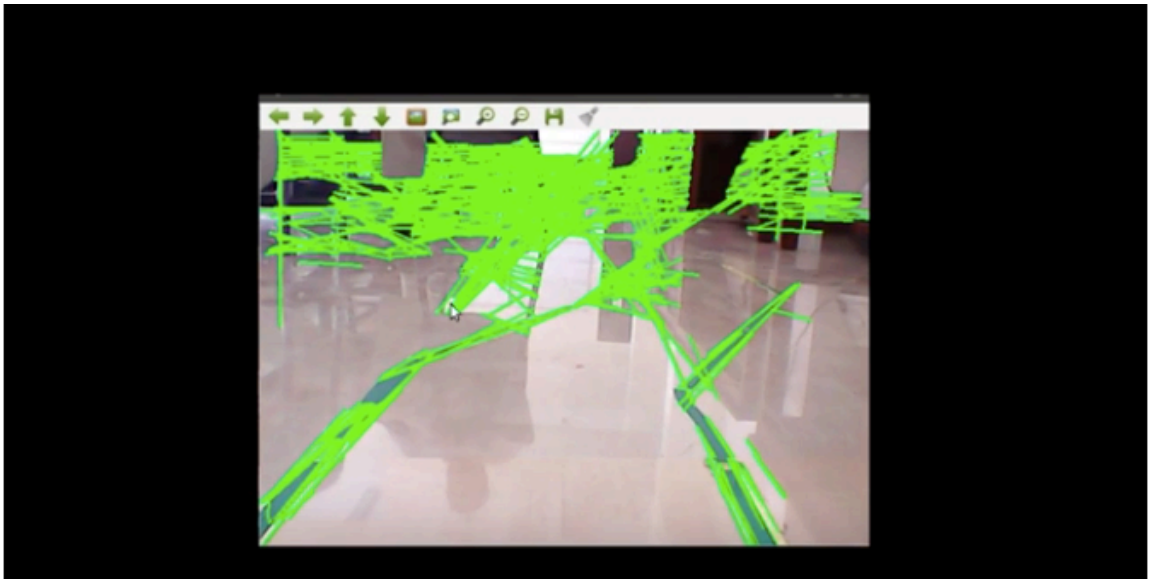
(Refer Slide Time: 06:05)



You had the road divider, you had the road edge on the right side, you had the road edge on the left side, and we included when we were talking about the braitenberg experiment the sensor is a camera and actuator is steering wheel.

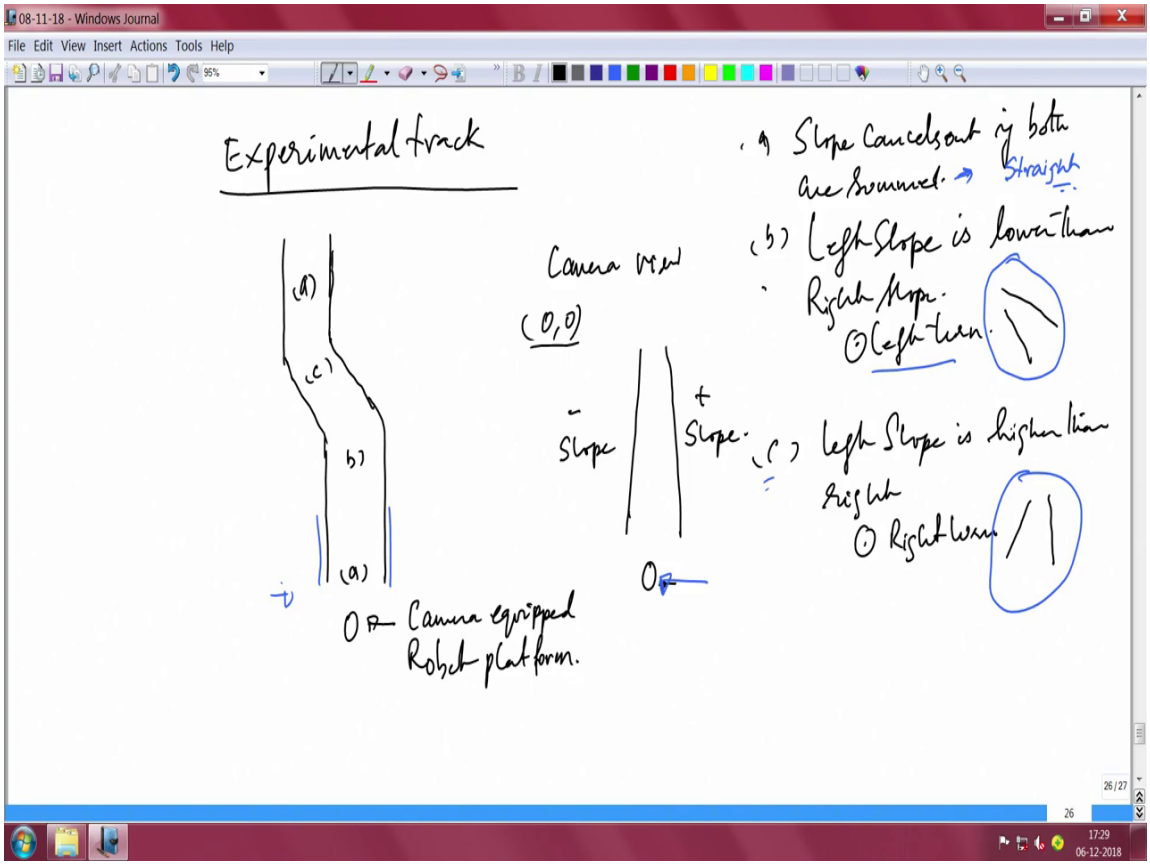
So, kindly note, that this is exactly what we were trying to achieve, when we try to do this demonstration as part of this particular module. So, now, let us turn our attention to what the camera is actually seeing with respect to algorithm 1.

(Refer Slide Time: 06:49)



So, now, let us see what is the cameras view for algorithm 1. It is quite obvious where you can see that there are 2 prominent lens left line and the right line and also there are a number of points above, a big mask of them. You can see that the whole system, you see that mass of lines above in the picture, these are all what the Hough line P actually tells you because you have specified a certain length, and a certain width and all of them are being pointed out. In addition to these 2 prominent edges, that you are looking for and we on which actually you wanted the robotic platform to move.

(Refer Slide Time: 08:30)



The experimental track shown in above picture, that you saw was more or less this. And, I have divided this complete experimental track into 3 sections; one section I call A, second one is B, and C is the third section. I am exploiting the camera view. The robotic platform with the camera equipped is pointed below trapezoidal lies in picture. And, as you can see this trapezoidal kind of picture where the 2 slopes, 2 lines are matching is actually being exploited nicely here.

Basically the left line will have a negative slope, the right one will have a positive slope, now map back into region A, what should happen it region A? At region A you can see a straight line, and 2 edges. So, what will happen? You go on summing the slopes of left side and right side continuously keep summing them, and you will find that each time you sum them you will get a 0. If, you keep seeing 0 or a small value close to 0, you keep declaring it as straight and keeps going.

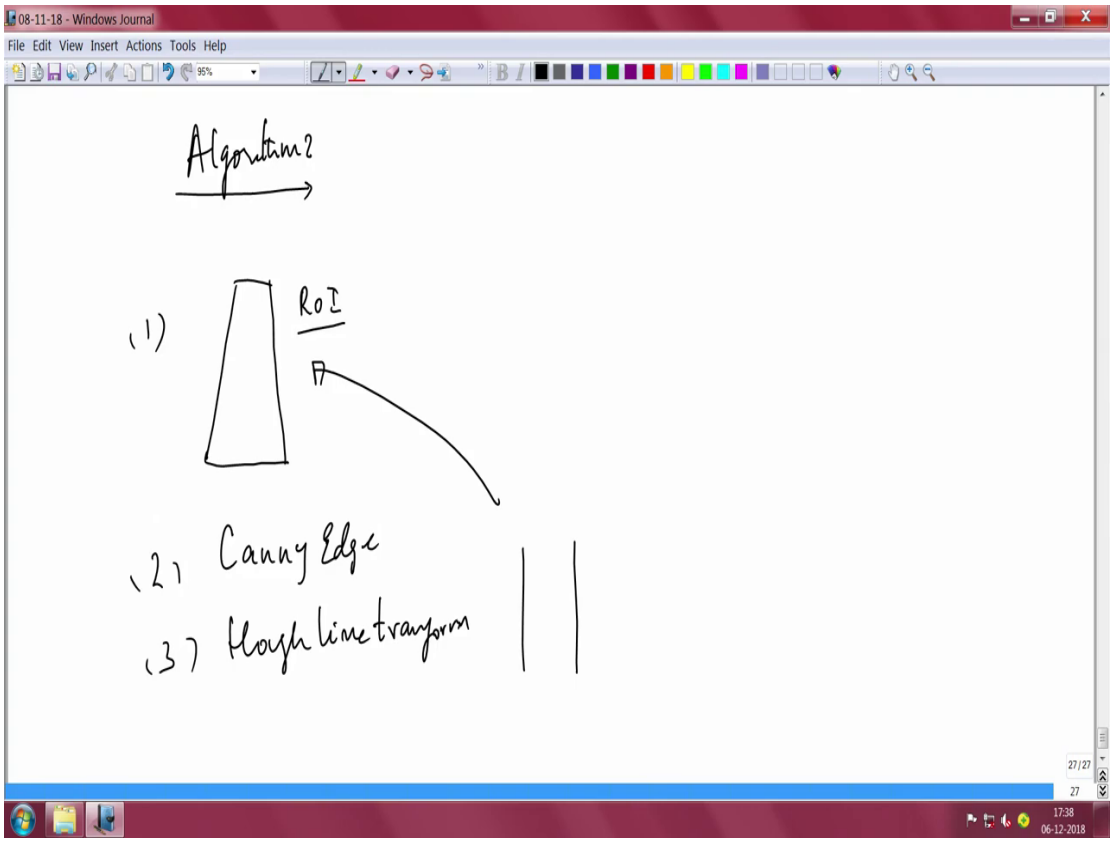
Now, when the platform comes to B, you find that the left slope is lower than the right slope.

You will see the left slope lower than the right slope, and what you will do now? You will actually end up with the left turn. You will give thrust to the left and the right side motor, when you give a thrust to the right side motor automatically the system will turn the whole platform to the left side. So, it is a bad algorithm, but I am sure you are following the Braitenberg experiment, which is being followed as much as possible and you keep working on it and keep improving it.

Similarly, you come to when the robotic platform comes to C the left slope is higher than the right slope. And, you take a right turn and the line will orient towards right, with that you will make a right turn and again you will go back to a because all these region is indeed right.

In algorithm 2 what is the cameras view? First of all we said algorithm 2 should be something that removes all the lines beyond the region of interest. If, you have to do that you have to cleverly get rid of all the lines, you should not even do canny edge for those which are beyond the region of interest. So, your region of interest should be declared as a first step.

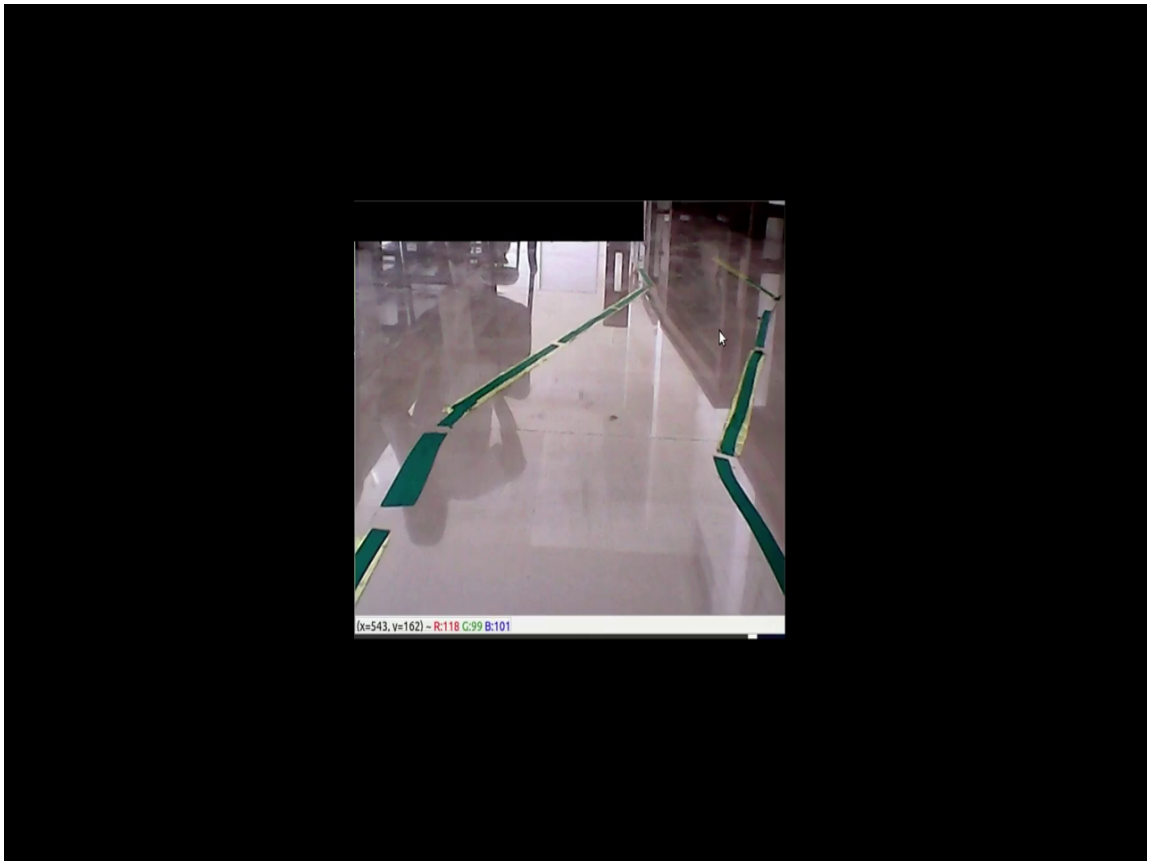
(Refer Slide Time: 12:37)



As show above there is your region of interest, then you feed that region of interest into canny edge and tell canny that the boundary in which you should detect the 2 prominent left edge and the right edge, edge detection should be there.

Then you are telling a Hough transform, canny edge will detect all edges as I said and Hough line will ensure that only required lines are detected. So, because an image can have all types of edges. You have to run the Houghline transform to make sure that you are actually interested in the exactly the 2 lines, which are within the region of interest. So, it should be within the region of interest. So, this is what we have done in order to show you the improved algorithm, which is a very small say the first algorithm is 1.0 this is 1.1. Let us go and look at 1.1 algorithm and understand how the cameras view of this particular video looks like. Let me draw your attention to the screen below.

(Refer Slide Time: 13:57)



So, you can see how beautiful the system really looks, it just has these 2 prominent edges on right and on the left. Now, the big take away from this slide is, this is giving you

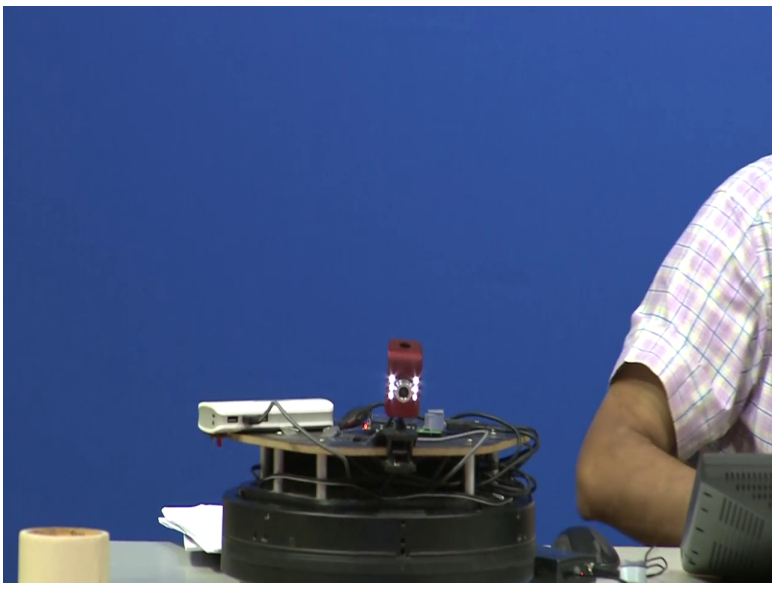
those 2 very prominent edges and let us see whether the platform actually follows this to left and right edge as you know as accurately as possible. So, let us run this video.

Refer - https://www.youtube.com/watch?v=_eUPGN8jGdc 14:00.

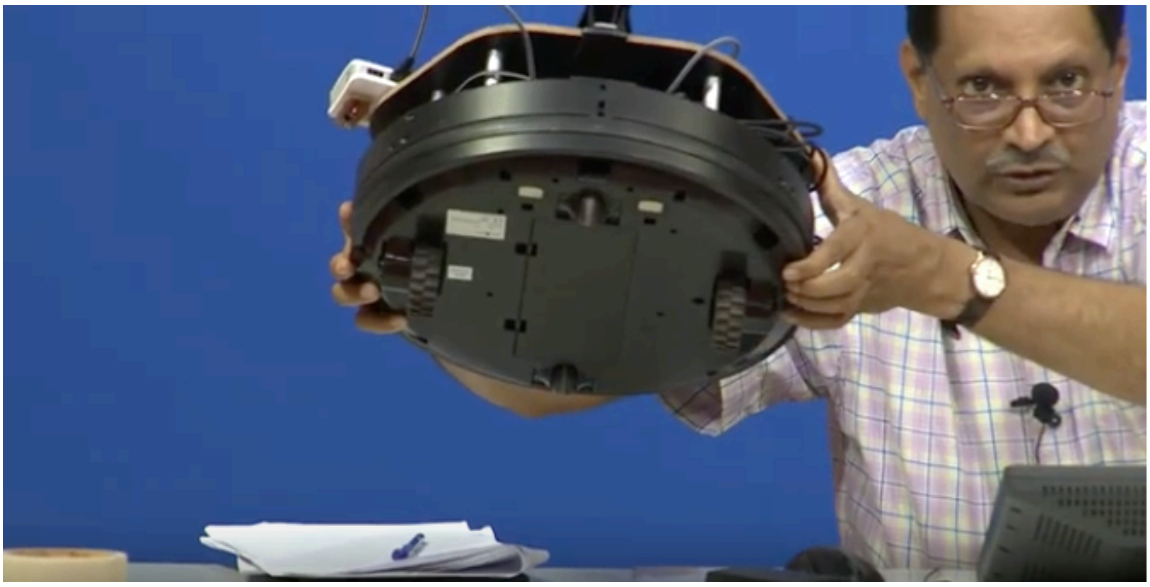
So, you can see that these algorithms are all good fantastic all been discovered for many years, but the programmer, the engineer who is working on this algorithms have to work on them extensively to ensure that lighting related issues, and edge detection, and all of them are taken care. You take extreme caution and care and not just go by sensor and actuator, but also believe on several other sensors, which you may have to actually rely on. And, perhaps this is the big limitation of just what used to happen many years ago, where you just used a sensor and a sort of automobile or a actuator, motor and you try to do a few things.

Therefore, all this things about machine learning, artificial intelligence have come in, where several roads, several edges are all trained and trained model is actually driving the system along with the sensors that like camera and other sensors, which essentially trying to identify the edges as well as to avoid obstacles when LiDARs and other related sensors are actually used. So, before we go on to the code let me show you a little more detail of this Kobuki system.

(Refer Slide Time: 17:09)



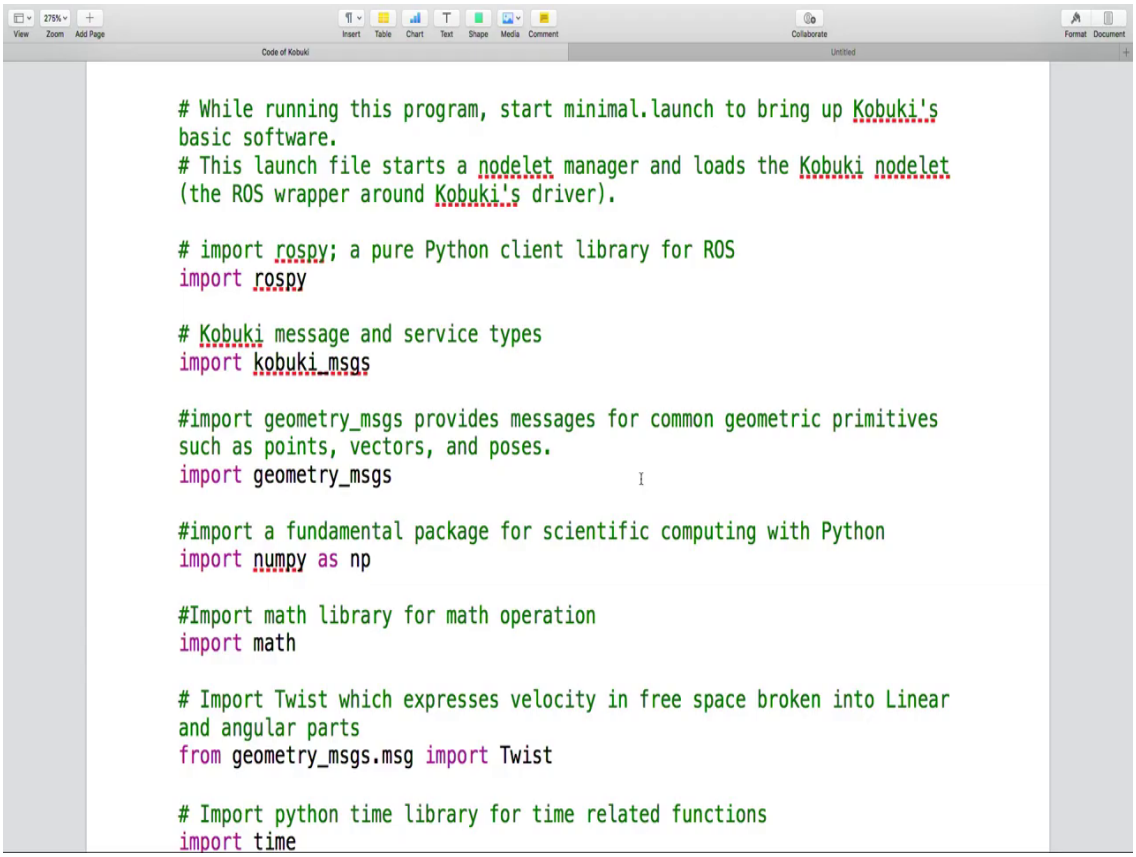
You can see the wheeled robot with camera above. We had used this camera and this was looking at the lane, this camera sensor was looking at the lane.



The wheels bottom of robot, you see 2 tires; opposite to each other.Which we were applying the thrust and the system was moving.

We will now turn our attention to the source code and get into the details of the source code to finally wrap up this discussion on use of canny edge and Hough line transform for the purposes of you know driving through a lane.

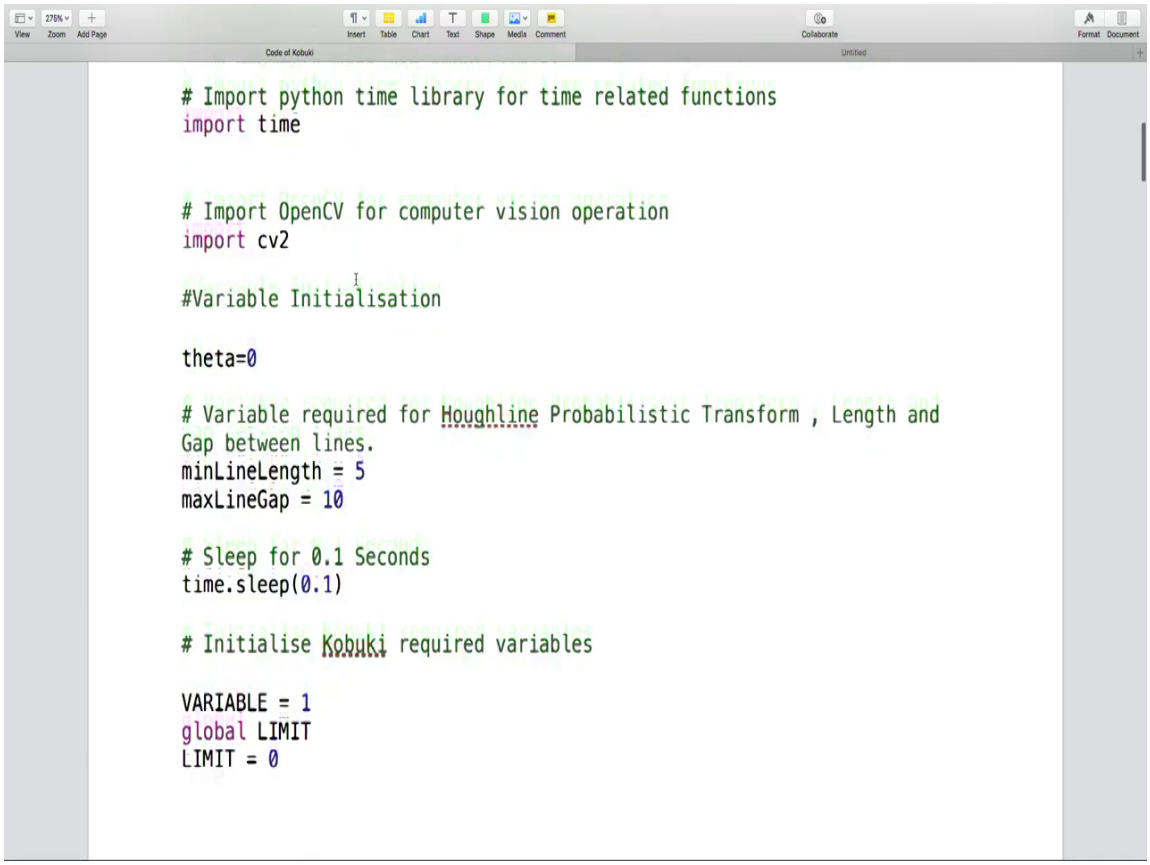
(Refer Slide Time: 18:05)



So, now, let us look at the basic code which we demonstrated to you. Essentially the kobuki is running with its own controller, and there is a controller on board and the controller is running Ubuntu and on top of Ubuntu operating system is running the robot operating system which is called ROS.

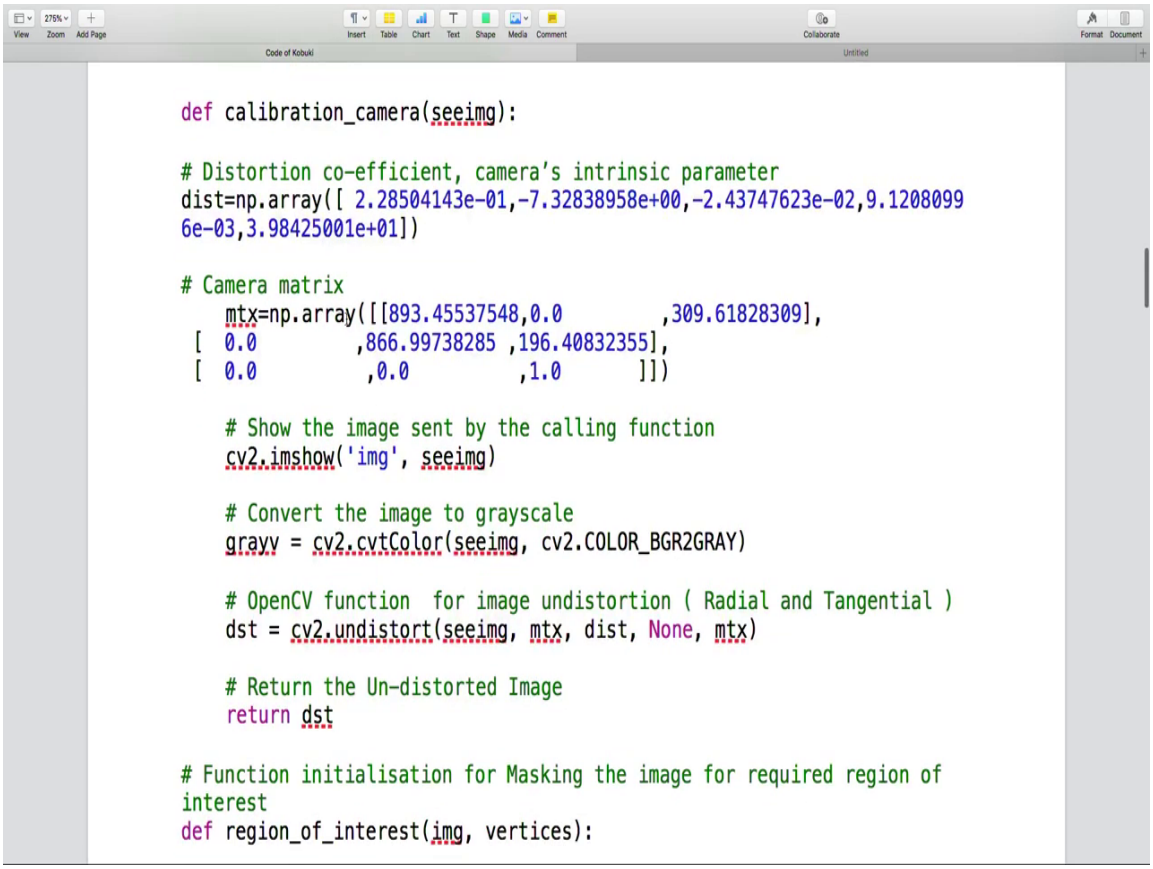
The ROS platform essentially is useful because it is able to interface between the autonomous vehicle, which is in our case the robotic platform kobuki and the camera sensor which is looking at the images. So, the first part of the code is all about the calibration.

(Refer Slide Time: 18:47)



If, you come down you will see that several python libraries have to be imported and you see the first function camera calibration.

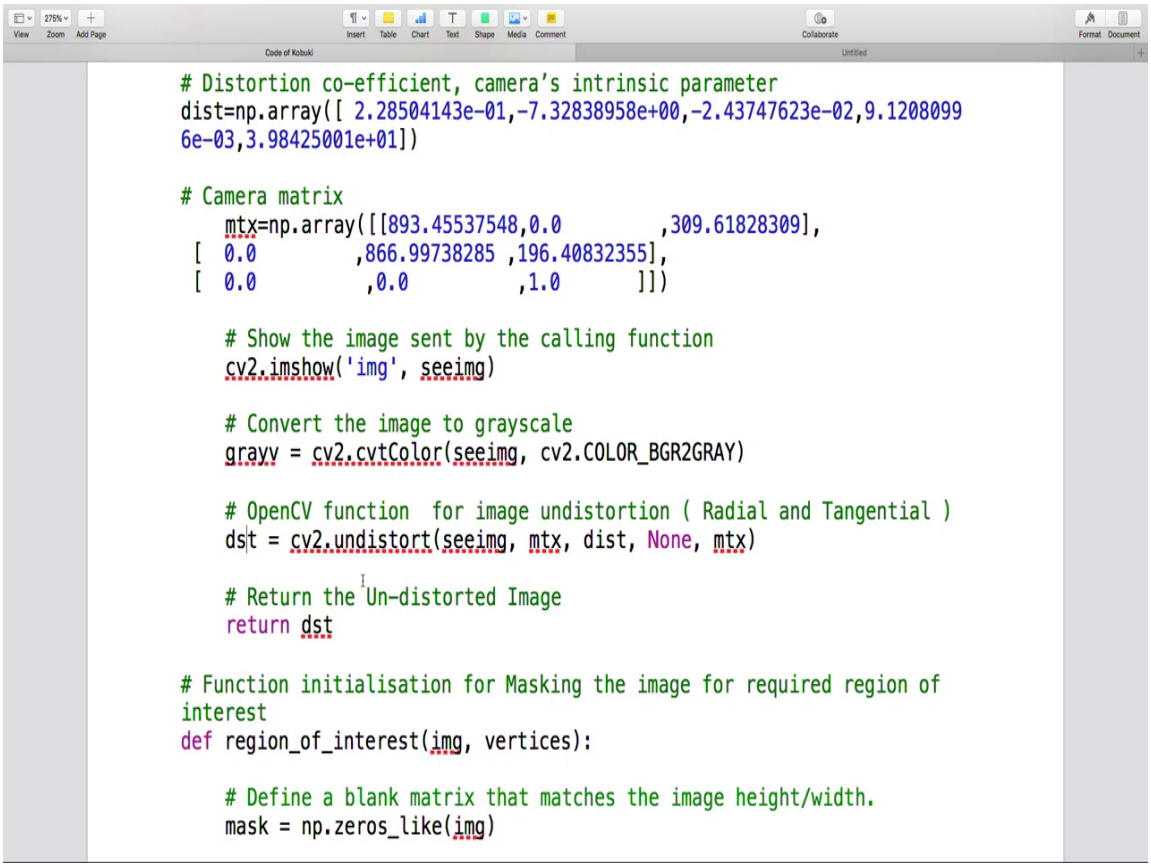
(Refer Slide Time: 18:57)



```
def calibration_camera(seeimg):

# Distortion co-efficient, camera's intrinsic parameter
dist=np.array([ 2.28504143e-01,-7.32838958e+00,-2.43747623e-02,9.1208099
6e-03,3.98425001e+01])

# Camera matrix
    mtx=np.array([[893.45537548,0.0         ,309.61828309],
 [  0.0         ,866.99738285 ,196.40832355],
 [  0.0          ,0.0          ,1.0       ]])

    # Show the image sent by the calling function
    cv2.imshow('img', seeimg)

    # Convert the image to grayscale
    grayy = cv2.cvtColor(seeimg, cv2.COLOR_BGR2GRAY)

    # OpenCV function  for image undistortion ( Radial and Tangential )
    dst = cv2.undistort(seeimg, mtx, dist, None, mtx)

    # Return the Un-distorted Image
    return dst

# Function initialisation for Masking the image for required region of
interest
def region_of_interest(img, vertices):
```

You can see that calibration underscore camera, whatever image is being seen is been calibrated, you get the distortion coefficients, and all of that is displayed out there in the first part. Then we also have the camera matrix, I mentioned to you about camera matrix that matrix is also obtained.

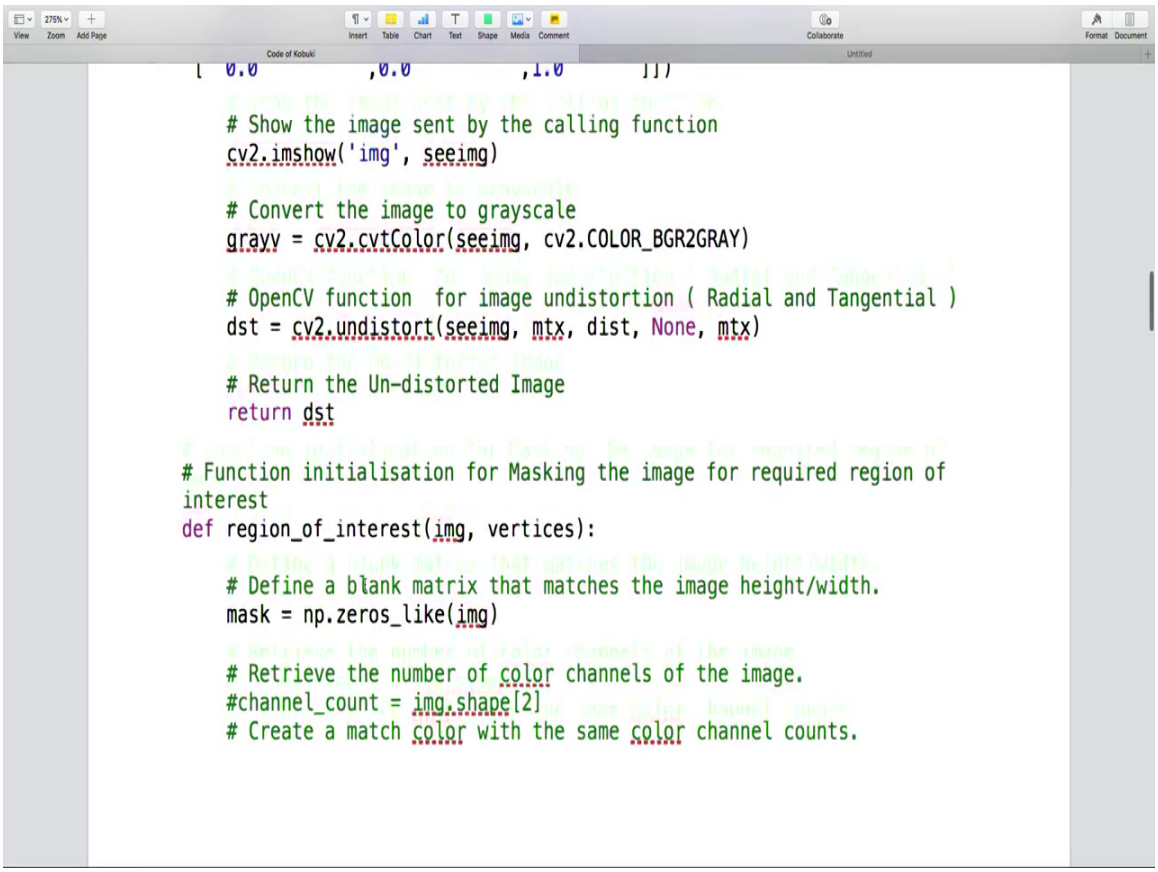
(Refer Slide Time: 19:16)



```
# Distortion co-efficient, camera's intrinsic parameter
dist=np.array([ 2.28504143e-01,-7.32838958e+00,-2.43747623e-02,9.1208099
6e-03,3.98425001e+01])

# Camera matrix
    mtx=np.array([[893.45537548,0.0         ,309.61828309],
 [  0.0         ,866.99738285 ,196.40832355],
 [  0.0          ,0.0          ,1.0       ]])

    # Show the image sent by the calling function
    cv2.imshow('img', seeimg)

    # Convert the image to grayscale
    grayy = cv2.cvtColor(seeimg, cv2.COLOR_BGR2GRAY)

    # OpenCV function  for image undistortion ( Radial and Tangential )
    dst = cv2.undistort(seeimg, mtx, dist, None, mtx)

    # Return the Un-distorted Image
    return dst

# Function initialisation for Masking the image for required region of
interest
def region_of_interest(img, vertices):

    # Define a blank matrix that matches the image height/width.
    mask = np.zeros_like(img)
```

So, this is for as far as the camera is concerned, it is still not giving you anything undistorted. So, you need an undistorted image. So, which means you have to run back to the opencv function called undistort and you will get a nice undistorted image.

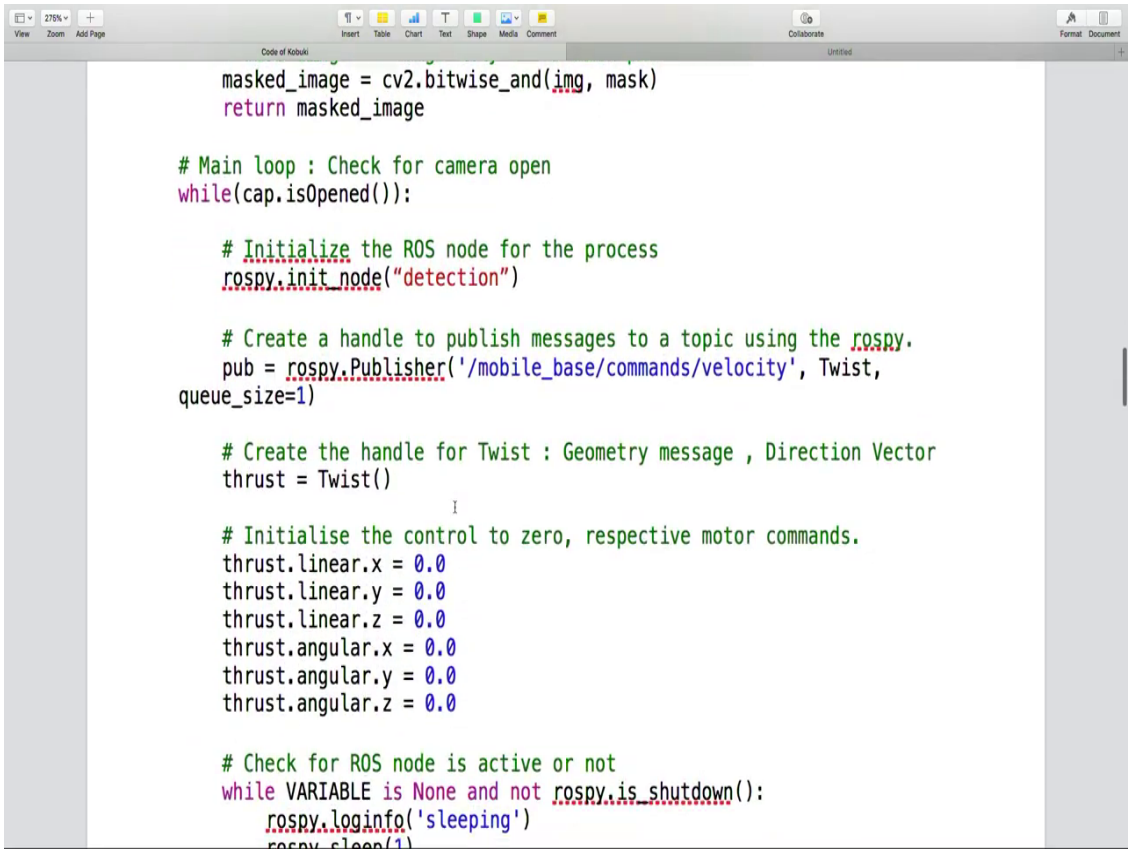
(Refer Slide Time: 19:45)



Now after all this is done you may have to do the region of interest, if you recall the camera was pointing completely towards the floor which essentially created a nice triangle for you and in that triangle only you are looking for edges and you are also looking only for lines in the triangle.

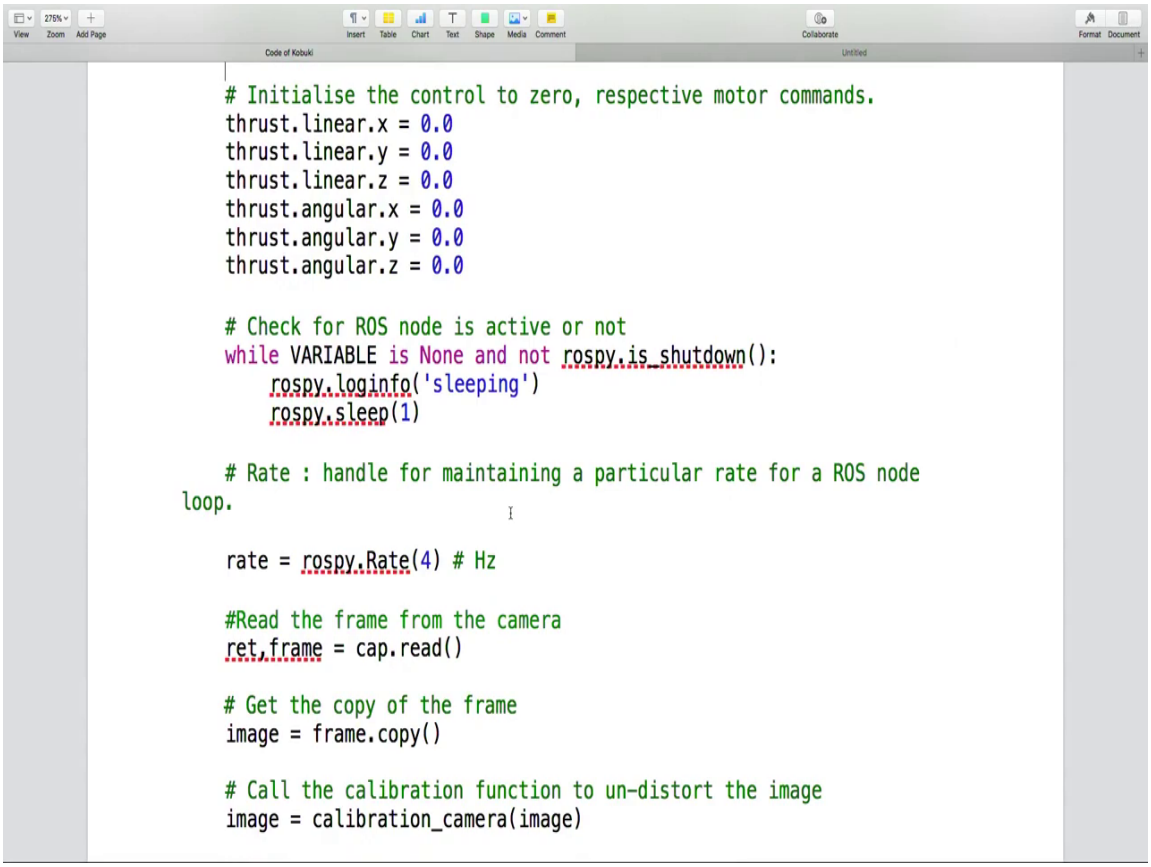
So, it all depends on how much region of interest you are actually looking at. So, here I want to pose a very interesting problem. The size of the triangle should be is actually left to your hands. If, you tilt the camera little up then it would look at the look at a larger image, and perhaps even they able to cite the turnings much ahead of the point where it as actually got to turn. So, it is your ingenuity on how you can actually write this code so that you will be able to make quick decisions and be able to write some real high performance algorithms.

(Refer Slide Time: 20:44)



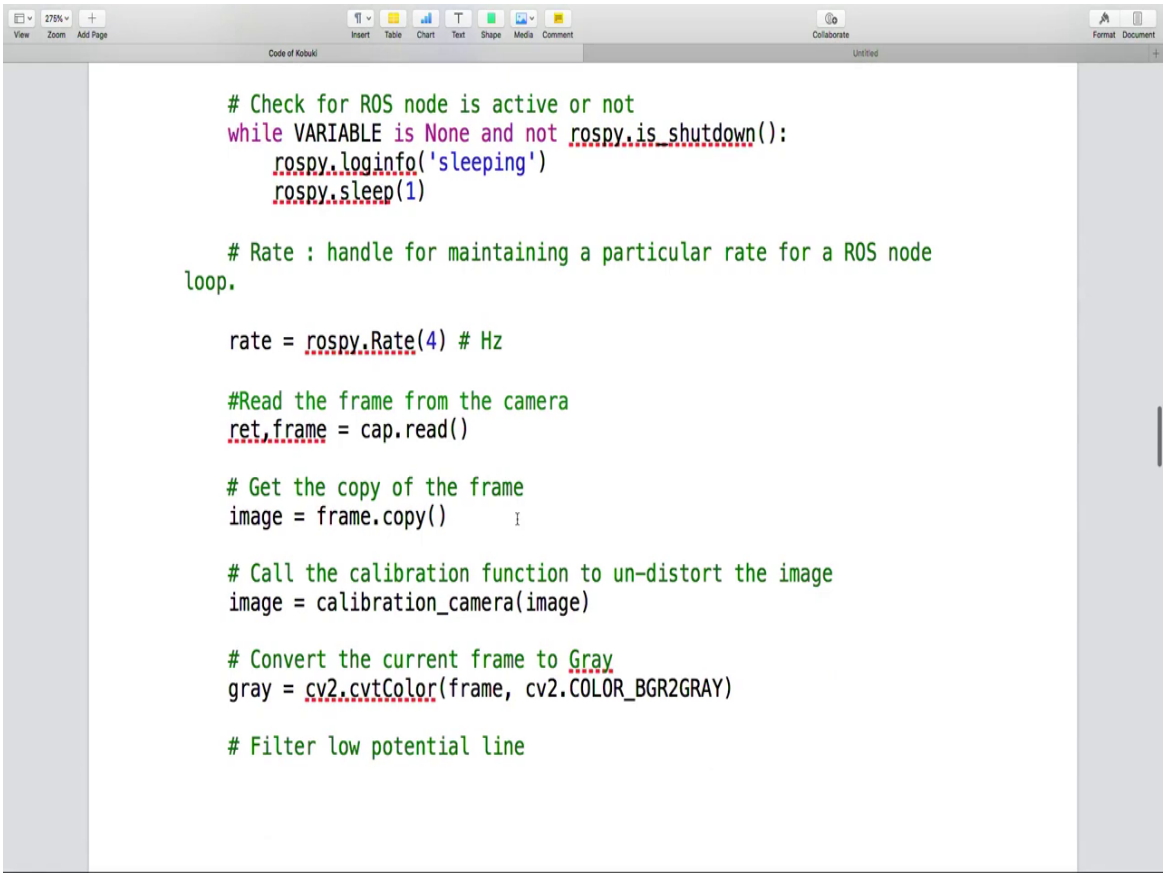
So, now, essentially the camera information which has to be acquired and the information has to be passed to give thrust to the required motors which are connected to the robotic platform. So, you can see that; create handle to publish messages to a topic called rospy, that I will keep taking the information from the camera and keep feeding it to the motors which are part of the robotic platform.

(Refer Slide Time: 21:15)



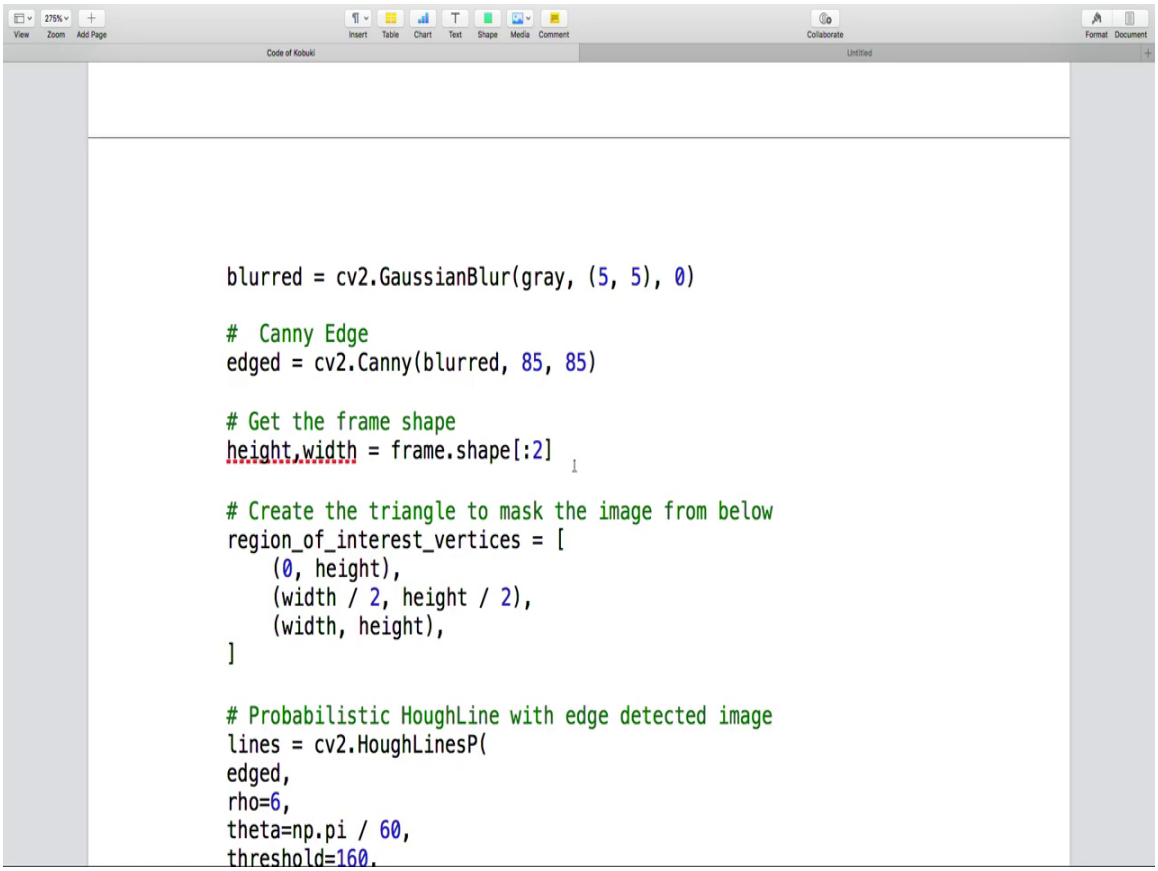
So, the geometry message, direction vector, thrust and all of that is you can see is all initialised. To begin with it has to be initialize to 0, and that is essentially what it is done. The rest part is all to check whether you know some safety conditions, whether we should do it if ROS node is down active and all that. So, these are some very simple things.

(Refer Slide Time: 21:40)



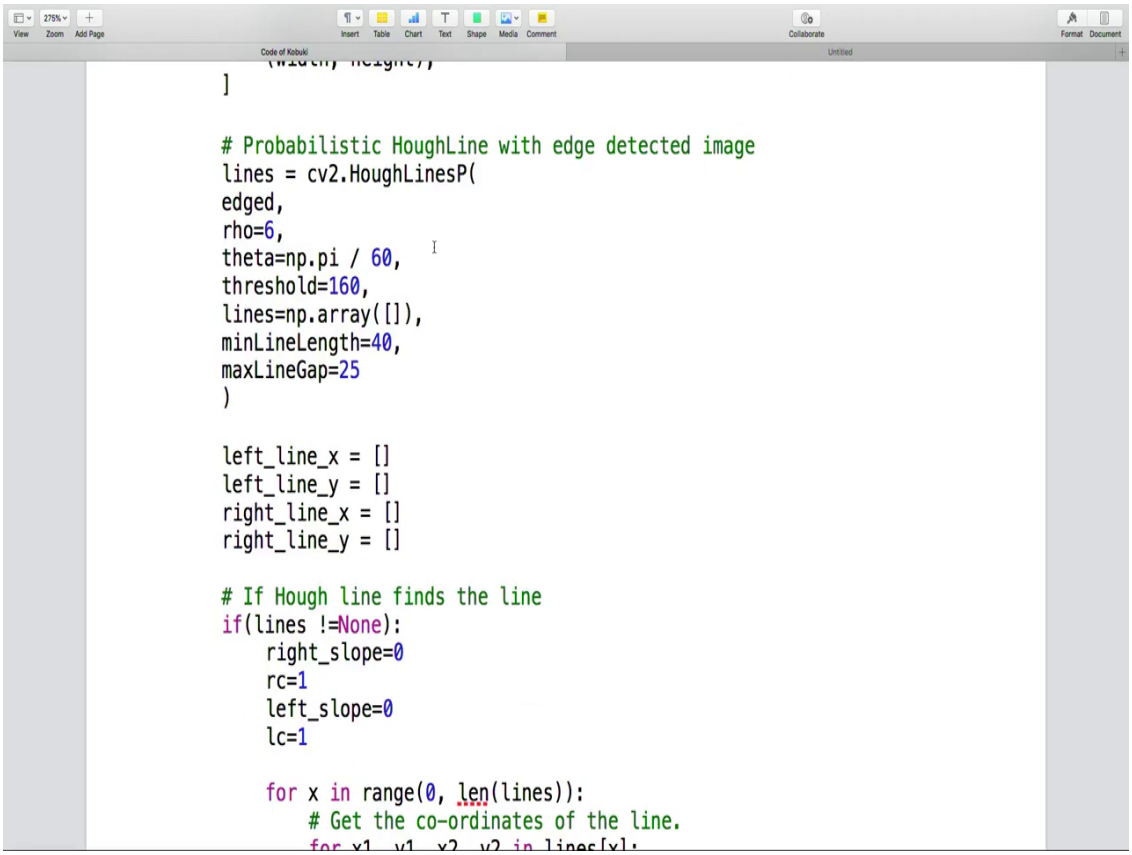
Let us now move on directly to place where the first part of the calibration is done, the first part of the canny edge has to start, you essentially have to start with blurring the image.

(Refer Slide Time: 21:44)



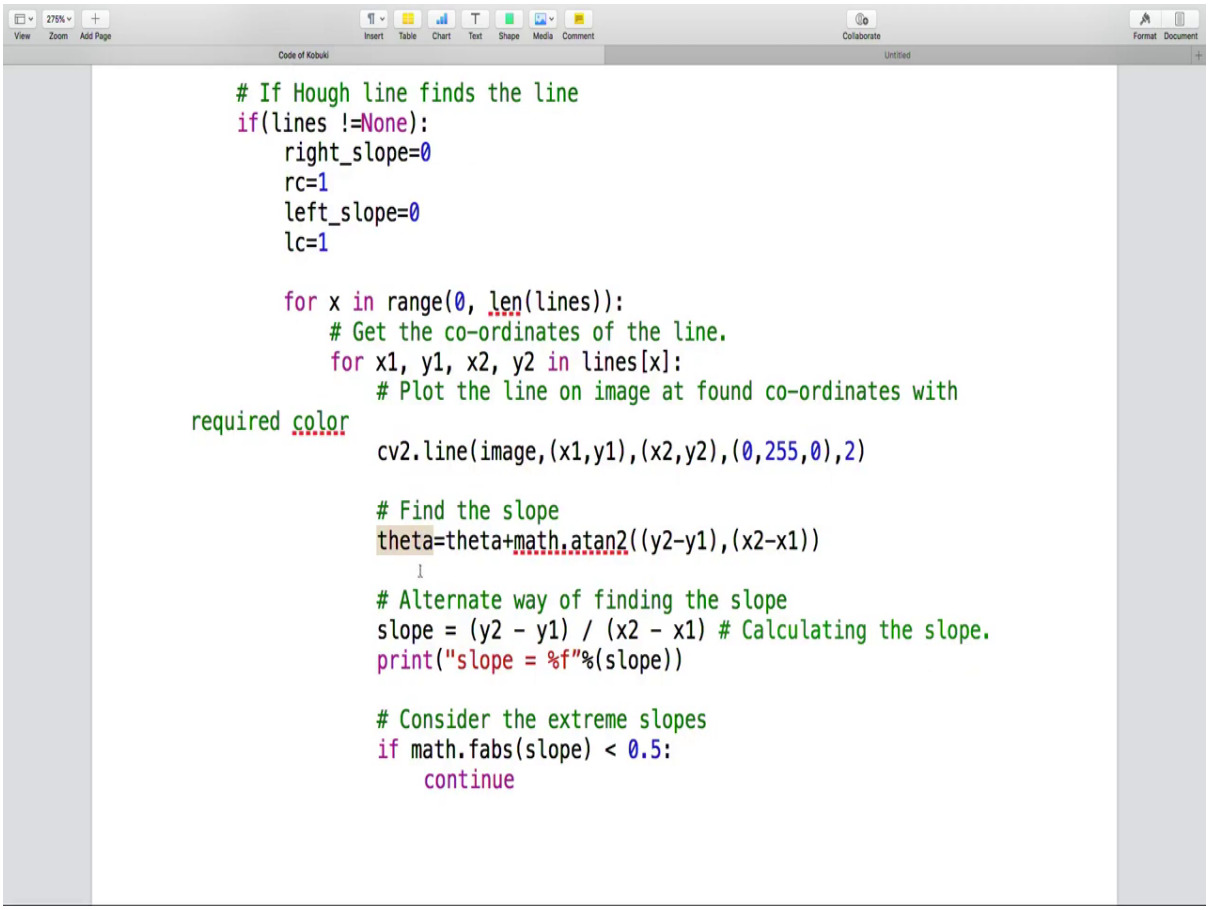
So, let us look at blurring. So, you can see it is blurred using cv2.GaussioanBlur. And, then the canny edge is called you have cv2.Canny which essentially feeding the blurred image and then the size that you essentially have which is 85. So, the canny edges essentially you are passing the blurred image and the minimum and the maximum value in this case we have just taken it as 85 comma both are the same.

(Refer Slide Time: 22:22)



Then you look at the triangle of interest, because this is a region of interest you are specifying several things with respect to the edges themselves. What is the height of the triangle that you are of interest it is width by 2 and height by 2 essentially that is the size of the triangle of interest. And, then you will be calling the Hough line.

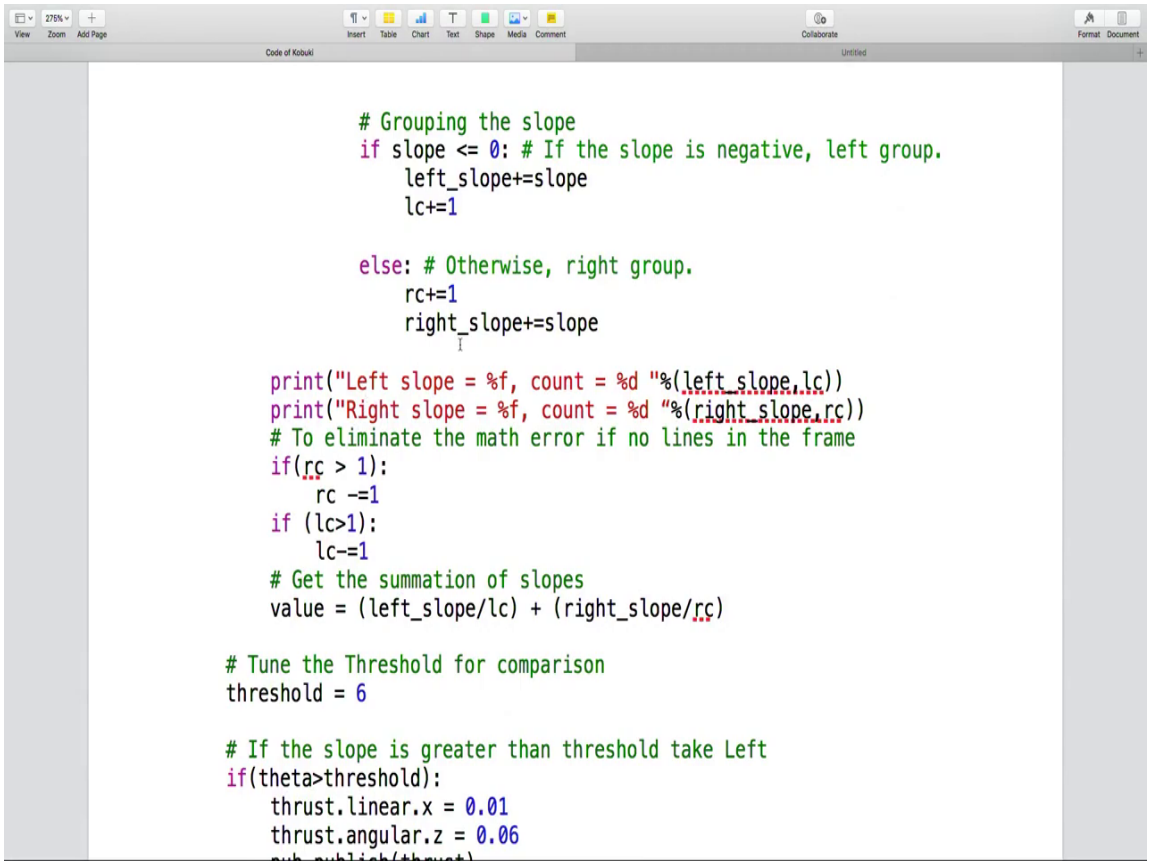
(Refer Slide Time: 22:50)



```
        # If Hough line finds the line
        if(lines !=None):
            right_slope=0
            rc=1
            left_slope=0
            lc=1

            for x in range(0, len(lines)):
                # Get the co-ordinates of the line.
                for x1, y1, x2, y2 in lines[x]:
                    # Plot the line on image at found co-ordinates with
required color
                    cv2.line(image,(x1,y1),(x2,y2),(0,255,0),2)

                    # Find the slope
                    theta=theta+math.atan2((y2-y1),(x2-x1))

                    # Alternate way of finding the slope
                    slope = (y2 - y1) / (x2 - x1) # Calculating the slope.
                    print("slope = %f"%(slope))

                    # Consider the extreme slopes
                    if math.fabs(slope) < 0.5:
                        continue
```

So, you can see directly the Hough line is called ; cv2.HoughlineP and then you will be estimating the slopes. I mentioned to you about slopes and you will continuously be estimating the slopes, to see where exactly the there is a change in the slope, find the slope, you can see and some other simple functions that have to be done to calculate, to correct it.

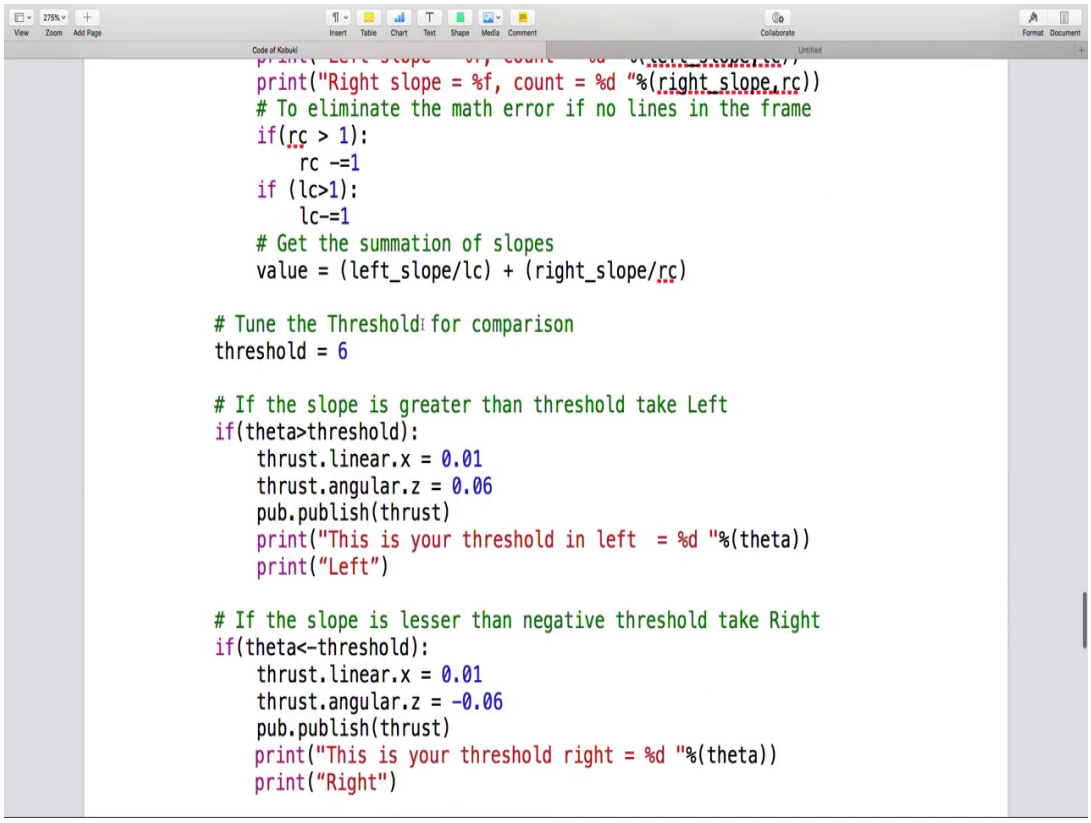
(Refer Slide Time: 23:04)



```
            # Grouping the slope
            if slope <= 0: # If the slope is negative, left group.
                left_slope+=slope
                lc+=1

            else: # Otherwise, right group.
                rc+=1
                right_slope+=slope

        print("Left slope = %f, count = %d "%(left_slope,lc))
        print("Right slope = %f, count = %d "%(right_slope,rc))
        # To eliminate the math error if no lines in the frame
        if(rc > 1):
            rc -=1
        if (lc>1):
            lc-=1
        # Get the summation of slopes
        value = (left_slope/lc) + (right_slope/rc)

    # Tune the Threshold for comparison
    threshold = 6

    # If the slope is greater than threshold take Left
    if(theta>threshold):
        thrust.linear.x = 0.01
        thrust.angular.z = 0.06
```

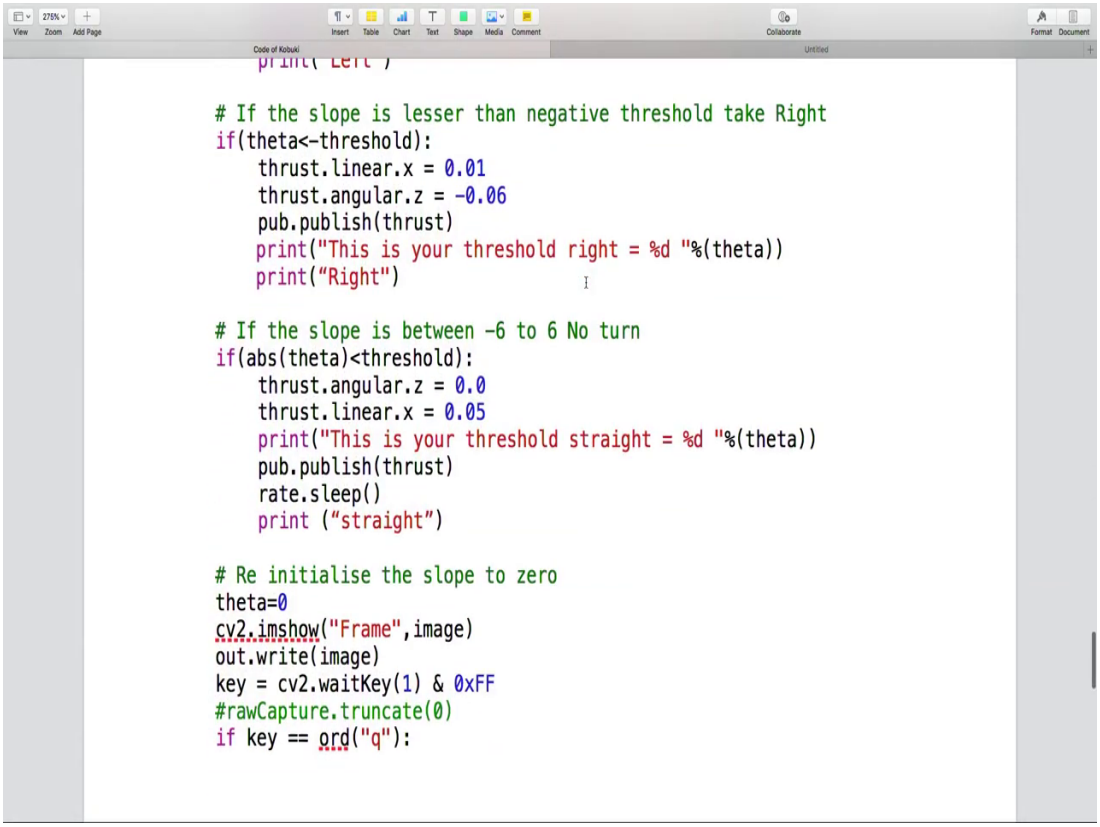
And, then you will essentially find the left slope and the right slope and then do some very simple math error correction if there any.

(Refer Slide Time: 23:14)



```
        print("Right slope = %f, count = %d "%(right_slope,rc))
        # To eliminate the math error if no lines in the frame
        if(rc > 1):
            rc -=1
        if (lc>1):
            lc-=1
        # Get the summation of slopes
        value = (left_slope/lc) + (right_slope/rc)

    # Tune the Threshold for comparison
    threshold = 6

    # If the slope is greater than threshold take Left
    if(theta>threshold):
        thrust.linear.x = 0.01
        thrust.angular.z = 0.06
        pub.publish(thrust)
        print("This is your threshold in left  = %d "%(theta))
        print("Left")

    # If the slope is lesser than negative threshold take Right
    if(theta<-threshold):
        thrust.linear.x = 0.01
        thrust.angular.z = -0.06
        pub.publish(thrust)
        print("This is your threshold right = %d "%(theta))
        print("Right")
```

(Refer Slide Time: 23:26)



```
    print("Left")

# If the slope is lesser than negative threshold take Right
if(theta<-threshold):
    thrust.linear.x = 0.01
    thrust.angular.z = -0.06
    pub.publish(thrust)
    print("This is your threshold right = %d "%(theta))
    print("Right")

# If the slope is between -6 to 6 No turn
if(abs(theta)<threshold):
    thrust.angular.z = 0.0
    thrust.linear.x = 0.05
    print("This is your threshold straight = %d "%(theta))
    pub.publish(thrust)
    rate.sleep()
    print ("straight")

# Re initialise the slope to zero
theta=0
cv2.imshow("Frame",image)
out.write(image)
key = cv2.waitKey(1) & 0xFF
#rawCapture.truncate(0)
if key == ord("q"):
```

And, then you will tune the threshold, you will apply a threshold. In this case I mention to you that we take 6 as a threshold. And, we mention to you already that a turn has to essentially be if the sum of the left and the right slopes is less than minus 6 you will be doing a right turn, which was essentially point A. And, if the sum is greater than 6 you will be doing a left turn. So, this was already mentioned and all that essentially means from the sensor which is the data being published by the ROS, you directly give it to the thrust motors here. And, essentially you will be turning either left or your right or you will be going straight.

You can see this, this is your threshold take a right turn, this is your threshold go straight and this is your threshold go left. All these 3 threshold conditions are checked based on the sum of slopes; this is the simplest possible algorithm you can think of. And, essentially you will be calculating on that slope.

The whole application essentially is taking care of using sum of slopes left, right and going straight. So, this algorithm as you can see is very naive in its turning left and turning right. You are not doing any calculation of any nature, you are just getting some number and then based on minus 6 you do something plus 6 you do something is very naive.

This is definitely not the way you would want to design an autonomous system. You have to do continuous estimation of the angle how much to turn. And, that essentially means you have to go away from the simple Briatenberg algorithm and come up with much more sophisticated algorithms. Indeed that is left to you, to start thinking now, how I can improve on this very simple threshold base thrusts that I given to motors into something much more useful much more practical.