CS 430

Robot Control - Path decisions

Part 1

Line following with light sensors.

- One or two light sensors under the robot.
- Algorithm for single light sensor:

```
if(white)
   turnLeft
else
   turnRight
```

- Algorithm for two light sensors uses a state machine.

States:

1. Two black readings go forward
2. Right black, left white -> slowly turn right until you return to state 1.
3. Right white, left black -> slowly turn left until you return to state 1.
4. Two white -> if previous state is 2 -> turn hard right
5. Two white -> if previous state is 3 -> turn hard left
6. Two white -> previous state is 5 -> Go to unknown state (Optional, and sometimes several states of 4, 5 have to be met until state 6 is invoked).

https://www.youtube.com/watch?v=mJV-KDqHgDQ

FAILS: https://www.youtube.com/watch?v=8LCWprnKDcg

Line following with a camera:

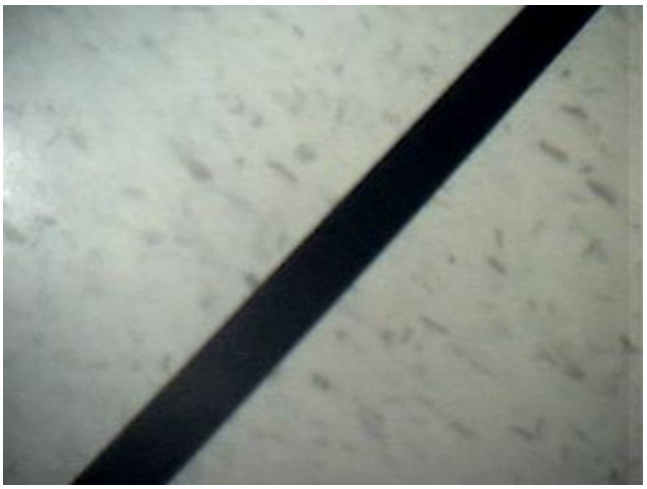
Images from a robot camera:

Same line different lighting. Lots of black noise outside of line.

How do you correct for these things and determine which way to vear, turn and proceed??

Other issues to think about:

- Diagonal lines



- Curved line

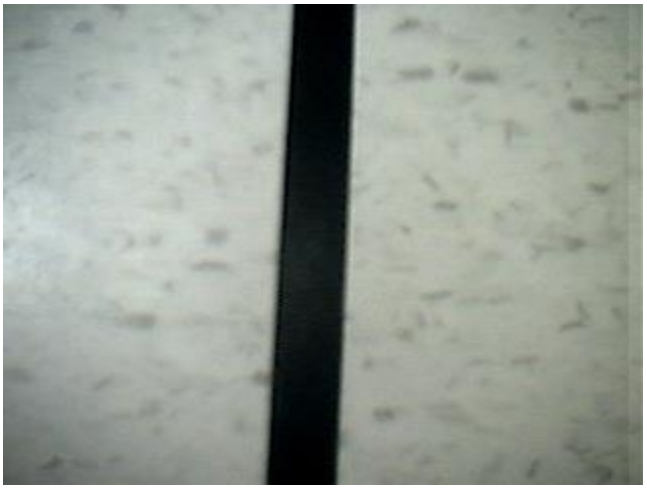
- Curved line crossing two tiles:

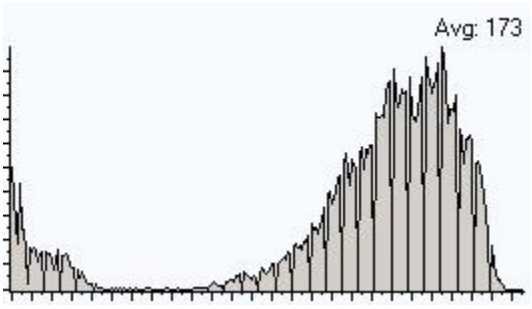
- End of line:

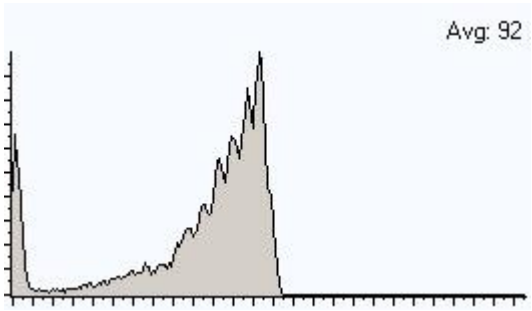
First we will cover the lighting issues:

The last couple of lectures on histograms is going to be useful now.

As you can see, histograms for lighter images slump towards the right (towards the 255 or highest value) whereas darker images have histograms with most pixels closer to zero. This is obvious but using the histogram representation we can better understand how transformations to the image change the underlying pixels.







The next step is to see if we can correct these two images so that they look closer to one another. We do that by normalizing the images ...

To counter the effects of bad lighting we have to normalize the image. Image normalization attempts to spread the pixel intensities over the entire range of intensities. Thus if you have a very dark image the resulting normalization process will replace many of the dark pixels with lighter pixels while keeping the relative positions the same, i.e. two pixels may be made lighter but the darker of the two will still be darker relative to the second pixel.

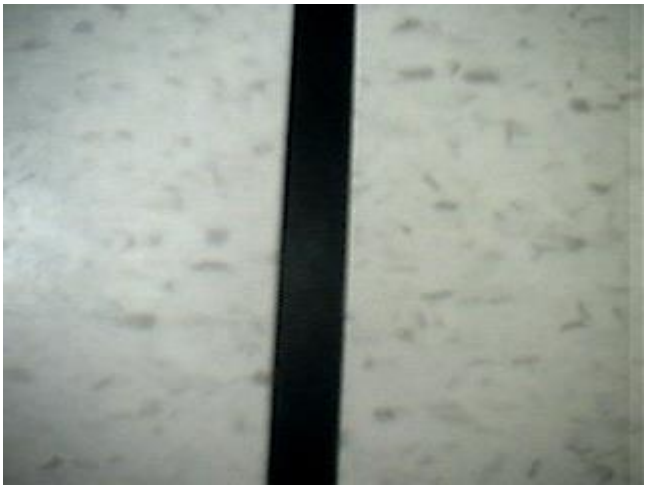
By evenly distributing the image intensities other image processing functions like thresholding which are based on a single cutoff pixel intensity become less sensitive to lighting.

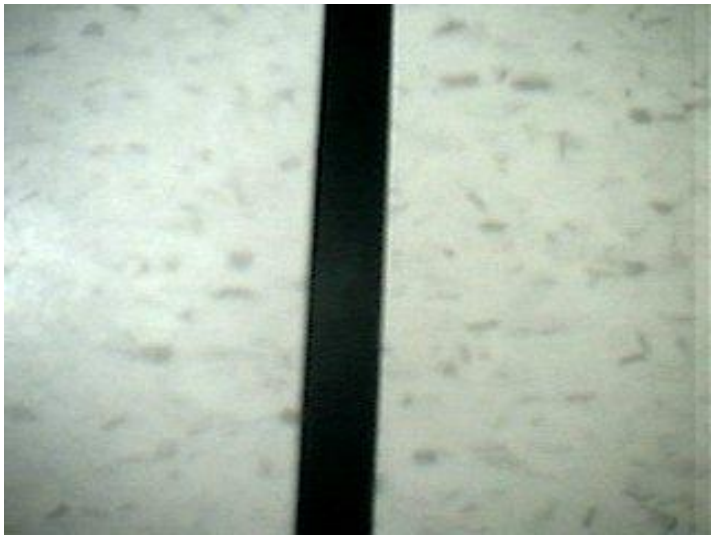
For example, the following images from the previous page show what happens during normalization. It is important to note that any image transformation that is meant to improve bad images must also preserve already good ones. In testing image processing functions be sure to always understand the negative side effects that any function can have.

Normalization is a <u>linear</u> process. If the intensity range of the image is 50 to 180 and the desired range is 0 to 255 the process entails subtracting 50 from each of pixel intensity, making the range 0 to 130. Then each pixel intensity is multiplied by 255/130, making the range 0 to 255. Auto-normalization in image processing software typically normalizes to the full dynamic range of the number system specified in the image file format.

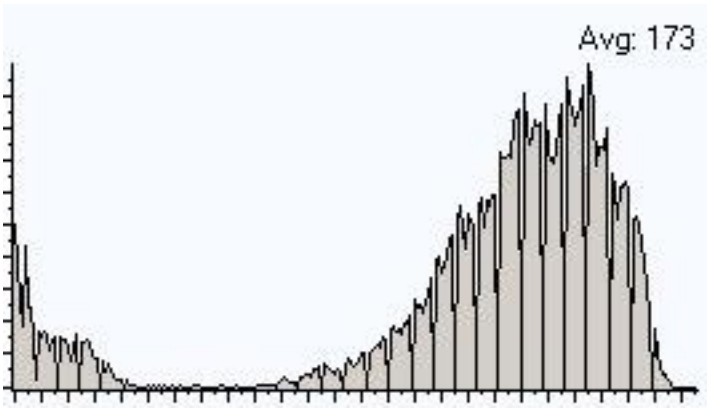
Image normalization attempts to spread the pixel intensities over the entire range of intensities.

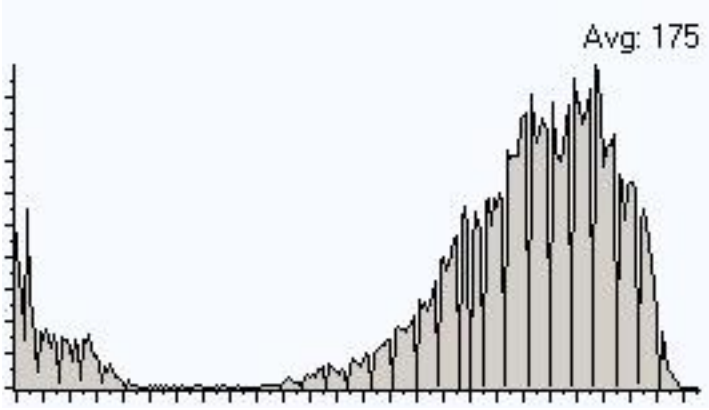
Note that when doing this procedure you must keep good pictures good, and create good pictures with bad pictures. If you create good from bad, but bad from good, it's a useless procedure.

Good picture before: 

After  Still Good

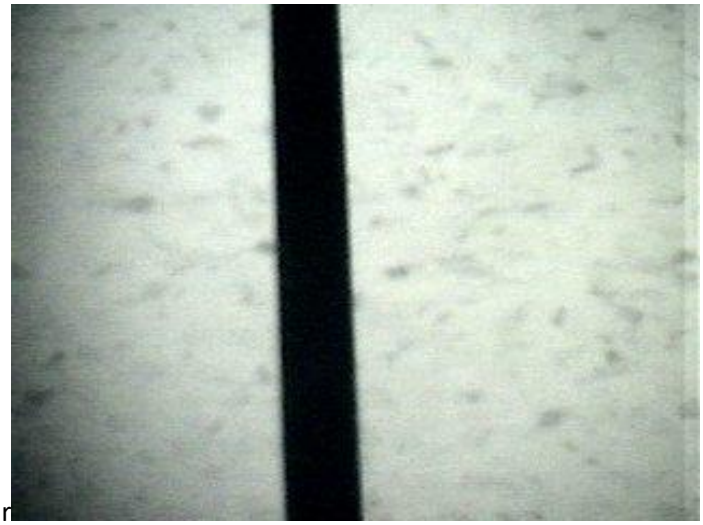
Histograms of good picture before and after the normalization (stretch):



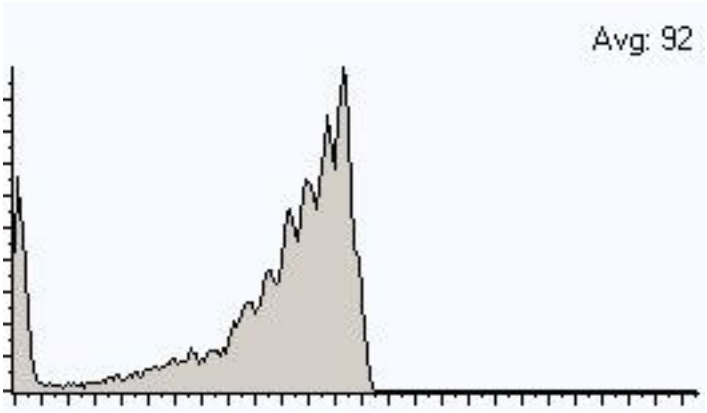


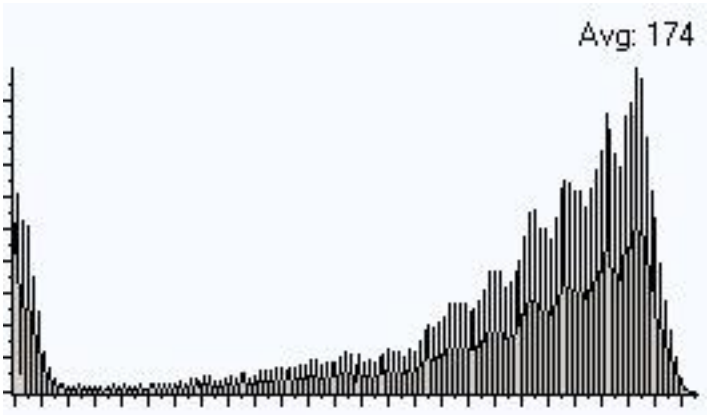
Now the dark picture

Before: 

After 

Histograms:





The bad image experienced a large amount of change as the image intensities did not cover the entire intensity range due to bad lighting. You can see from the histogram that the image intensities are now more evenly distributed.

Also you can note how the new histogram appears to be not as solid as the original. This is due to how the intensity values are stretched. Since the new image has exactly the same number of pixels as the old image the new image still has many pixels intensity values that do not exist and therefore show up as gaps in the histogram. Adding another filter like a mean blur would cause the histogram to become more solid again as the gaps would be filled due to smoothing of the image.

Next we need to start focusing on extracting the actual lines in the images.

Image normalization attempts to spread the pixel intensities over the entire range of intensities.

Next we need to start focusing on extracting the actual lines in the images. We need to find properties or features of the image that can help us steer the robot.

We need to find the line, extract the line from the rest of the image?

How would you do that?

Edge Detection

Find the edges. To find the edges we need to do a convolution filter. There are lots of convolution filters we will use this semester, but for now we are going to use a very simple convolution filter.

A convolution (taken straight from Wikipedia so you know it's true) - **convolution** is a mathematical operation on two functions *f* and *g*, producing a third function that is typically viewed as a modified version of one of the original functions.

The way we perform edge detection is to run the image through a convolution filter that is 'focused' on detecting line edges. A convolution filter is a matrix of numbers that specify values that are to be multiplied, added and then divided from the image pixels to create the resulting pixel value.

Using a convolution filter on an image means you recalculate a pixel based on the surrounding pixels, you use a mask on each individual pixel.

Example convolution filter for line detection:

-1	-1	-1
-1	8	-1
-1	-1	-1

The row=2, column=2 pixel and its neighborhood from the image above: The row=2, column=2 pixel and its neighborhood from the image above:

34	22	77
50	150	77
93	0	77

To apply the convolution filter multiply the filter values with the image data block. Work with each pixel and its 3x3 neighborhood:

-1*34	-1*22	-1*77
-1*50	8*150	-1*77
-1*93	-1*0	-1*77

Then sum all the values:
(-34)+(-22)+(-77)+
(-50)+(1200)+(-77)+
(-93)+(0)+(-77) = 770

Divide by the divisor and add the bias.
(770/divisor)+bias=770 (in this example divisor=1, bias=0)

If the new pixel value is > 255 set it to 255
If the new pixel value is < 0 set it to 0

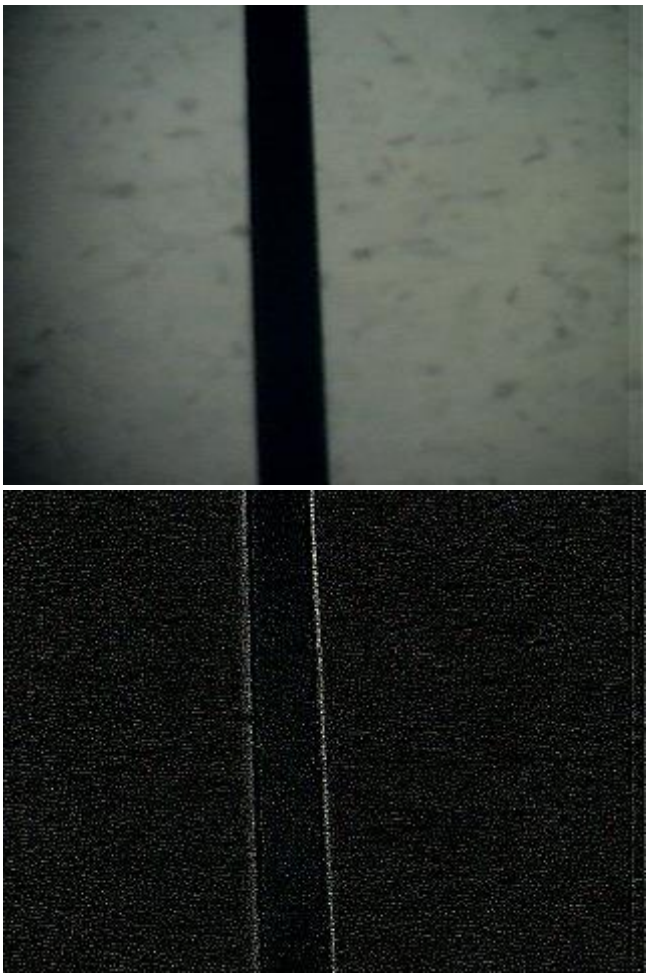
34	22	77
50	255	77
93	0	77

Continue with all other 3x3 blocks in the image using original values. For example the next image block could be

22	77	48
150	77	158
0	77	219

Note the 3x3 "window" is shifted to the right by one and that the new pixel value is NOT used but stored as a second new image.

Now after the images above were put through the edge detector filter the images looked like the following:



Not bad, but not good enough. The edges have definitely been detected, but there is a lot of noise, and the lines are not continuous, they have breaks and false edges.

We need a better technique.

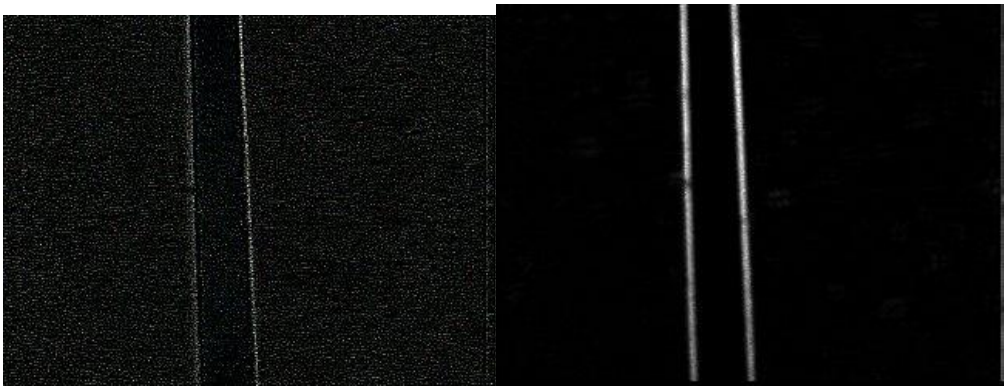
1. Larger filter to start. 5x5, this will produce thicker lines

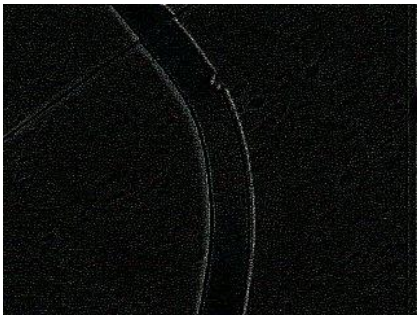
-1	-1	-1	-1	-1
-1	0	0	0	-1
-1	0	16	0	-1
-1	0	0	0	-1
-1	-1	-1	-1	-1

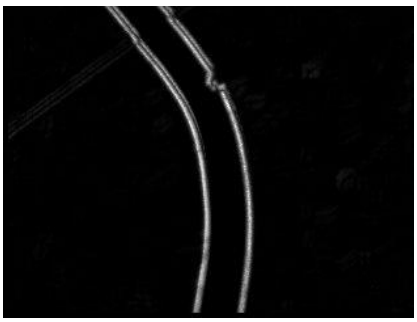
2. Further reduce the speckling issue we will square the resulting pixel value. This causes larger values to become larger but smaller values to remain small. The result is then normalized to fit into the 0-255 pixel value range.

3. Threshold the final image by removing any pixels lower than a 40 intensity value.

The results of this modified technique:







Now we can definitely see the edges. What do the edges tell us? How do we make a decision based on this information?

1. Addition??
2. COG??
    - ✓ We'll use this one.

COG

To calculate the COG of an image add all the x,y locations of non-black pixels and divide by the number of pixels counted. The resulting two numbers (one for x and the other for y) is the COG location.

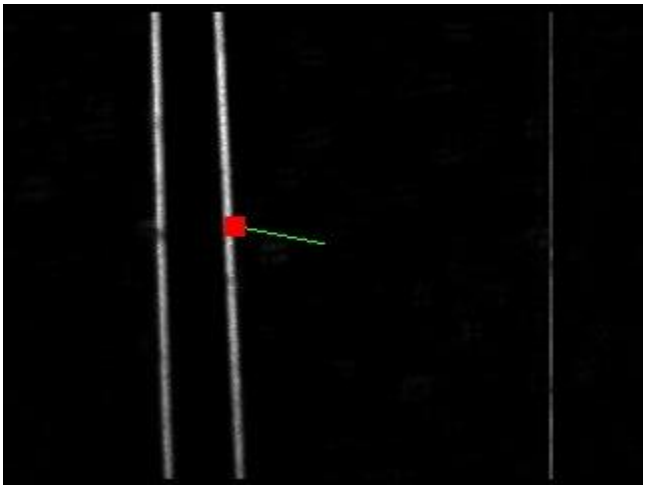
At this point, however, we will use the COG or Center of Gravity of the image to help guide our robot.

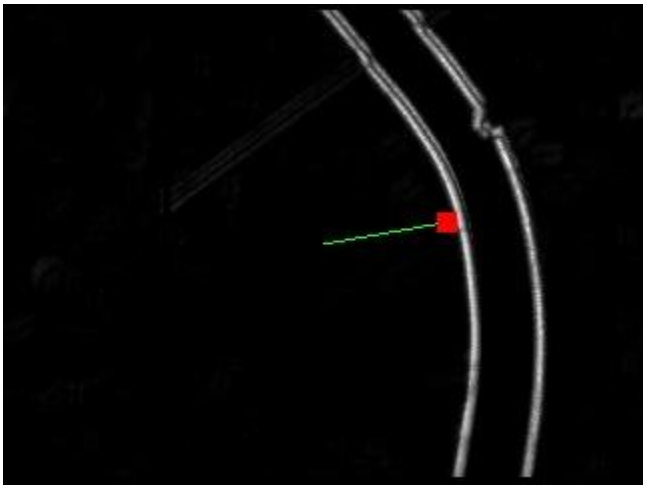
The COG of an object is the location where one could balance the object using just one finger. In image terms it is where one would balance all the white pixels at a single spot. The COG is quick and easy to calculate and will change based on the object's shape or position in an image.

To calculate the COG of an image add all the x,y locations of non-black pixels and divide by the number of pixels counted. The resulting two numbers (one for x and the other for y) is the COG location.

Let's do that for a couple images and see what we get!

Let's do that for a couple images and see what we get! I'm so excited…….. I don't get out much.





Now just like the line following robot at the middle we can make decisions based on one value.

If this is a 256x256 image the middle is 128.

The first image center is obviously left of 128 so turn left slowly.

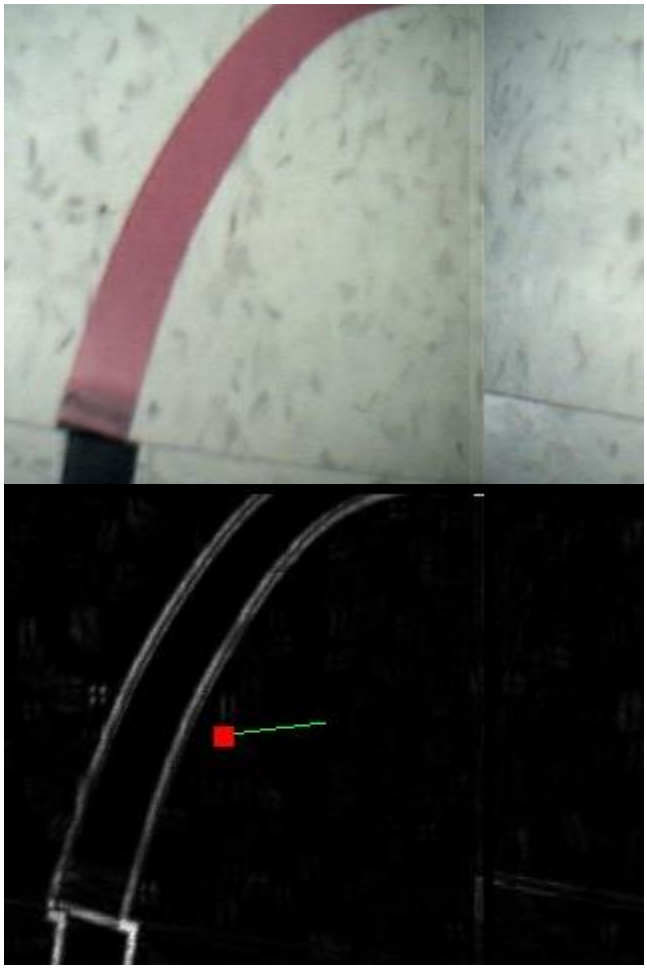
The second image is to the right of 128 so turn right slowly.

If it’s close to 20, or 240 we can turn faster.

What does the height of the COG tell you?

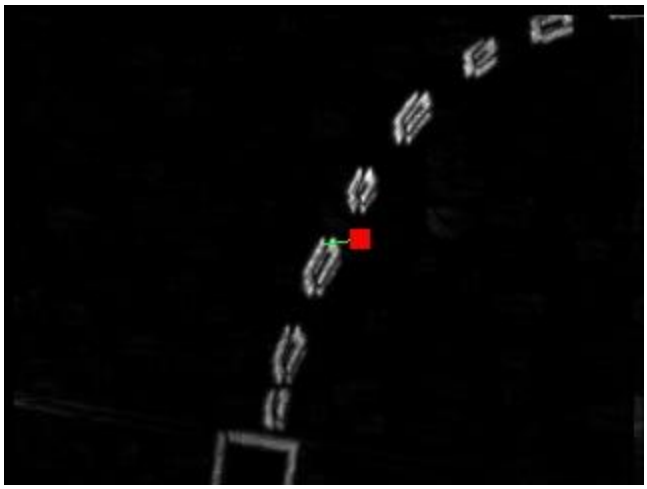
More results ...

This technique works regardless of the color of the line ... as long as the line can be well differentiated from the background we can detect it!

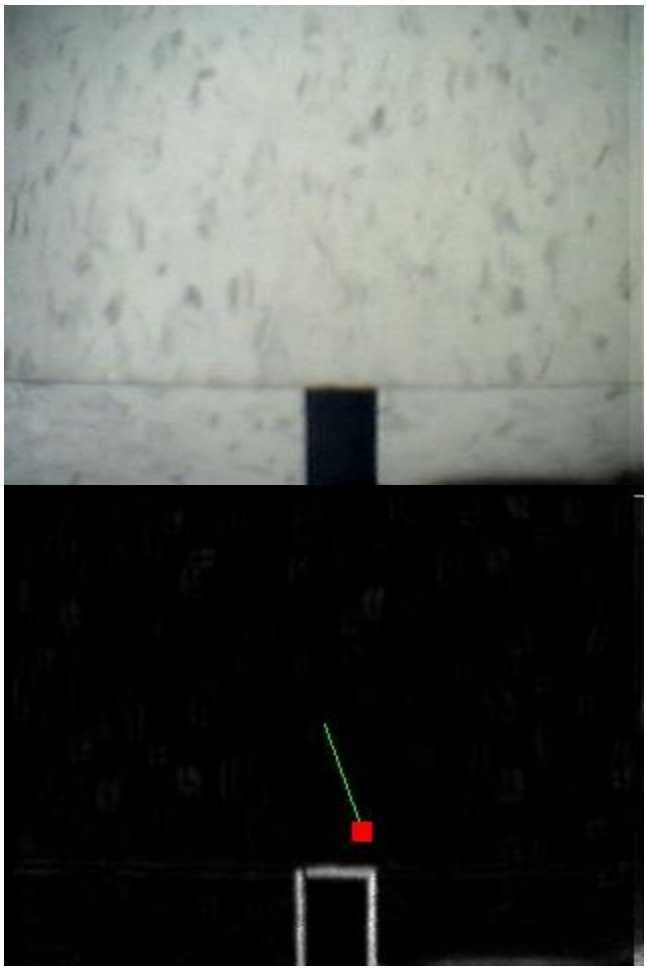


It even works if we reverse the original black line on white tile assumption.

In fact, we don't really even need a connected line!



Here is a case where the end of the line is reached. Notice that the COG has fallen below the center of the screen. Perhaps we can use this condition to stop the robot!



Oops, a mistake! The robot will start following the wrong line ... but no technique is perfect.