Alejandro Garcia
3/11/2020
"I pledge my honor that I have abided by the stevens honor system"

Problem 1

In order to do the processing, the grayscale image was converted into a two dimensional matrix of the first channel. This works because the rgb values of a grayscale image are the same for all channels.

```
 img = np.zeros((img_orig.shape[0], img_orig.shape[1]))

    for y in range(len(img_orig)):
        for x in range(len(img_orig[y])):
            img[y][x] = img_orig[y][x][0]
```

Next the functions to apply convolution and a gaussian filter were developed using the following formulas.
2d Convolution Formula:

$$f[x,y] * g[x,y] = \sum_{n_1=-\infty}^{\infty} \sum_{n_2=-\infty}^{\infty} f[n_1,n_2] \cdot g[x-n_1, y-n_2]$$

```
def Convolution(img, kernel):
    height = img.shape[0]
    width = img.shape[1]
    kheight = kernel.shape[0] // 2
    kwidth = kernel.shape[1] // 2

    pad = ((kheight, kwidth), (kheight, kwidth))
    imgout = np.empty(img.shape, dtype=np.float64)
    img = np.pad(img, pad, mode='constant', constant_values=0)

    for i in np.arange(kheight, height + kheight):
        for j in np.arange(kwidth, width + kwidth):
            partition = img[i - kheight:i + kheight + 1, j - kwidth:j +
kwidth + 1]
            imgout[i - kheight, j - kwidth] = (partition*kernel).sum()
    return imgout
```

Gaussian Formula:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/(2\sigma^2)}$$

```
def GuassianFilter(sigma):
    fsize = 2 * int(4 * sigma + 0.5) + 1
    gaussian = np.zeros((fsize, fsize), np.float64)
    m = fsize // 2
    n = fsize // 2

    for x in range(-m, m+1):
        for y in range(-n, n+1):
            x1 = 2*np.pi*(sigma**2)
            x2 = np.exp(-(x**2 + y**2)/(2* sigma**2))
            gaussian[x+m, y+n] = (1/x1)*x2

    return gaussian
```

Our hessian detector, needed to use sobel filters as derivative operators. Which means that each convolution with an x sobel or a y sobel would give us our derivative of the image. A gaussian filter was then applied as instructed. Giving us the values of our hessian matrix. And allowing us to find our hessian determinant.

$$\mathcal{H}(\vec{x},\sigma) = \begin{bmatrix} L_{xx}(\vec{x},\sigma) & L_{xy}(\vec{x},\sigma) \\ L_{xy}(\vec{x},\sigma) & L_{yy}(\vec{x},\sigma) \end{bmatrix}$$

Hessian matrix

```
    gaussian = GuassianFilter(0.84)

    sobelx = np.array([[1, 0, -1], [2, 0, -2], [1, 0, -1]])
    sobely = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]])

    dx = Convolution(img, sobelx)
    dy = Convolution(img, sobely)
    dxx = Convolution(img, sobelx)
    dyy = Convolution(img, sobely)

    dxx2 = np.square(dxx)
    dyy2 = np.square(dyy)
    dxy = dx * dy

    Ixx2 = Convolution(dxx2, gaussian)
    Iyy2 = Convolution(dyy2, gaussian)
    Ixy = Convolution(dxy, gaussian)

    det_hessian = Ixx2*Iyy2 - np.square(Ixy)
```

Since our main parameter was our threshold. We then check for each point that is that is less than our threshold and set it to 0. Then obtaining a list with all our point values.

```
def hessian(img_orig, threshold):
```

...

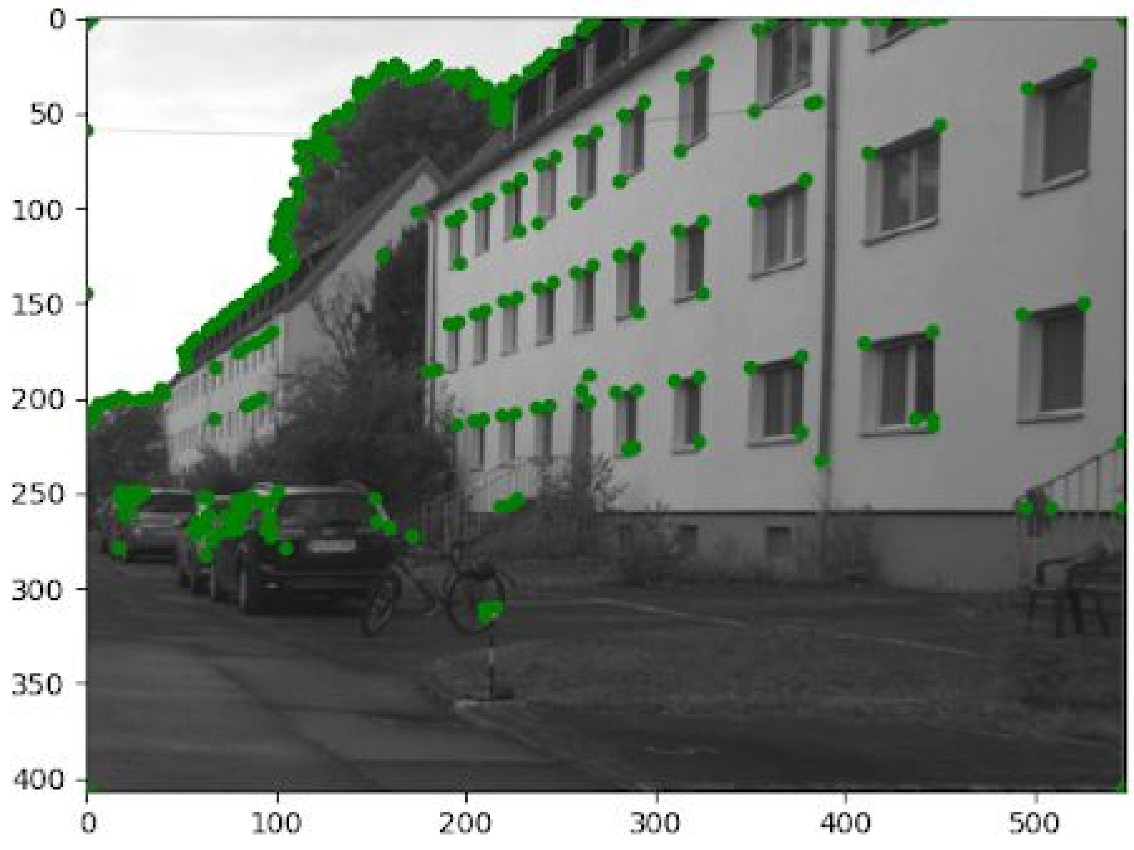
```
for x in range(len(det_hessian)):
        for y in range(len(det_hessian[x])):
            if det_hessian[x][y] <= threshold:
                det_hessian[x][y] = 0
```

With all our points, we apply a non maximum suppression with a 3 by 3 matrix to filter out unnecessary data.

```
def non_max_suppression(points):
    for y in range(len(points) - 2):
        for x in range(len(points[y]) - 2):
            sample = []

            for yy in range(3):
                for xx in range(3):
                    sample += [points[y+yy][x+xx]]

            max_idx = np.argmax(sample)

            for yy in range(3):
                for xx in range(3):
                    if yy*3 + xx != max_idx:
                        points[y+yy][x+xx] = 0

    return points
```

Finally we plot our points and the output the image. Below is the output of the hessian corner detection with a threshold of 0.4.

```
points = np.where(points != 0)

    for p in zip(*points[::-1]):
        plt.gca().add_patch(plp.Circle(p, 3, color='green'))

    plt.savefig('hessian.png')
    plt.show()
```



Problem 2

The main parameters of the ransac function would take an inlier threshold and distance threshold. Given that we have the image and the points needed to create four strong probability lines. A list for our inliers and line was created. As well as a list for previously plotted inliers

```
def ransac(img, points, inlier_threshold, dis_threshold):
    li = []
    plt.imshow(img)
    arbitrary_coordinates_system_already_previous_examined_fully = []
```

For at least 100 times we will do the following:

Step 1.

Take two random points but only once.

```
        ran_range = len(points[0]) - 1
        plt.axis([0, len(img[0]), 0, len(img)])
        plt.gca().invert_yaxis()
        x = []
        y = []

        ran_idx = random.randrange(ran_range)
        (t1,t2) = (points[0][ran_idx],points[1][ran_idx])
        ran_idx = random.randrange(ran_range)
        (t3,t4) = (points[0][ran_idx],points[1][ran_idx])

        if ((t1,t2),(t3,t4)) in
arbitrary_coordinates_system_already_previous_examined_fully:
            print("Collision:", t1, t2, t3, t4)
            continue


arbitrary_coordinates_system_already_previous_examined_fully.append(((t1,t
2),(t3,t4)))

        x.append(t1)
        x.append(t3)
        y.append(t2)
        y.append(t4)
```

Step 2.

Plot a line of fit

```
x = np.array(x)
        y = np.array(y)




        m, b = np.polyfit(x, y, 1)
```

Step 3.

For each point. We will compare the threshold value with the distance. If the distance is less than the threshold value then those are our inliers

```
        inlier_c = 0

        for j in range(len(points[0])):
            y1 = points[0][j]
            x1 = points[1][j]

            if m == 0:
                continue

            b1 = y1 + x1 / m
            xp = (b1 - b) / (m + 1 / m)
            yp = m * xp + b
            d = math.sqrt((y1 - yp)**2 + (x1 - xp)**2)

            if d < dis_threshold:
                inlier_c += 1
                y = np.append(y, yp)
                x = np.append(x, xp)
```

Step 4.

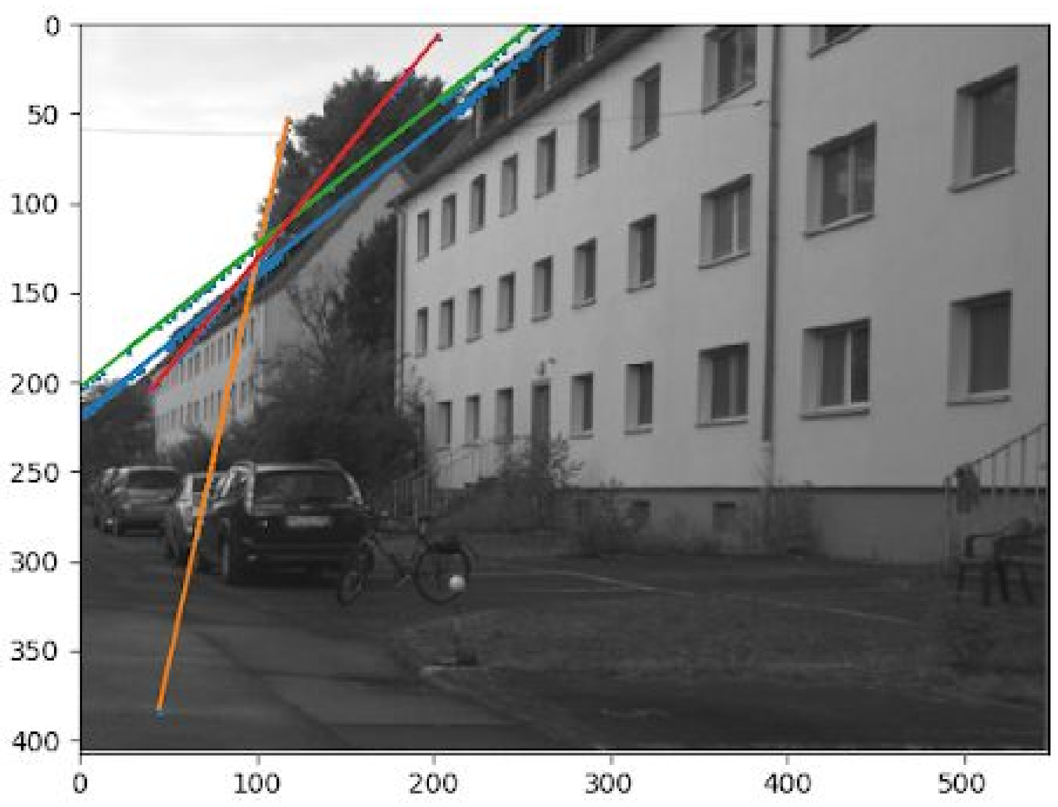
If our inlier count is greater than the inlier threshold then we want to fit a new line with all the inliers.

```
        if inlier_c >= inlier_threshold:
            m, b = np.polyfit(x, y, 1)
            li.append((inlier_c, (m,b), (x,y)))
```

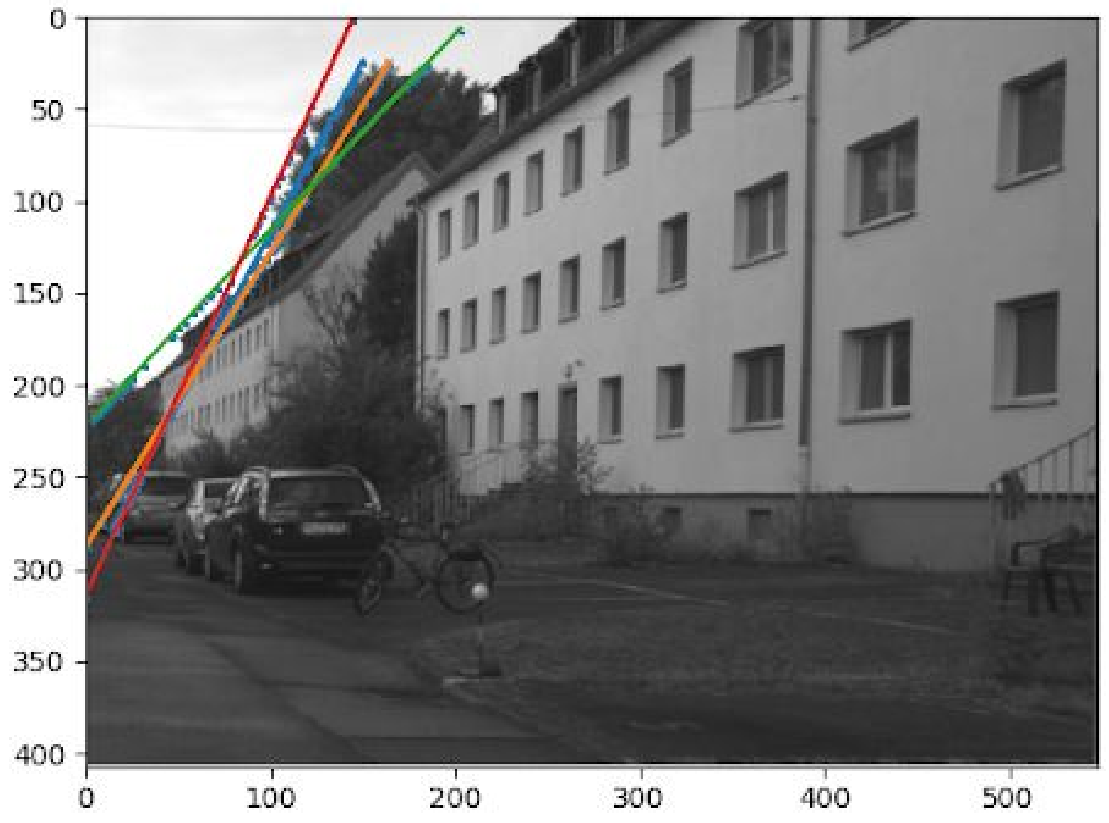
Finally we sort our list and take our four strongest lines. Then plot those lines on the image

```
for i in range(4 if len(li) >= 4 else len(li)):
        x = li[i][2][0]
        m = li[i][1][0]
        b = li[i][1][1]
        y = li[i][2][1]
        for j in range(len(x)):
            plt.gca().add_patch(plp.Rectangle((x[j]-1, y[j]+1), 3, 3,
linewidth=1))
        plt.plot(x, x*m+b)
    plt.savefig('ransac.png')
    plt.show()
```

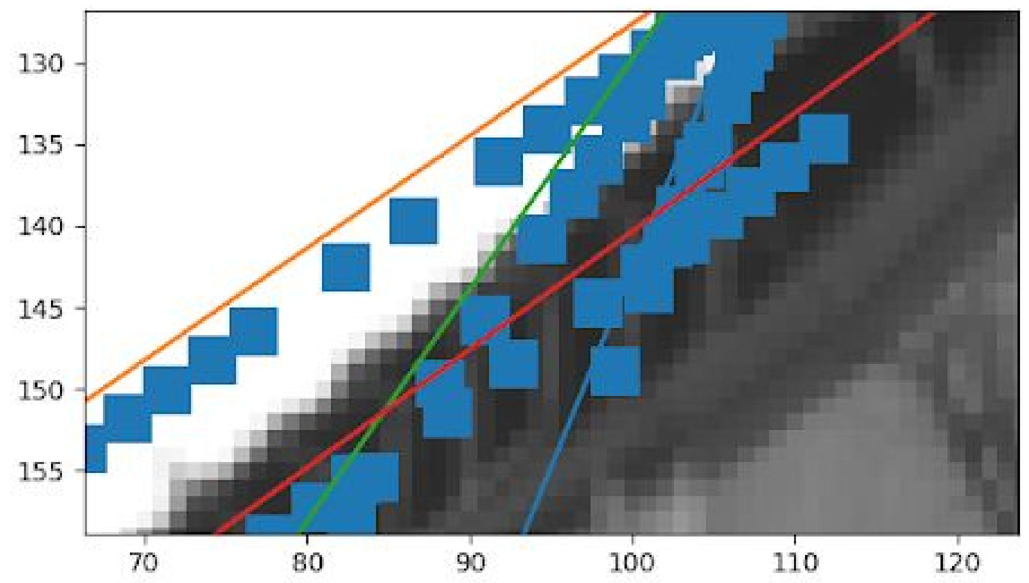
The results look as follows.



Since it is probability we obtain a different result each time. Having similar results to the last test. Below is an example of their similarities.

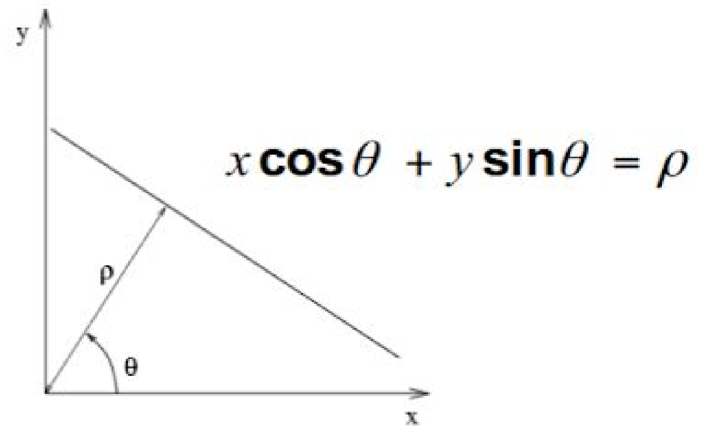


All inliers were plotted as 3 by 3 squares as shown below.



Problem 3

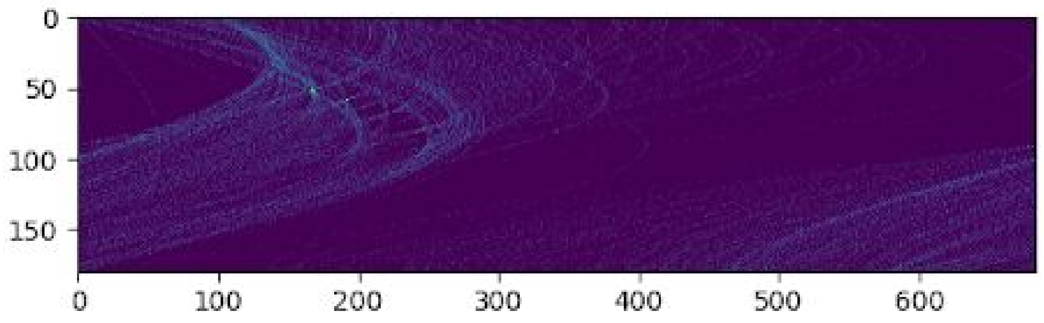
The main parameters of the hough transform function were the bins of the accumulator. However, the case arises that the accumulator needs to be built. The main function of the accumulator was to find a rho and theta that fits the model...



This function will then generate a vote in each bin of the accumulator.

```
def accumulator(img, points):

    height = img.shape[0]
    width = img.shape[1]
    diag_len = int(round(math.sqrt(width * width + height * height)))
    thetas = np.deg2rad(np.arange(0.0, 180.0, 1))

    accumulator = np.zeros((len(thetas), diag_len), np.float64)

    for i in range(len(points[0])):
        y = points[0][i]
        x = points[1][i]

        for j in range(len(thetas)):
            rho = int(x * math.cos(thetas[j]) + y * math.sin(thetas[j]))
            accumulator[j][rho] += 1
    plt.imshow(accumulator)
    plt.savefig('accumulator.png')
    plt.show()

    return accumulator
```

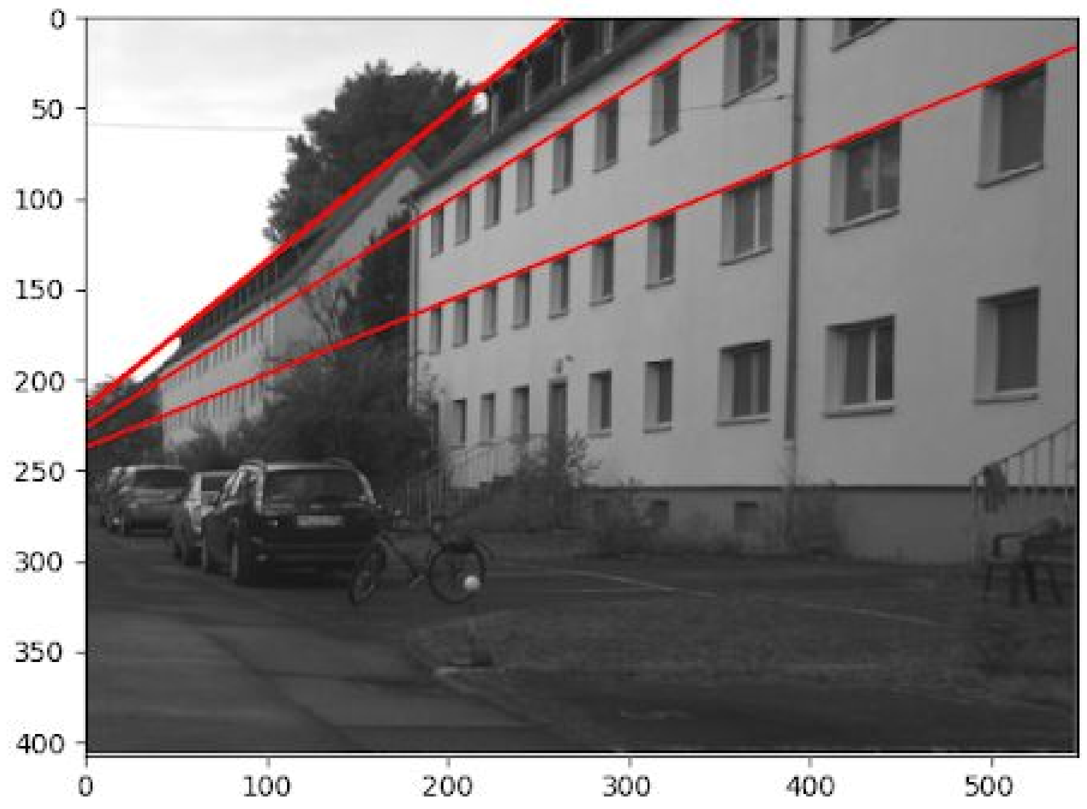
Below is an example of the accumulator plotted. Where y the theta and the x is the rho

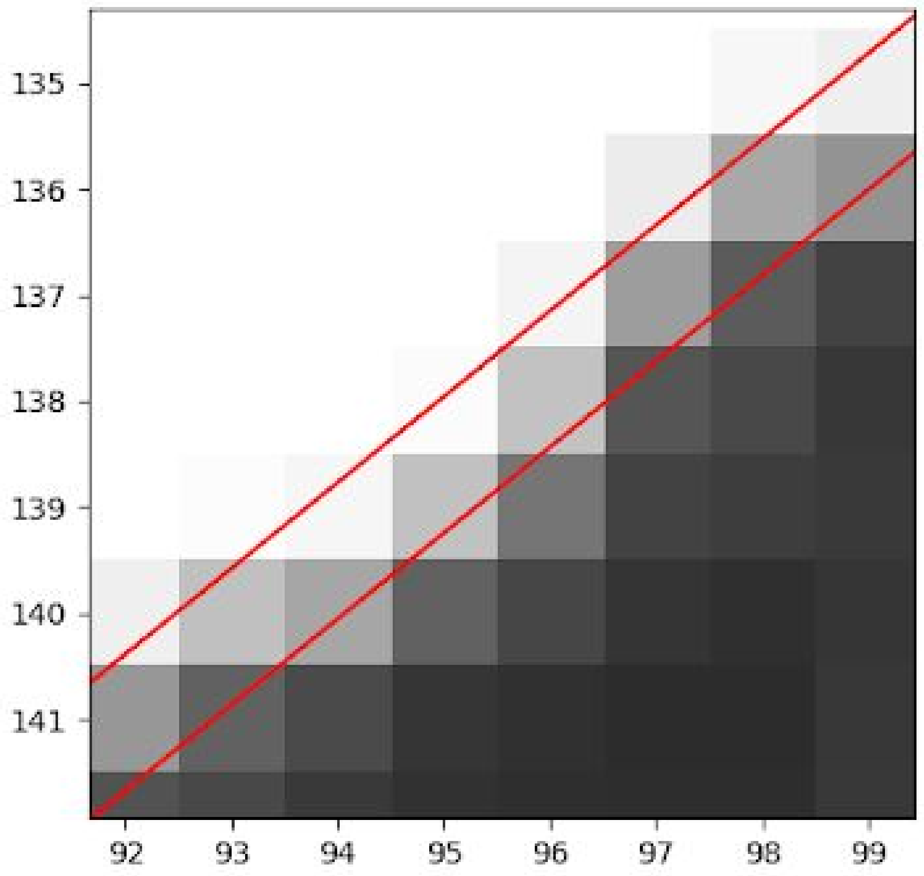


The hough transform will then take in the bins of the accumulator to produce a model for the four most voted bins. This is done by taking in a copy of the accumulator and finding the most voted bin and the setting it equal to 0. This is repeated four times. Then for each of the theta and rhos we will plot a linear function accordingly. Where the slope is the inverse of the slope of the polar representation.

```
def hough(img, accumulator):
    plt.imshow(img)

    height = img.shape[0]
    width = img.shape[1]

    max_idx = []
    copy = accumulator.copy()
    for i in range(4):
        idx = unravel_index(copy.argmax(), copy.shape)
        max_idx.append(idx)
        copy[max_idx[i]] = 0
    for theta, rho in max_idx:
        x = rho * np.cos(theta*math.pi/180)
        y = rho * np.sin(theta*math.pi/180)
        m = -x/y

        x1 = np.linspace(0, width, 2)
        b = -m * x + y
        y1 = m * x1 + b

        plt.plot(x1, y1, 'r')



    plt.axis([0, len(img[0]), 0, len(img)])
    plt.gca().invert_yaxis()
    plt.savefig('hough_transform.png')
    plt.show()
```

The four strongest lines were then plotted below. Two of the lines were closely held together. A close up is demonstrated for proof.




135
136
137
138
139
140
141
92
93
94
95
96
97
98
99