Ryan O’Shea
Spring 2020

CPE 558 HW1: Line Detection

Code:

```
"""
Ryan O'Shea

Description: Computer Vision Homework 1 Line detection.
1. Apply a guassian blur to the image then extract key points using Hessian detection
2. Use the RANSAC algorithm to find 4 straight lines in the extracted key points with the most
support in the image
3. Apply a Hough transform to the extracted points to detect 4 straight lines with the
strongest support in the image

I pledge my honor that I have abided by the Stevens Honor System

"""

import cv2 # Image loading, writing, and displaying tools
import numpy as np # Matrix functions
import math
import random
import time

# Function to apply a filter to an image
def convolute(img, filt):
# Get offset of starting position based on filter
offset = int(math.floor(filt.shape[0])/2)

# Matrix to store result of filtering in
h = np.zeros((img.shape[0], img.shape[1]))

# Loop through the pixels in the image and apply the filter
for i in range(0, img.shape[0]):
for j in range(0, img.shape[1]):
# If the filter will be out of bounds just take the original pixel value
if(i < offset or j < offset or (i + offset > img.shape[0] - 1) or (j + offset > img.shape[1] - 1)):
h[i][j] = img[i][j]
# The filter and image perfectly overlap so apply the filter
else:
sum = 0.0
# Loop through overlapping pixels on image and filter
for k in range(0, filt.shape[0]):
for g in range(0, filt.shape[1]):
sum += filt[k][g] * img[i + k - offset][j + g - offset]
# Put the sum into the correct spot in the matrix
```

```
h[i][j] = sum

return h

# Gets the value at a point (x, y) in a gaussian distribution
def guassianValues(x, y, std_dev):
g = (1/(2 * math.pi * std_dev**2))
g *= math.e**(-((x**2 + y**2)/(2*std_dev**2)))

return g

# Applies a guassian filter to the input image and returns the filtered image
def gaussianFilter(img, size, std_dev):
# Conver the image to a numpy array
arr = np.array(img)

# Initialize the filter as an nxn matrix of 0s
gaus_filt = np.zeros((size, size))

# Get the index of the central point in the array. For example the center of a
# 5x5 array would be the point arr[2,2]
center = int(math.floor(size/2))

# Populate the filter with the proper values
for i in range(size):
for j in range(size):
# The x and y values to pass into the Guassian value function are the x and
gaus_filt[i][j] = guassianValues(abs(center - i), abs(center - j), std_dev)

# Apply the guassian filter to the image
filtered = convolute(arr, gaus_filt)

return filtered

# Applies a sobel filter to the input image to get the vertical edges
def sobelFilterVert(img):
sobel_y = np.array([[1, 0, -1],
[2, 0, -2],
[1, 0, -1]])

sobel_y_img = convolute(img, sobel_y)

return sobel_y_img

# Applies a sobel filter to the input image to get the horizantal edges
def sobelFilterHoriz(img):
sobel_x = np.array([[1, 2, 1],
[0, 0, 0],
[-1, -2, -1]])

sobel_x_img = convolute(img, sobel_x)

return sobel_x_img

# Calcultes the determinate of the Hessian of the passed in image and thresholds it
def getHessian(img):
# Create the output image
ouput = np.zeros((img.shape[0], img.shape[1]))

# Get the second derivatives of the image
img_xx = sobelFilterHoriz(sobelFilterHoriz(img))
img_yy = sobelFilterVert(sobelFilterVert(img))
img_xy = sobelFilterHoriz(sobelFilterVert(img))

# Get determinant of the Hessian matrix
det = np.zeros((img.shape[0], img.shape[1]))
for i in range(img.shape[0]):
for j in range(img.shape[1]):
det[i][j] = img_xx[i][j]*img_yy[i][j] - img_xy[i][j]*img_xy[i][j]

# Get max and min values to use for thresholding
max_val = np.amax(det)
min_val = np.amin(det)

val_range = max_val - min_val

# Nomalize image
for i in range(det.shape[0]):
for j in range(det.shape[1]):
det[i][j] = (det[i][j] - min_val) * (255/val_range)

# Threshold the determinant
for i in range(det.shape[0]):
for j in range(det.shape[1]):
# If the pixel value is lower than the threshold set it to 0
if det[i][j] < 150:
det[i][j] = 0
else:
det[i][j] = 255

# Apply non maximum suppression to the image
suppressed = nonMaxSuppression(det)

return suppressed
```

```
# Applies non maximum suppression to the passed in array
def nonMaxSuppression(img):
output = img

# Loop through the pixels in the image
for i in range(1, img.shape[0] - 1):
for j in range(1, img.shape[1] - 1):
# Check if the current pixel is not the largest in the surrounding 3x3 area
if img[i][j] != max(img[i-1][j-1], img[i-1][j], img[i-1][j+1], # First row
img[i][j-1], img[i][j], img[i][j+1], # Second row
img[i+1][j-1], img[i+1][j], img[i+1][j+1]): # Third row
# Pixel is not the largest so set it to 0
output[i][j] = 0
else:
# It is the max so set the rest to 0
for k in range(0, 3):
for g in range(0, 3):
if not (k == 1 and g == 1):
output[i + k -1][[j + g -1]] = 0
return output

# Function to get a list of all the pixels with values above a certain threshold
def getPoints(img, threshold):
# List of x, y pairs stored as tuples
points = []

# Loop through all pixels in image and get all coordinates of pixels with value above threshold
for i in range(0, img.shape[0]):
for j in range(0, img.shape[1]):
if img[i][j] > threshold:
points.append((j, i))

return points

# Returns two random points from the list of passed in points
def pickRandPoints(pointsList):
# Get the random index of the first point
idx1 = random.randint(0, len(pointsList) - 1)

# Pick another radom index that id different from the first one
idx2 = random.randint(0, len(pointsList) - 1)

# Ensure that the two indexes are not the same
while(idx2 == idx1):
idx2 = random.randint(0, len(pointsList) - 1)
```

```
# Get the random points
temp_point1 = pointsList[idx1]
temp_point2 = pointsList[idx2]

return temp_point1, temp_point2

# Get the slope m and intercept c of a line based on the two passed in points
def getLine(point1, point2):
# Get x1 and y1 from the correct indexes
x1 = point1[0]
y1 = point1[1]

# Get x2 and y2 from the correct indexes
x2 = point2[0]
y2 = point2[1]

# Check if x1 and x2 and the same (AKA points make a vertical line)
if(x1 == x2):
# Set slope to infinity
m = math.inf
else:
m = (y2 - y1)/(x2 - x1)

# Find the intercept c using the point slope intercept formula
# y - y1 = m(x - x1)
c = -m*x1 + y1

return m, c

# Gets the perpendicular distance between a point and a line
def getDistFromLine(m, c, x, y):
# Get the point on the line where the perpendicular line from the point to the line intersects
line_x = (x + (m*y) - (m*c))/(1 + m**2)
line_y = ((m*x) + ((m**2)*y) - ((m**2)*c))/(1 + m**2) + c

# Calculate the distance between the point on the line (line_x, line_y) and point (x, y)
dist = math.sqrt(((line_x - x)**2) + ((line_y - y)**2))

return dist, line_x, line_y

# Run the RANSAC algorithm on the passed in image to determine the 4 best lines in the
image
def RANSAC(img, norm_img, num_lines, num_points):
# Get the list of points to pick from
points = getPoints(img, 0)

# Variable to keep track of how many lines we've found
```

```
lines = 0

while(lines < num_lines):
# Get 2 random points from the list
point1, point2 = pickRandPoints(points)

# Get the model for the line between the two points
m, c = getLine(point1, point2)

# List to hold all the points that are close enough to line model
inliers = []

# Variable for the largest distance a line can have based on its inliers
max_line_size = 0

# Loop through all the points to determine inliers
for point in points:
# Get distance between line and point
dist, line_x, line_y = getDistFromLine(m, c, point[0], point[1])

# Check if the point is close enough to the line
if dist < 3:
# Add the point to the array
inliers.append((point[0], point[1]))

# Check if the number of inliers for the line is above our specified needed amount
if(len(inliers) > num_points):
# Increase number of good fit lines we've found
lines += 1

# Remove the inliers that were used to prevent reuse
for point in inliers:
# Remove the point
points.remove((point[0], point[1]))

# Plot the inliers as 3x3 squares
for i in range(0, 3):
for j in range(0, 3):
# Check if the pixel is out of bounds
if ((point[0] + i - 1) > img.shape[0]) or ((point[1] + j - 1) > img.shape[1]):
continue
else:
img[point[1] + j - 1][point[0] + i - 1] = 255

# Loop through the rest of the points to find the two points with
# Largest distance betwwen them
for secondary_point in inliers:
```

```
# Calculate the distance between the two points
point_dist = math.sqrt(((point[0] - secondary_point[0])**2) + ((point[1] -
secondary_point[1])**2))

# Determine if the line made by the two points is bigger
if point_dist > max_line_size:
max_line_size = point_dist

# Variable to hold the two points that make the largest line for a set of inliers
most_dist_points = ((point[0], point[1]), (secondary_point[0], secondary_point[1]))

# Plot the line from one most distant inlier to the other on the image with points
cv2.line(img, most_dist_points[0], most_dist_points[1], (255, 255, 255), thickness=1)

# Plot the line from one most distant inlier to the other on the normal image
cv2.line(norm_img, most_dist_points[0], most_dist_points[1], (0, 0, 0), thickness=2)
# Once the four strongest lines have been found show them on the image
if lines == 4:
cv2.imshow("RANSAC point image", img)
key = cv2.waitKey(0)

if key == ord('q'):
exit()

cv2.imshow("RANSAC Normal image", norm_img)
key = cv2.waitKey(0)

if key == ord('q'):
exit()

# Applies a Hough transform to the passed in image in order to find the 4 strongest supported
lines
def HoughTran(img, norm_img, num_lines):
# Get number of rows and columns
rows = img.shape[0]
cols = img.shape[1]

# Maximum and minimum values that rho can
# According to equation ρ = x cos θ + y sin θ
max_val = rows + cols
min_val = -cols

# Because we can't directly plot negative rho values we need to remap the range of rho
# to be 0 to range and then use an offset of the highest negative value to properly
# get the indexes of points
offset = -min_val
```

```
val_range = max_val - min_val

# Accumulator
accum = np.zeros((val_range, 181))

# Get the feature points from the image
points = getPoints(img, 0)

# Loop through all the points
for point in points:
x = point[0]
y = point[1]

# Loop through all the angles
for angle in range(0, 181):
# Get angle in radians
rads = math.radians(angle)
# Calculate rho
rho = int(x*math.cos(rads) + y*math.sin(rads) + offset)

# Add votes to the rho, theta pair in the accumulator
accum[rho][angle] += 20 # Add 20 instead of 1 so it displays nicely

# Show image
cv2.imshow("Accumulator", accum/255)
cv2.waitKey(0)

# Current number of lines drawn
lines = 0

# Highest value in the hough transform
highest = 0

# Apply non maximum suppression to the acumulator
suppressed = nonMaxSuppression(accum)
cv2.imshow("Suppresed Accumulator", suppressed/255)
cv2.waitKey(0)

# Loop until we've plotted the desired number of lines
while lines < num_lines:
# Find the max value in the hough transform which should correclate to the parameters of the
strongest line
for i in range(suppressed.shape[0]):
for j in range(suppressed.shape[1]):
if suppressed[i][j] > highest:
max_params = (j, i) # Store in form (theta, rho)
highest = suppressed[i][j]
```

```
# Reset highest value back to 0
highest = 0

# Set the pixels around the chosen model equal to 0
for i in range(-10, 10):
for j in range(-10, 10):
suppressed[max_params[1] + i][max_params[0] + j] = 0

# Get theta and rho from max_params
theta = math.radians(max_params[0])
rho = max_params[1] - offset # Undo the offset we added to get the true value of rho

# Get line parameters
a = math.cos(theta)
b = math.sin(theta)
x0 = a*rho
y0 = b*rho

# Generate two points that will span the image using the line parameters
pt1 = (int(x0 + 1000*(-b)), int(y0 + 1000*(a)))
pt2 = (int(x0 - 1000*(-b)), int(y0 - 1000*(a)))

# Draw the line on the point image
cv2.line(img, pt1, pt2, (255, 255, 255), thickness=1)

# Draw the line on the normal image
cv2.line(norm_img, pt1, pt2, (0, 0, 0), thickness=3)

lines += 1

cv2.imshow("Hough Lines", img)
cv2.waitKey(0)

cv2.imshow("Hough Lines Original image", norm_img)
cv2.waitKey(0)

if __name__ == "__main__":
# Initialize the random seed
random.seed(time.time())
# Get the image
img = cv2.imread("road.png")

# Convert to graysale
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

cv2.imshow("Input Image", img)
cv2.waitKey(0)
```

```
# Apply a 5x5 Guassian filter to the image
guas_img = gaussianFilter(img, 5, 1)

# Get Key points in the image using a Hessian detector
pot_points = getHessian(guas_img)

# Show the key points
cv2.imshow("Supressed key points", pot_points.copy())
cv2.waitKey(0)

# Run the RANSAC algorithm on the key points to find the 4 best lines with at least 40 inliers
RANSAC(pot_points,copy(), img.copy(), 4, 40)

# Apply a Hough transform to the image to get the 4 best lines
HoughTran(pot_points.copy(), img.copy(), 4)
```

<u>Code Explanation and Results:</u>
The code reads in the image from the provided file, road.png, converts it to grayscale and then begins the pre-processing part of the code.

<u>Pre-Processing</u>
The pre-processing starts by first creating a Gaussian filter using the Guassian value equation provided in the slides. I found that a 5x5 Guassian filter worked well for this purpose. The filter is applied to the image using the sliding window technique. The result of applying the filter to the image can be seen below in Figure 1.

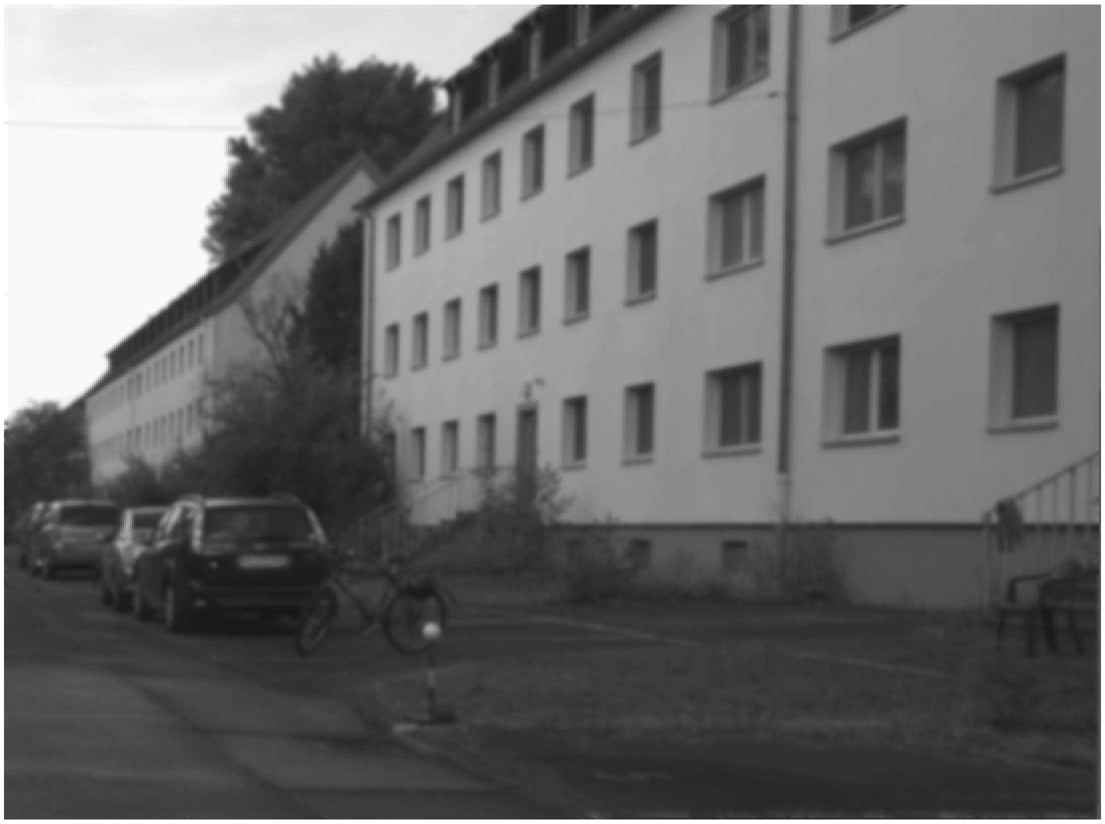
Figure 1: Guassian Filtered Image

Sobel filters are then applied to the image as derivatives operators in order to find the second x, second y, and x of y derivatives of the image. These derivative matrices are used to then calculate the determinant of the Hessian matrix. The determinant of the Hessian matrix was then thresholded by setting all pixels with values below 150 to 0 and all pixels with values above 150 to 255. The result was showed all the areas on the image where the Hessian detector responds strongly. Non maximum suppression was then applied to this result in order to isolate representatives of the best Hessian detector responses. The thresholded Hessian detector response and non maximum suppressed version can be seen below in Figure 2. The Hessian detector responds strongly to many of the corners and straight lines that make up the features of the building which is the ideal behavior bur it also responds strongly to the tree in the upper left corner of the image. While not ideal this is expected because there is a very large intensity difference between the dark tree and the very light sky.

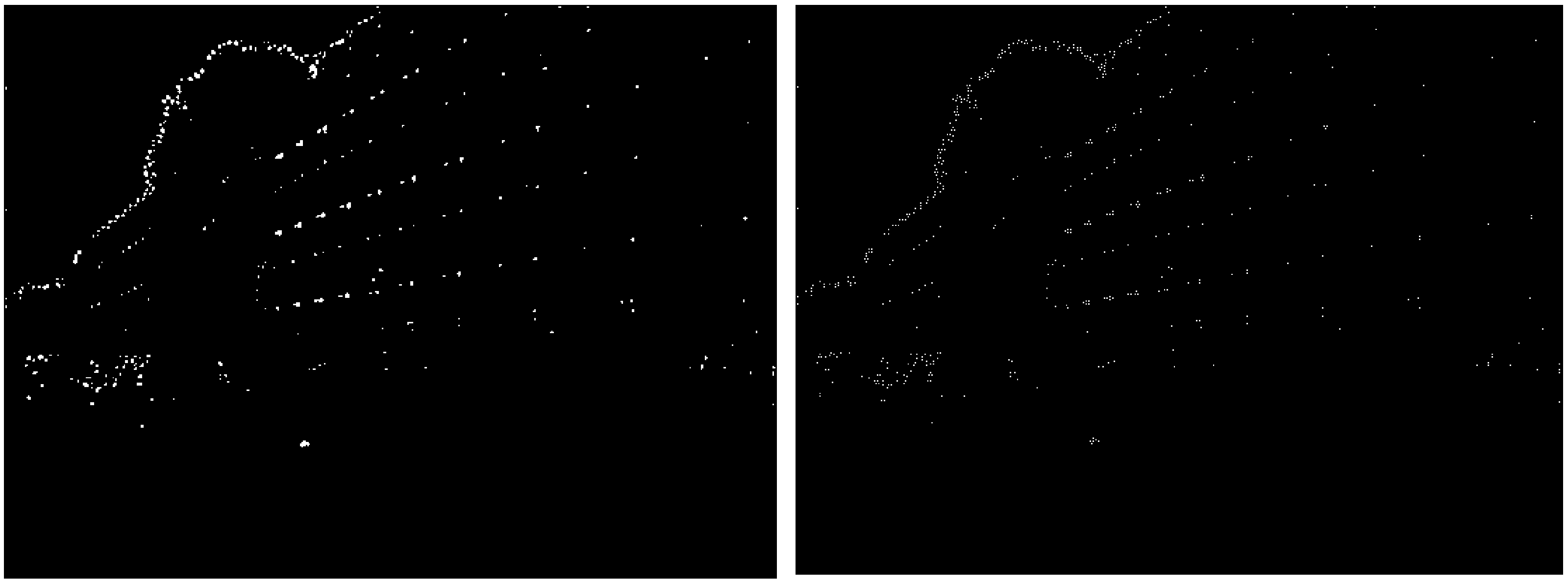
Figure 2: Thresholded Hessian Response (Left) and Non Maximum Suppressed Response (Right)

<u>RANSAC:</u>

The RANSAC portion of the code takes in the output of pre-processing step which should be an array of all the areas on the image where the Hessian detector responded very strongly after non maximum suppression was applied. The algorithm first gets all the points in the image where the detector responded strongly and stores them as tuples in list that can be easily accessed later. Two points are then drawn at random from the list and the parameter for the equation of a line, slope m and y intercept c, are found using some basic algebra. The rest of the points are then looped through in order to determine the points that are close enough to the line to be inliers for the line. This is done by calculating the perpendicular distance between the point and the line and checking if it is below a certain threshold. An example of this process can be seen below in Figure 3 where a line was drawn between the proposed line and every point that is within a distance of 100. Through trial and error I found that a distance threshold of 3 produced the best results.

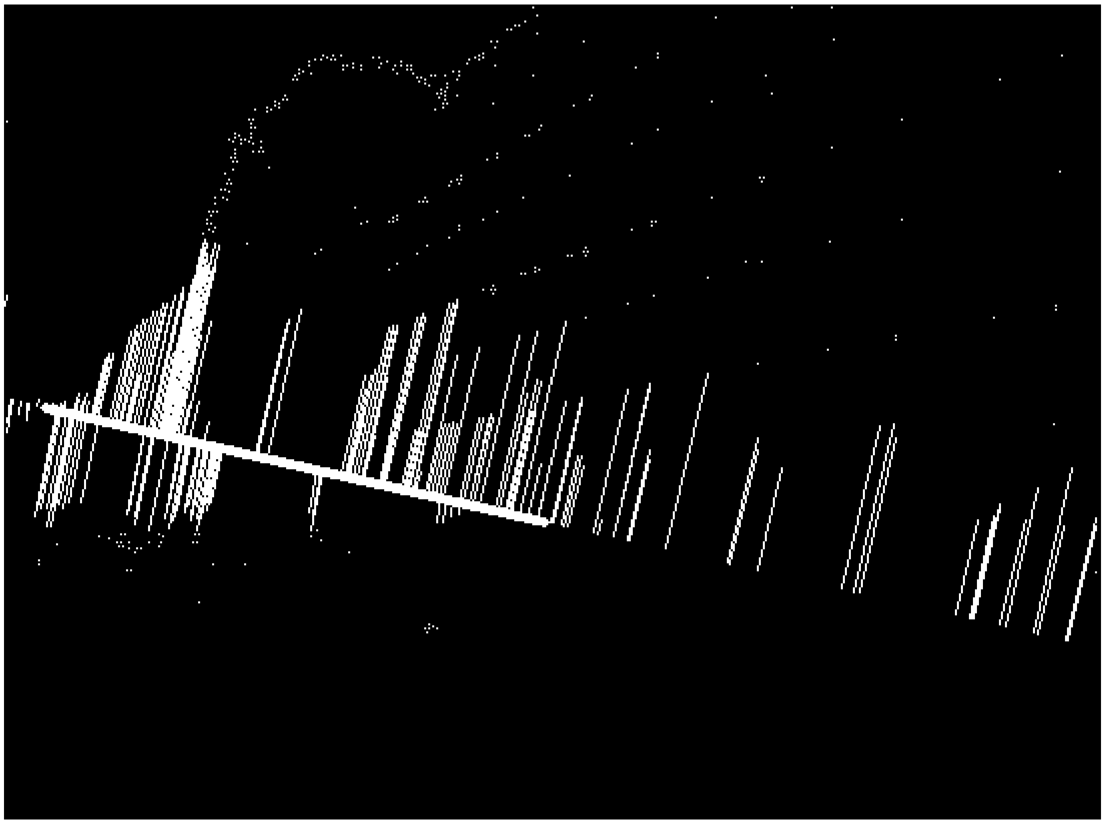
Figure 3: Distance Calculation Between Line and Points

Once all the points have been checked the algorithm checks if the number of inliers found is above the specified threshold. If the number is high enough then the line and inliers will be plotted on the image and the inliers will be removed from the list of inliers in order to prevent repeat lines to be proposed. Through more trial and error I found that approximately 40 inliers produced the best results when combined with the maximum distance from the line of 3. Once a line is accepted all of its inliers are checked in order to find the two farthest inliers from each other who will in turn produce the longest line. The final line is drawn onto the image using these two inliers. This whole process is repeated until 4 lines that meet all the requirements have been accepted and drawn on the image. The results of this process can be seen below in Figure 4.

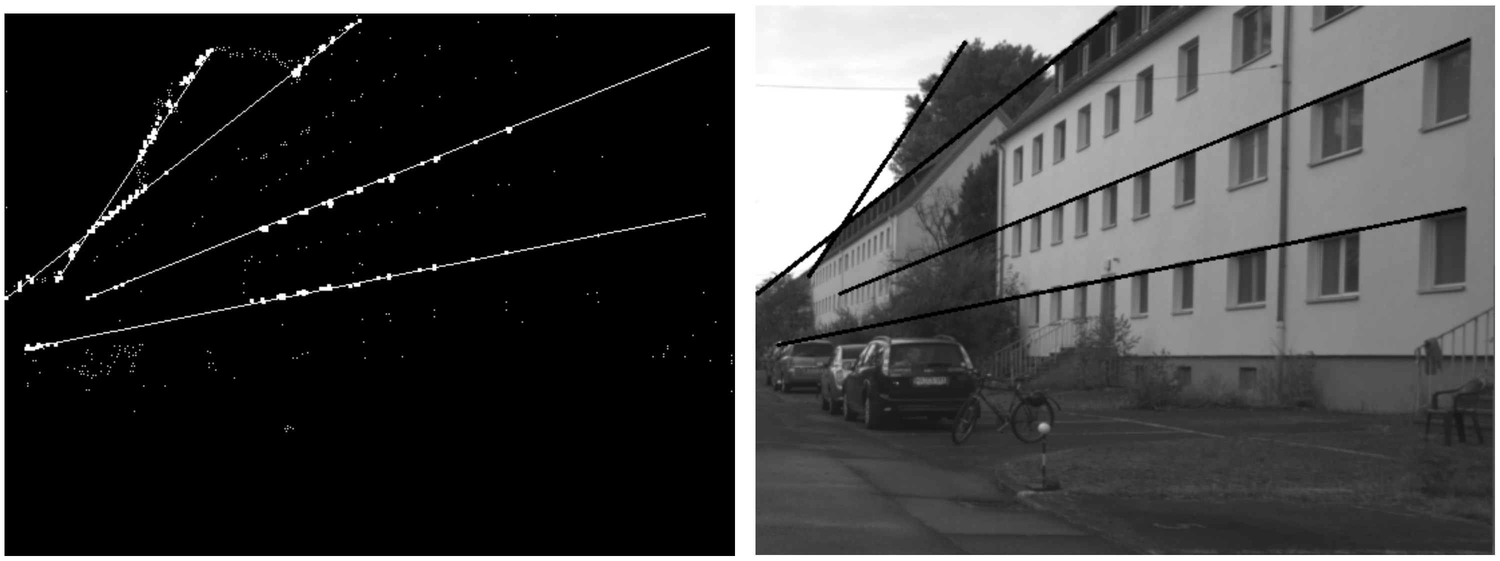
Figure 4: Four Best Lines Produced by RANSAC Plotted

Because RANSAC works on random;y drawn point, the algorithm will occasionally produce results different than the ones shown above. The results are not drastically different but the algorithm might find a different strong fitting line than the 4 that are shown above. There are also occasions when points that look like they should fit a drawn line are not considered part of the lines inliers. This could be because the points are actually just outside the maximum allowed distance for inliers or there is a bug somewhere in the code but most of the time the lines proposed by the algorithm work very well.

Hough Transform:
Like RANSAC, the Hough transform portion of the code takes in the output of pre-processing step which should be an array of all the areas on the image where the Hessian detector responded very strongly after non maximum suppression was applied. The algorithm first determines the dimensions it will need to make the accumulator. To do this, the highest and lowest values for the function $\rho = x \cos \theta + y \sin \theta$ are determined. The high and low values are used to determine the total range of rho values which is used as one dimensions of the the accumulator while the value 181 is used for the theta dimension of the accumulator. The range of values is used for the first dimension because we cannot directly represent the negative values that rho will take on in the accumulator. To offset this we use an offset which is just the negative of the low value which is added to the rho produced by the equation to create a proper index to use. The algorithm gets all the points in the image where the detector responded strongly and stores them as tuples in list that can be easily accessed later. The points are then looped through and rho is found for each angle between 0 and 180 according to equation $\rho = x \cos \theta + y \sin \theta$. Rho and theta are used as indexes to increment the value of the bin of the accumulator that they correspond to. Each combination of rho and theta corresponds to a line model in the image space which allows each point to vote for 180 different models that the point could possibly belong to. Points that fall on the same line should all vote for the same or very similar lines which will show up as overlapping bright spots on the visualization of the Hough space below in Figure 5. The point in the Hough space with the highest value should correspond to the line in the image space with the most support based on the number of image space points that voted for it. Before finding the points with the most support non maximum suppression is applied to the Hough space in order to remove near identical lines from being proposed. Once the strongest point in suppressed Hough space is found the model is converted back in two linear form and two points are generated in order to draw a line between on the image. The model in Hough space, as well as the models around it in a 10x10 grid are set to 0 in order to further prevent duplicates from being proposed,

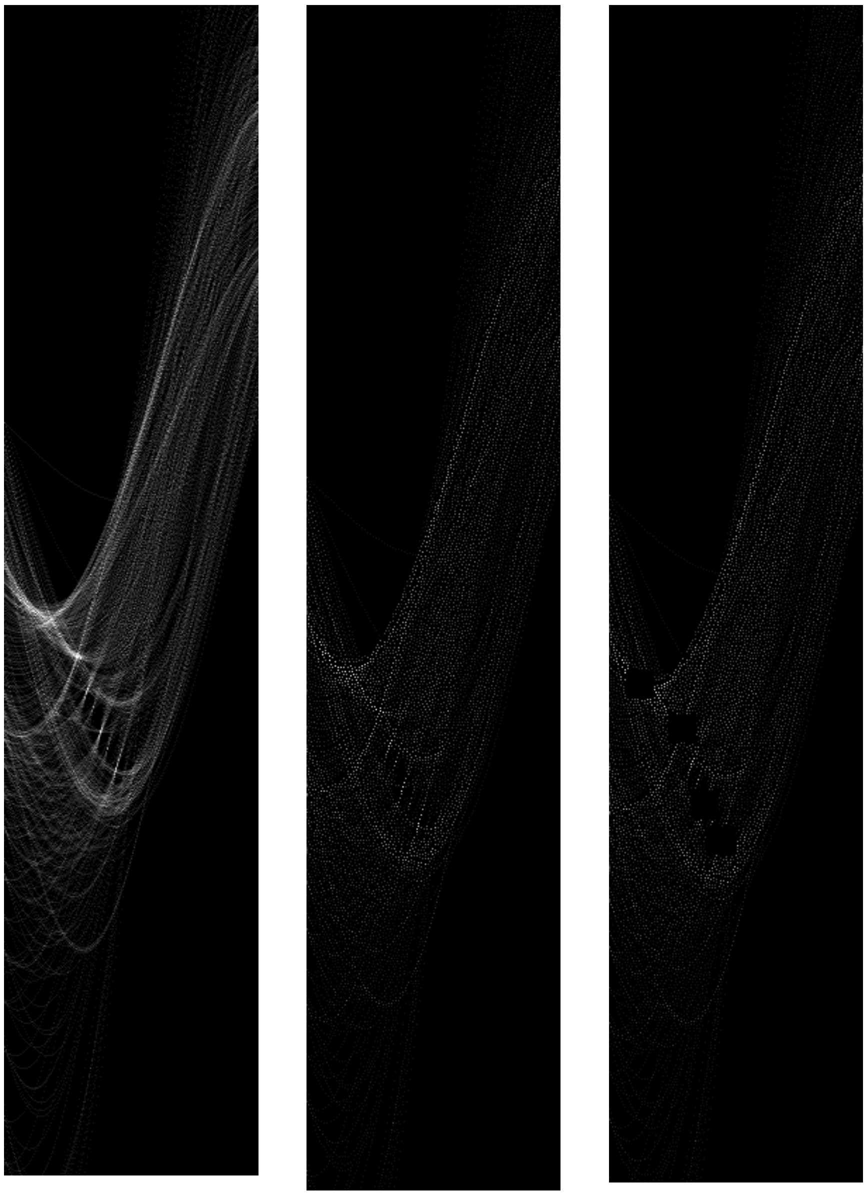
Figure 5: Visualized Hough Space (Left), Suppressed Hough Space (Center), Areas Removed From Hough Space (Right)

The 4 models with the highest number of votes are chosen and drawn on the image as shown below in Figure 6. Without the large amount of model removal from the Hough space after model proposal the algorithm proposed multiple models that use the large amount of detected features from the tree in the upper left corner of the image. The 4 lines can also be seen overlaid on the input image in Figure 7 below. With the many model removal method the Hough transform produces very similar results to RANSAC but extends the lines cross the entire image without the need to connect the two farthest inliers. Unlike RANSAC. The Hough transform is not random and will therefore produce the same results every time.

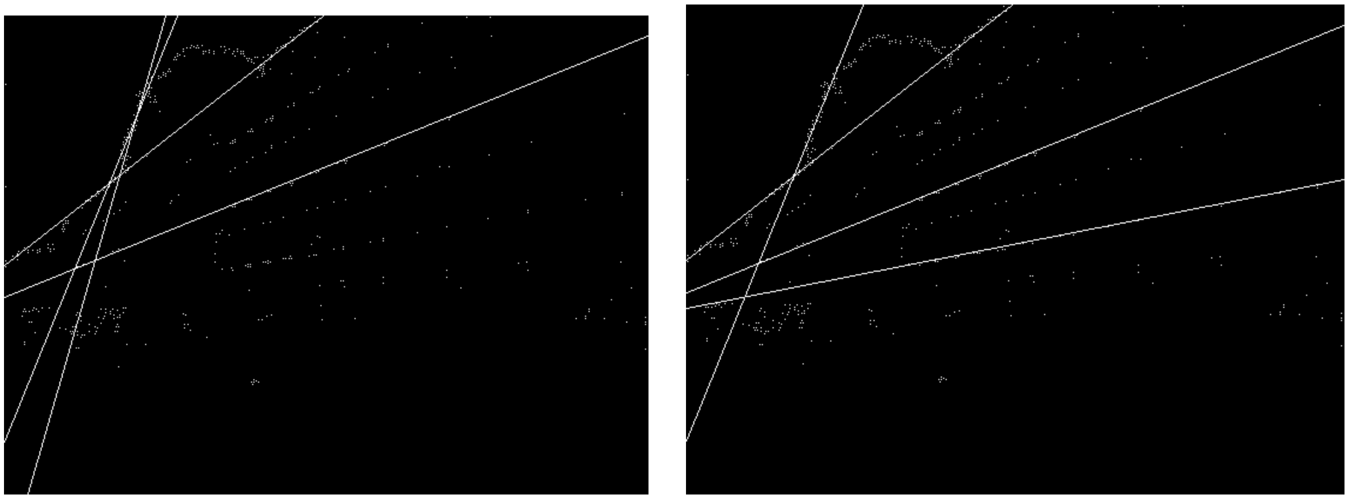

Figure 6: Hough Lines Produced with 1 Model Removal (Left), Hough Lines Produced with Many Model Removal (Right)

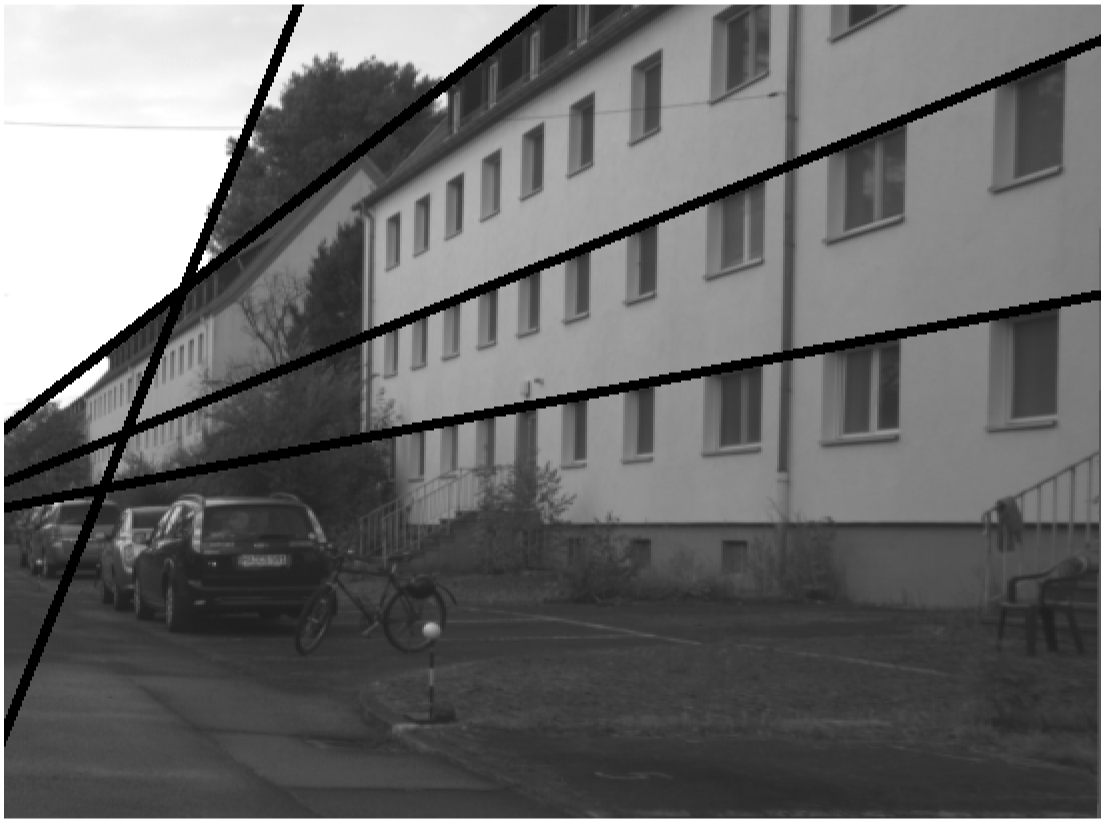

Figure 7: Hough Lines on Original Image