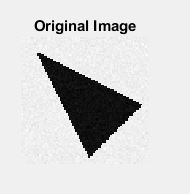
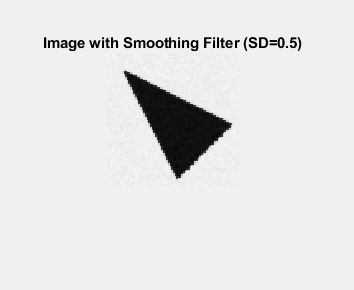
3.

a.

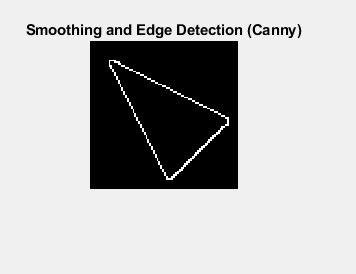
Below is the original image.



After applying a gaussian filter with standard deviation=0.5. Applying with higher filters makes edges too blurry. Applying it with smaller standard deviation makes the noise too high.



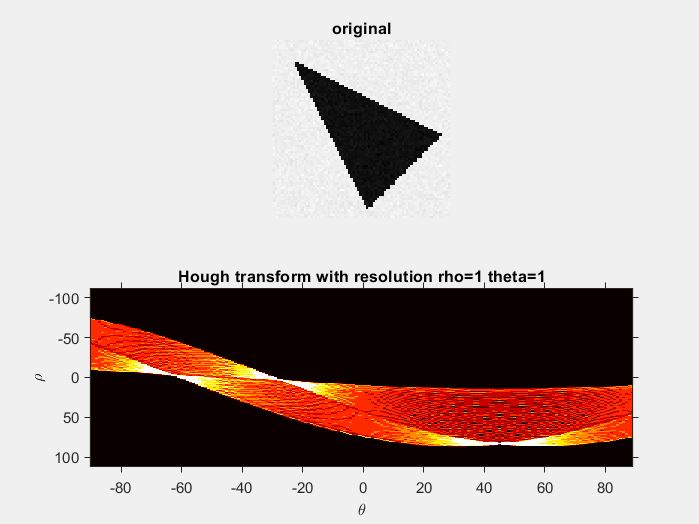
Using the Canny method to detect the edges. The edges are very well defined.



b.

Applying the Hough transform, choosing rho and theta resolution = 1 so as to create very local areas of high intensity voting and at the same time not too small of a resolution to make the location of the actual highest center of voting ambiguous.

There are 3 areas of significant voting.



Analyzing this Hough graph, we see that the highest values, index pairs are

|  |  |  |
| --- | --- | --- |
| Value (number of votes) | Coordinates (60[col] by 77[row]) | (theta, rho) |
| 68 | 731 = 10, 38 | (-60,0) |
| 67 | 1657 = 22, 40 | (-24,6) |
| **62** | 653 = 9, 37 | (-63,-3) |
| **62** | 1735 = 23, 41 | (-21,9) |
| **62** | 3532 = 46, 67 | (47,87) |

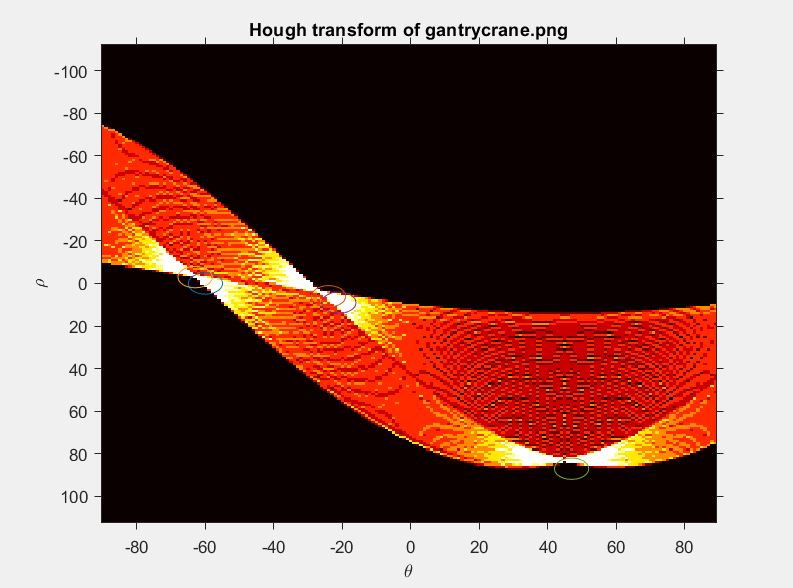
Note that in the Hough transform matrix the 3rd highest value appeared 3 times. Two of these appearances are mere repeats of the 1st and 2nd highest theta,rho values.

To convert from (col, row) to (theta, rho),

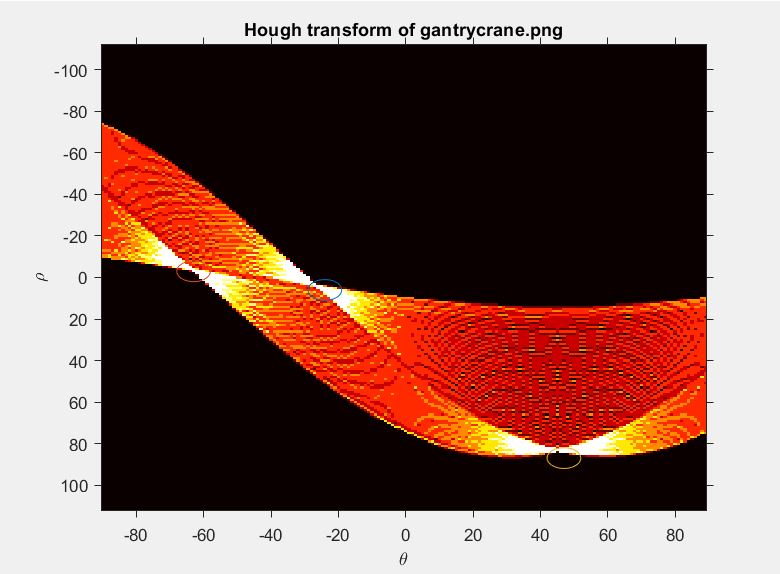
Theta = -90 + ((89+90)/60)\*col

Rho = -114 + ((114+117)/77)\*row

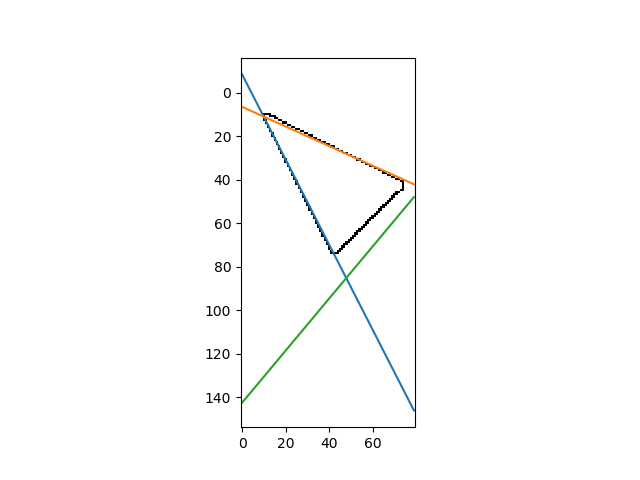
Plotting and inspecting each (theta,rho) value on the graph and choosing the best fit,



We choose (-63,-3), (-24,6), and (47,87) as the final 3 pairs.



The plots of the 3 lines are below.



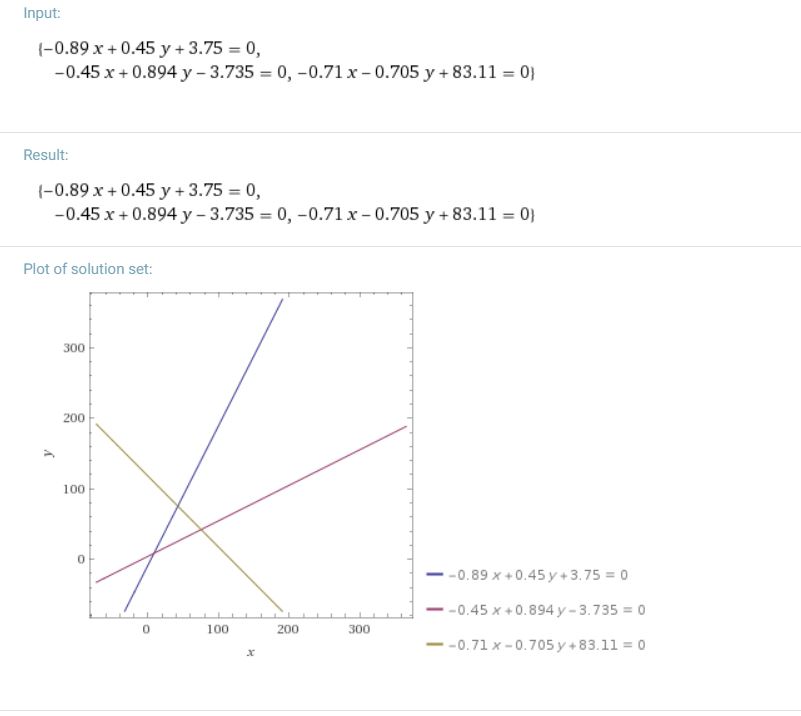
c.

We apply k-means with lines as representatives as discussed in class

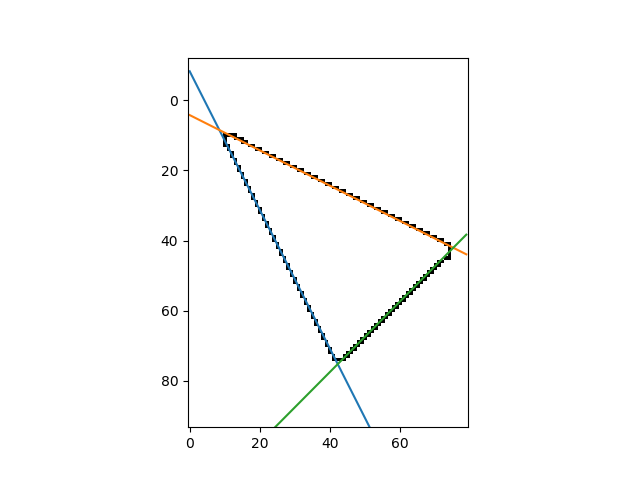
1. We process the image to generate points for the edges
2. We randomly assign edges to lines
3. Create the best fit line through each of its points
4. Check for terminating condition (I used if the number of points in each bucket stayed the same after so many iterations)
5. GoTo step 2

The source code in Python is attached or can be found on github.com at “https://github.com/kelvinL3/kMeans-Fitting-Lines-to-Points”

This method converged to the same solution most of the time. This solution was graphed on Wolfram Alpha:



Plotted in matplotlib, with the edge data, it looks very consistent:



import scipy.io as sio

import matplotlib.pyplot as plt

import numpy as np

from numpy import linalg as LA

import math

import sys

# takes in a matrix representation of the edges of an image

# edges are marked with 1s, 0s everywhere else

# returns a list of the pixels where the edges are

def preprocessing():

# Il = sio.loadmat('figure2-1.mat')

Il = sio.loadmat('edges.mat')

I = Il['bw05']

# print(I)

# plt.matshow(I)

plt.imshow(I, cmap='Greys')

# plt.show()

I = np.array(I)

# turn everything into points

points = []

it = np.nditer(I, flags=['multi\_index'])

while not it.finished:

if it[0].item(0) is not 0:

points.append(Point(it.multi\_index[0],it.multi\_index[1]))

it.iternext()

return points

# represents a pixel on the edge

class Point:

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

def \_\_str\_\_(self):

return "("+str(self.x)+" ,"+str(self.y)+")"

\_\_repr\_\_ = \_\_str\_\_

# class for assigning the points to the lines

class P1:

@staticmethod

def assign\_points\_to\_lines(points, lines, buckets):

for p in points:

buckets[P1.closest\_distance(p, lines)].append(p)

@staticmethod

def closest\_distance(p, lines):

min\_dist = sys.maxsize

min\_ind = 0

# print("\nnew")

for i in range(len(lines)):

dis = P1.distance(p, lines[i])

# print(dis)

if dis < min\_dist:

# print("decrease")

min\_dist = dis

min\_ind = i

# print("this is always 2?? ", i)

return min\_ind

@staticmethod

def distance(p, line):

x = math.fabs(p.x\*line[0] + p.y\*line[1] + line[2]) / math.sqrt(line[0]\*\*2 + line[1]\*\*2)

# print("distance is", x, " ", p.x, line[0], p.y, line[1], line[2])

return x

# class for changing the lines to fit the points

class P2:

@staticmethod

def update\_lines(lines, buckets):

# for line,bucket in lines,buckets:

for i in range(3):

line = lines[i]

bucket = buckets[i]

if len(bucket) is 0:

continue

x\_mean = P2.mean(bucket,0)

y\_mean = P2.mean(bucket,1)

A = P2.mean\_two(bucket, 0,0) - x\_mean\*\*2

B = P2.mean\_two(bucket, 0,1) - x\_mean\*y\_mean

C = P2.mean\_two(bucket, 1,1) - y\_mean\*\*2

w,v = LA.eig(np.array([[A,B],[B,C]]))

if w[0]<=0 or w[1]<=0:

print("SOMETHING WRONG\n", A, B, C)

import sys

sys.exit(0)

# print("Eigenvector:", v)

if w[0]<w[1]:

# print(str(v[:,0][0]) + " "+str(v[:,0][1]))

line[0] = v[:,0][0]

line[1] = v[:,0][1]

line[2] = - line[0]\*x\_mean - line[1]\*y\_mean

else:

line[0] = v[:,1][0]

line[1] = v[:,1][1]

line[2] = - line[0]\*x\_mean - line[1]\*y\_mean

@staticmethod

def mean(bucket, index):

ans = 0.0

length = 0

for point in bucket:

p = (point.x,point.y)

ans += p[index]

length += 1

return ans/length

def mean\_two(bucket, i1, i2):

ans = 0.0

length = 0

for point in bucket:

p = (point.x,point.y)

ans += p[i1]\*p[i2]

length += 1

return ans/length

def delete\_buckets(buckets):

del buckets[0][:]

del buckets[1][:]

del buckets[2][:]

points = preprocessing()

# print(points)

# initialize

lines = [[0 for x in range(3)] for y in range(3)]

# these lines arent really necessary, but just helps show that the equations of the lines are indeed moving

for i in range(3):

lines[i][0] = np.random.random()

lines[i][1] = np.random.random()

lines[i][2] = np.random.random()

buckets = [[] for i in range(3)]

# randomly assign points

for p in points:

x = np.random.randint(3)

buckets[x].append(p)

prev = [0]\*3

converge\_times = 20

iterations = 0

print("\nInitial Lines Configuration")

print(lines[0])

print(lines[1])

print(lines[2])

# initialize the 3 lines

while True:

for i in range(100):

# update the lines to the points

P2.update\_lines(lines, buckets)

# check if the lines didnt change that much / the lines the points are assigned to didnt change at all

# delete buckets

delete\_buckets(buckets)

# assign the points to the lines

P1.assign\_points\_to\_lines(points, lines, buckets)

if prev[0] is len(buckets[0]) and prev[1] is len(buckets[1]) and prev[2] is len(buckets[2]):

# print("Converge???")

converge\_times-=1

if converge\_times is 0:

print("FINAL CONVERGE")

break

# print("length of buckets", len(buckets[0]), len(buckets[1]), len(buckets[2]))

prev[0] = len(buckets[0])

prev[1] = len(buckets[1])

prev[2] = len(buckets[2])

iterations += 1

break

print("\nTook", iterations, "Iterations")

print(lines[0])

print(lines[1])

print(lines[2])

# # for graphing

# def graph(function):

# x = np.array(range(0,80))

# y = eval(function)

# plt.plot(x, y)

# graph('-3.75/0.45 + (0.89/0.45)\*x')

# graph('3.735/0.894 + (0.45/0.894)\*x')

# graph('83.11/0.705 + (- 0.71/0.705)\*x')

# graph('-3.75/0.45 + (0.88/0.45)\*x')

# graph('6/0.91 + (0.41/0.91)\*x')

# graph('87/0.61 - (0.73/0.61)\*x')

# plt.show()