READ ME

Requires Python PILLOW and PyPNG to run.

Takes as input an image "road.pgm" (must be pgm format, I converted on this site: https://convertio.co/png-pgm/). Image must be present in the same directory as the __init__.py file.

Also will ask for 4 arguments as input: the RANSAC distance threshold, the RANSAC inlier threshold, the Hough transform theta accuracy, and the Hough transform rho accuracy. For my output images, I used values of 1.2, 60, 1.5, and 1 respectively for each parameter.

The program will output 5 images: the hessian version of the road image, the ransac keypoints, the ransac lines plotted on the hessian image, the hough keypoints, and the hough lines plotted on the hessian image.

```
import math
import copy
import sys
from copy import deepcopy
import png
import random
from math import atan2
from math import atan
from PIL import Image, ImageDraw
from PIL.ImageQt import rgb

img_width = 0
img_height = 0

def read_pgm(img_file):
    assert img_file.readline() == b'P5\n'

    global img_width, img_height
    (img_width, img_height) = [int(i) for i in img_file.readline().split()]
    depth = int(img_file.readline())
    assert depth <= 255

    #creating raw image matrix
    img_matrix = []
    for y in range(img_height):
        row = []
        for y in range(img_width):
            row.append(ord(img_file.read(1)))
        img_matrix.append(row)
    return img_matrix

def gaussian_blur(img_width, img_height, sigma, img_matrix):
    #creating Gaussian kernel
    kernel = []
    calc1 = 1 / (2 * math.pi * sigma**2)
    calc2 = -1 / (2 * sigma**2)
    for x in range(0-sigma*3, 0+sigma*3+1):
        row = []
        for y in range(0-sigma*3, 0+sigma*3+1):
            row.append(calc1 * math.exp(calc2 * (x**2 + y**2)))
        kernel.append(row)


    padded_img = img_matrix
    # padding existing rows by copying edges
    for p in range(img_height):
        for s in range(sigma*3):
            padded_img[p].append(padded_img[p][img_width-1+s*2])
```

```
            padded_img[p].insert(0, padded_img[p][0])
    # adding padding to top and bottom by copying edges
    for z in range(sigma*3):
        padded_img.append(padded_img[img_height-1+z*2])
        padded_img.insert(0, padded_img[0])

    #applying Gaussian kernel to padded image
    blurred_img = []
    matrix_sum = 0
    for x in range(img_height):
        blurred_row = []
        for y in range(img_width):
            for i in range(sigma*6+1):
                for j in range(sigma*6+1):
                    matrix_sum += kernel[i][j] * padded_img[x+i][y+j]
            if matrix_sum > 255:
                matrix_sum = 255
            blurred_row.append(round(matrix_sum))
            matrix_sum = 0
        blurred_img.append(blurred_row)
    return blurred_img


def sobel_edge(blurred_img):

    sobel_x = [[1, 0, -1], [2, 0, -2], [1, 0, -1]]
    sobel_y = [[1, 2, 1], [0, 0, 0], [-1, -2, -1]]

    # padding existing rows by copying edges
    for a in range(img_height):
        for b in range(1):
            blurred_img[a].append(blurred_img[a][img_width-1+b*2])
            blurred_img[a].insert(0, blurred_img[a][0])
    # adding padding to top and bottom by copying edges
    for c in range(1):
        blurred_img.append(blurred_img[img_height-1+c*2])
        blurred_img.insert(0, blurred_img[0])

    #applying sobel filters to smoothed image
    sobel_imgx = []
    sobel_imgy = []
    sobel_dir = []
    sobel_sumx = 0
    sobel_sumy = 0

    for d in range(img_height):
        sobel_rowx = []
        sobel_rowy = []
```

```
        sobel_dir_row = []
        for e in range(img_width):
            for f in range(3):
                for g in range(3):
                    sobel_sumx += sobel_x[f][g] * blurred_img[d+f][e+g]
                    sobel_sumy += sobel_y[f][g] * blurred_img[d+f][e+g]

            #making sure I don't divide by 0 in arctan
            if sobel_sumx == 0:
                direction = math.pi/2
            else:
                direction = math.atan2(sobel_sumy, sobel_sumx)

            sobel_rowx.append(sobel_sumx)
            sobel_rowy.append(sobel_sumy)
            sobel_dir_row.append(direction)

            sobel_sumx = 0
            sobel_sumy = 0

        sobel_imgx.append(sobel_rowx)
        sobel_imgy.append(sobel_rowy)
        sobel_dir.append(sobel_dir_row)

    #keep values separate so sobel_img matrix can still be easily turned into an image
    return sobel_imgx, sobel_imgy, sobel_dir


#non-maximum suppression
def maxsup(sobel_image, sobel_direction):

    maxsup_image = deepcopy(sobel_image)

    for y in range(len(maxsup_image)):
        for x in range(len(maxsup_image[0])):
            #checking horizontal, between 67.5 and 112.5 or -67.5 and -112.5 degrees
            if((sobel_direction[y][x] < 1.9634954 and sobel_direction[y][x] > 1.178097) or
            (sobel_direction[y][x] > -1.9634954 and sobel_direction[y][x] < -1.178097)):
                #right OOB (out of bounds)
                if(x+1 > len(maxsup_image[0])-1):
                    if maxsup_image[y][x] >= maxsup_image[y][x-1]:
                        maxsup_image[y][x-1] = 0
                    else:
                        maxsup_image[y][x] = 0
                #left OOB
                elif(x-1 < 0):
                    if maxsup_image[y][x] >= maxsup_image[y][x+1]:
                        maxsup_image[y][x+1] = 0
```

```
            else:
                maxsup_image[y][x] = 0
        else:
            if(maxsup_image[y][x] >= maxsup_image[y][x-1] and
            maxsup_image[y][x] >= maxsup_image[y][x+1]):
                maxsup_image[y][x-1]= 0
                maxsup_image[y][x+1] = 0
            elif(maxsup_image[y][x+1] >= maxsup_image[y][x] and
            maxsup_image[y][x+1] >= maxsup_image[y][x-1]):
                maxsup_image[y][x-1] = 0
                maxsup_image[y][x] = 0
            else:
                maxsup_image[y][x] = 0
                maxsup_image[y][x+1] = 0
    #checking vertical, between 22.5 and -22.5 or 157.5 and -157.5 degrees
    if((sobel_direction[y][x] < 0.3926991 and sobel_direction[y][x] > -0.3926991) or
    (sobel_direction[y][x] > 2.7488936 or sobel_direction[y][x] < -2.7488936)):
        #bottom OOB
        if(y+1 > len(maxsup_image)-1):
            if maxsup_image[y][x] >= maxsup_image[y-1][x]:
                maxsup_image[y-1][x] = 0
            else:
                maxsup_image[y][x] = 0
        #top OOB
        elif(y-1 < 0):
            if maxsup_image[y][x] >= maxsup_image[y+1][x]:
                maxsup_image[y+1][x] = 0
            else:
                maxsup_image[y][x] = 0
        else:
            if(maxsup_image[y][x] >= maxsup_image[y-1][x] and
            maxsup_image[y][x] >= maxsup_image[y+1][x]):
                maxsup_image[y-1][x] = 0
                maxsup_image[y+1][x] = 0
            elif(maxsup_image[y+1][x] >= maxsup_image[y][x] and
            maxsup_image[y+1][x] >= maxsup_image[y-1][x]):
                maxsup_image[y-1][x] = 0
                maxsup_image[y][x] = 0
            else:
                maxsup_image[y][x] = 0
                maxsup_image[y+1][x] = 0
    # checking diagonal 1 (top left to bottom right and vice versa),
    # between 22.5 and 67.5 or -112.5 and -157.5 degrees
    elif((sobel_direction[y][x] <= 1.178097 and sobel_direction[y][x] >= 0.3926991) or
    (sobel_direction[y][x] >= -2.7488936 and sobel_direction[y][x] <= -1.9634954)):
        #bottom right OOB
        if(y+1 > len(maxsup_image)-1 or x+1 > len(maxsup_image[0])-1):
            if maxsup_image[y][x] >= maxsup_image[y-1][x-1]:
```

```
            maxsup_image[y-1][x-1] = 0
        else:
            maxsup_image[y][x] = 0
    #top left OOB
    elif(y-1 < 0 or x-1 < 0):
        if maxsup_image[y][x] >= maxsup_image[y+1][x+1]:
            maxsup_image[y+1][x+1] = 0
        else:
            maxsup_image[y][x] = 0
    else:
        if(maxsup_image[y][x] >= maxsup_image[y-1][x-1] and
        maxsup_image[y][x] >= maxsup_image[y+1][x+1]):
            maxsup_image[y-1][x-1] = 0
            maxsup_image[y+1][x+1] = 0
        elif(maxsup_image[y+1][x+1] >= maxsup_image[y][x] and
        maxsup_image[y+1][x+1] >= maxsup_image[y-1][x-1]):
            maxsup_image[y-1][x-1] = 0
            maxsup_image[y][x] = 0
        else:
            maxsup_image[y][x] = 0
            maxsup_image[y+1][x+1] = 0
# checking diagonal 2 (top right to bottom left and vice versa),
# between 112.5 and 157.5 or -22.5 and -67.5 degrees
elif((sobel_direction[y][x] <= 2.7488936 and sobel_direction[y][x] >= 1.9634954) or
(sobel_direction[y][x] >= -1.178097 and sobel_direction[y][x] <= -0.3926991)):
    #top right OOB
    if(y+1 > len(maxsup_image)-1 or x-1 < 0):
        if maxsup_image[y][x] >= maxsup_image[y-1][x+1]:
            maxsup_image[y-1][x+1] = 0
        else:
            maxsup_image[y][x] = 0
    #bottom left OOB
    elif(y-1 < 0 or x+1 > len(maxsup_image[0])-1):
        if maxsup_image[y][x] >= maxsup_image[y+1][x-1]:
            maxsup_image[y+1][x-1] = 0
        else:
            maxsup_image[y][x] = 0
    else:
        if(maxsup_image[y][x] >= maxsup_image[y-1][x+1] and
        maxsup_image[y][x] >= maxsup_image[y+1][x-1]):
            maxsup_image[y-1][x+1] = 0
            maxsup_image[y+1][x-1] = 0
        elif(maxsup_image[y+1][x-1] >= maxsup_image[y][x] and
        maxsup_image[y+1][x-1] >= maxsup_image[y-1][x+1]):
            maxsup_image[y-1][x+1] = 0
            maxsup_image[y][x] = 0
        else:
            maxsup_image[y][x] = 0
```

```
                    maxsup_image[y+1][x-1] = 0
    key_locations = []
    #standardizing remaining edges
    for a in range(len(maxsup_image)):
        for b in range(len(maxsup_image[0])):
            if maxsup_image[a][b] > 0:
                maxsup_image[a][b] = 255
                key_locations.append((a, b))

    return maxsup_image, key_locations


def hessian_detector(sobel_imgx, sobel_imgy):
    sobel_x = [[1, 0, -1], [2, 0, -2], [1, 0, -1]]
    sobel_y = [[1, 2, 1], [0, 0, 0], [-1, -2, -1]]

    hessian_img = [[0 for col in range(img_width)] for row in range(img_height)]
    sobel_sumxx = 0
    sobel_sumxy = 0
    sobel_sumyy = 0
    for x in range(1, img_height-1):
        for y in range(1, img_width-1):
            for a in range(-1, 2):
                for b in range(-1, 2):
                    #taking second derivatives
                    sobel_sumxx += sobel_x[a+1][b+1] * sobel_imgx[x+a][y+b]
                    sobel_sumxy += sobel_y[a+1][b+1] * sobel_imgx[x+a][y+b]
                    sobel_sumyy += sobel_y[a+1][b+1] * sobel_imgy[x+a][y+b]

            determinant = sobel_sumxx * sobel_sumyy - sobel_sumxy**2

            #thresholding to find corners
            if (determinant > 100000 or determinant < -80000):
                hessian_img[x][y] = 255

            sobel_sumxx = 0
            sobel_sumxy = 0
            sobel_sumyy = 0

    return hessian_img

def ransac(dist_threshold, required_inliers, key_locations):
    #don't need 500 samples, but I keep it high just to be safe
    num_samples = 700
    #dist_threshold = 1.2

    #creating first 4 lines found that meet inlier threshold
```

```
    line1_points, key_locations = ransac_helper(key_locations, dist_threshold, num_samples,
required_inliers)
    line2_points, key_locations = ransac_helper(key_locations, dist_threshold, num_samples,
required_inliers)
    line3_points, key_locations = ransac_helper(key_locations, dist_threshold, num_samples,
required_inliers)
    line4_points, key_locations = ransac_helper(key_locations, dist_threshold, num_samples,
required_inliers)

    im = Image.open("hessianroad.png")
    d = ImageDraw.Draw(im)

    line1max = (0, 0)
    line1min = (img_width, img_height)
    line2max = (0, 0)
    line2min = (img_width, img_height)
    line3max = (0, 0)
    line3min = (img_width, img_height)
    line4max = (0, 0)
    line4min = (img_width, img_height)


    #clamping all points in lines to max 8 bit value
    lines_image = [[0 for col in range(img_width)] for row in range(img_height)]
    for i in line1_points:
        if(i[1] > line1max[0]):
            line1max = i[::-1]
        if(i[1] < line1min[0]):
            line1min = i[::-1]
        lines_image[i[0]][i[1]] = 255

    for i in line2_points:
        if(i[1] > line2max[0]):
            line2max = i[::-1]
        if(i[1] < line2min[0]):
            line2min = i[::-1]
        lines_image[i[0]][i[1]] = 255

    for i in line3_points:
        if(i[1] > line3max[0]):
            line3max = i[::-1]
        if(i[1] < line3min[0]):
            line3min = i[::-1]
        lines_image[i[0]][i[1]] = 255

    for i in line4_points:
        if(i[1] > line4max[0]):
            line4max = i[::-1]
```

```
        if(i[1] < line4min[0]):
            line4min = i[::-1]
        lines_image[i[0]][i[1]] = 255

    line_color = "red"
    d.line((line1min, line1max), fill=line_color, width=3)
    d.line((line2min, line2max), fill=line_color, width=3)
    d.line((line3min, line3max), fill=line_color, width=3)
    d.line((line4min, line4max), fill=line_color, width=3)
    del d
    im.save("ransaclines.png")

    return lines_image


#helper function runs once for each line found
def ransac_helper(key_locations, dist_t, num_samples, req_inliers):
    while(num_samples > 0):
        #copying key_locations so I don't screw up original
        key_locs = deepcopy(key_locations)
        point1 = random.choice(key_locations)
        point2 = random.choice(key_locations)
        while (point1 == point2):
            point2 = random.choice(key_locations)

        line_is_vert = False
        #creating standard form of line for perpendicular distance formula
        if(point2[1]-point1[1] != 0):
            slope = ((-1*point2[0])-(-1*point1[0])) / (point2[1]-point1[1])
            line_A = slope * -1
            #not used, just left for clarity
            line_B = 1
            line_C = slope * point1[1] + point1[0]
            denom = ((line_A)**2 + 1)**(1/2)
        else:
            #making sure I'm not dividing by 0 trying to find slope of vertical lines
            line_is_vert = True

        inliers = []

        x = 0
        while x < len(key_locs):
            if(not line_is_vert):
                dist = abs(line_A * key_locs[x][1] + (-1*key_locs[x][0]) + line_C) / denom
            #only for vertical lines
            else:
                dist = abs(key_locs[x][1] - point1[1])
```

```
            if dist <= dist_t:
                inliers.append(key_locs[x])
                del key_locs[x]
                x -= 1
            x += 1
        #65 is a good number to use
        if len(inliers) >= req_inliers:
            return inliers, key_locs

        num_samples -= 1
    print("failed")
    return inliers, key_locs

def hough_transform(theta_s, rho_s, key_locations):
    #theta from 0 to -180 degrees, rho from 0 to length of diagonal
    diagonal = round((img_height**2 + img_width**2)**(1/2))

    extra_theta = 0
    if(180 % theta_s != 0):
        extra_theta = 180%theta_s / (180//theta_s)
    extra_rho = 0
    if(diagonal % rho_s != 0):
        extra_rho = diagonal%rho_s / (diagonal//rho_s)
    hough_matrix = [[[] for col in range(round(180//theta_s))] for row in
range(round(diagonal//rho_s))]
    for i in range(len(key_locations)):
        point1 = key_locations[i]

        for j in range(i+1, len(key_locations)):
            point2 = key_locations[j]
            if((point2[1]-point1[1]) != 0):
                slope = ((-1*point2[0])-(-1*point1[0])) / (point2[1]-point1[1])
                line_A = slope * -1
                line_C = slope * point1[1] + point1[0]
                theta = (-180/math.pi) * (atan(slope) - (math.pi / 2))
                rho = abs(line_C) / (line_A**2 + 1)**(1/2)
            else:
                theta = 0
                rho = abs(point2[0]-point1[0])

            #need to remember line points so I can plot them
            if(j == i+1):
                hough_matrix[round(rho // (rho_s+extra_rho))][round(theta //
(theta_s+extra_theta))].append(key_locations[i])
            hough_matrix[round(rho // (rho_s+extra_rho))][round(theta //
(theta_s+extra_theta))].append(key_locations[j])

    #format: a index, b index, # of points in line
```

```
line1 = (0, 0, 0)
line2 = (0, 0, 0)
line3 = (0, 0, 0)
line4 = (0, 0, 0)
#finding four lines with most support
for a in range(len(hough_matrix)):
    for b in range(len(hough_matrix[0])):
        if(len(hough_matrix[a][b]) > line4[2]):
            if(len(hough_matrix[a][b]) < line3[2]):
                line4 = (a, b, len(hough_matrix[a][b]))
            elif(len(hough_matrix[a][b]) < line2[2]):
                line3 = (a, b, len(hough_matrix[a][b]))
            elif(len(hough_matrix[a][b]) < line1[2]):
                line2 = (a, b, len(hough_matrix[a][b]))
            else:
                line1 = (a, b, len(hough_matrix[a][b]))

im = Image.open("hessianroad.png")
d = ImageDraw.Draw(im)

line1max = (0, 0)
line1min = (img_width, img_height)
line2max = (0, 0)
line2min = (img_width, img_height)
line3max = (0, 0)
line3min = (img_width, img_height)
line4max = (0, 0)
line4min = (img_width, img_height)


#clamping all points in lines to max 8 bit value
hough_image = [[0 for col in range(img_width)] for row in range(img_height)]
for i in hough_matrix[line1[0]][line1[1]]:
    if(i[1] > line1max[0]):
        line1max = i[::-1]
    if(i[1] < line1min[0]):
        line1min = i[::-1]
    hough_image[i[0]][i[1]] = 255

for i in hough_matrix[line2[0]][line2[1]]:
    if(i[1] > line2max[0]):
        line2max = i[::-1]
    if(i[1] < line2min[0]):
        line2min = i[::-1]
    hough_image[i[0]][i[1]] = 255

for i in hough_matrix[line3[0]][line3[1]]:
    if(i[1] > line3max[0]):
```

```
            line3max = i[::-1]
        if(i[1] < line3min[0]):
            line3min = i[::-1]
        hough_image[i[0]][i[1]] = 255

    for i in hough_matrix[line4[0]][line4[1]]:
        if(i[1] > line4max[0]):
            line4max = i[::-1]
        if(i[1] < line4min[0]):
            line4min = i[::-1]
        hough_image[i[0]][i[1]] = 255

    line_color = "red"
    d.line((line1min, line1max), fill=line_color, width=3)
    d.line((line2min, line2max), fill=line_color, width=3)
    d.line((line3min, line3max), fill=line_color, width=3)
    d.line((line4min, line4max), fill=line_color, width=3)
    del d
    im.save("houghlines.png")

    return hough_image

def main():

    img = open('road.pgm', 'r+b')
    image_matrix = read_pgm(img)

    #1.2
    ransac_dist = float(input("Enter RANSAC distance threshold: "))
    #55
    ransac_inliers = int(input("Enter RANSAC inlier threshold: "))
    #1.5
    hough_theta_acc = float(input("Enter Hough Transform theta accuracy: "))
    #1
    hough_rho_acc = float(input("Enter Hough Transform rho accuracy: "))

    blur_img = gaussian_blur(img_width, img_height, 1, image_matrix)
    sobelx, sobely, sobel_direction = sobel_edge(blur_img)
    hessian_img = hessian_detector(sobelx, sobely)
    final_hessian, key_locations = maxsup(hessian_img, sobel_direction)

    fout = open("hessianroad.png", 'wb')

    w = png.Writer(img_width, img_height, greyscale=True)
    w.write(fout, final_hessian)

    fout.close()
```

```
    lines = ransac(ransac_dist, ransac_inliers, key_locations)
    hough = hough_transform(hough_theta_acc, hough_rho_acc, key_locations)


    fout = open("ransacpoints.png", 'wb')

    w = png.Writer(img_width, img_height, greyscale=True)
    w.write(fout, lines)

    fout.close()

    fout = open("houghpoints.png", 'wb')

    w = png.Writer(img_width, img_height, greyscale=True)
    w.write(fout, hough)

    fout.close()

    print("Done.")


if __name__ == "__main__":
    main()
```