```python
'''
HW1
Noah Suttora
I pledge my honor that I have abided by the Stevens Honor System.
'''
import cv2
import numpy as np
import math
import random

# spatially-weighted average formula
def gaussianFormula(x, y, stdev):
    pi = math.pi
    e = math.e
    return 1/(2*pi*stdev**2) * e**(-((x**2+y**2)/(2*stdev**2)))

# calculate gaussian filter based on window size and standard deviation
def gaussianFilter(image, size=5, stdev=1):
    # initialize 5x5 window and its center
    gFilter = np.zeros((size, size))
    gCenter = [math.floor(size/2), math.floor(size/2)]

    # calc filter at cell based on distance from center
    for i in range(size):
        for j in range(size):
            gFilter[i][j] = gaussianFormula(abs(gCenter[0]-i), abs(gCenter[1]-j), stdev)

    return gFilter

# derivative operator for vertical edges
def sobelX():
    return np.array([   [-1, 0, 1],
                        [-2, 0, 2],
                        [-1, 0, 1]])

# derivative operator for horizontal edges
def sobelY():
    return np.array([   [1, 2, 1],
                        [0, 0, 0],
                        [-1, -2, -1]])

# apply kernel filter to image with new output
# help in understanding and implementing function from:
#   https://medium.com/analytics-vidhya/2d-convolution-using-python-numpy-43442ff5f381
def convolute(image, kernel):
    # kernel and image shapes
    xKernShape = kernel.shape[0]
    yKernShape = kernel.shape[1]
    xImgShape = image.shape[0]
    yImgShape = image.shape[1]

    # n_out = n_in - k_size + 1 (n_out: num output features, n_in: num input features)
    xOutput = xImgShape - xKernShape + 1
    yOutput = yImgShape - yKernShape + 1
    output = np.zeros((xOutput, yOutput))

    for y in range(yImgShape):
        # kernel out of bounds column-wise so exit convolution algorithm
        if y > yImgShape - yKernShape:
            break
        else:
            for x in range(xImgShape):
                # kernel out of bounds row-wise so go to next column
                if x > xImgShape - xKernShape:
                    break
                else:
                    # h[m,n] = Σ g[k,l] * f[m+k,n+l]
                    output[x, y] = (kernel * image[x: x + xKernShape, y: y + yKernShape]).sum()

    return output
```

```python
# matrix of all second partial derivatives of Sobel filters
def hessian(image, threshold=130):
    # first and second partial derivates
    xImage = convolute(image, sobelX())
    xxImage = convolute(xImage, sobelX())
    yImage = convolute(image, sobelY())
    yyImage = convolute(yImage, sobelY())
    xyImage = convolute(yImage, sobelX())

    # second partial derivative shapes
    xShape = xxImage.shape[0]
    yShape = xxImage.shape[1]

    # calculate determinant
    det = np.zeros((xShape, yShape))
    for i in range(xShape):
        for j in range(yShape):
            det[i][j] = xxImage[i][j] * yyImage[i][j] - xyImage[i][j] * xyImage[i][j]

    # min-max normalization: (X-min)/(max-min)
    for i in range(xShape):
        for j in range(yShape):
            det[i][j] = (det[i][j] - np.amin(det)) / ((np.amax(det) - np.amin(det))/255)

    # threshold the determinant
    for i in range(int(0.7*xShape)):
        for j in range(yShape):
            # below threshold so don't count
            if det[i][j] < threshold:
                det[i][j] = 0
            else:
                # above threshold so count
                det[i][j] = 255

    # fix miscalculating points
    for i in range(int(0.7*xShape), xShape):
        for j in range(yShape):
            det[i][j] = 0

    return det

# thin clusters to single pixel
def nonMaximumSuppression(image):
    # iterate image except edges because of window size
    for i in range(1, image.shape[0]-1):
        for j in range(1, image.shape[1]-1):
            # 3x3 neighborhood window
            window = [  image[i-1][j-1], image[i-1][j], image[i-1][j+1],    # 1st row
                        image[i][j-1], image[i][j], image[i][j+1],          # 2nd row
                        image[i+1][j-1], image[i+1][j], image[i+1][j+1]]    # 3rd row
            # suppress non max
            if image[i][j] != max(window):
                image[i][j] = 0
            else:
                for k in range(0, 3):
                    for l in range(0, 3):
                        # suppress everything but max
                        if not (k==1 and l==1):
                            image[i+k-1][j+l-1] = 0

    return image
```

```python
# random sample consensus to determine 4 best lines
def ransac(imagePoints, roadImage, maxLines=4, inlierThresh=28, distThresh=2):
    points = []
    for row in range(imagePoints.shape[0]):
        for col in range(imagePoints.shape[1]):
            # pixel is white point
            if imagePoints[row][col] > 0:
                # append point (col - x, row - y)
                points.append((col, row))

    lines = 0
    # keep running until 4 good lines found
    while lines < maxLines:
        # get 2 unique random points
        randPoints = random.sample(points, 2)

        x = [i[0] for i in randPoints]
        y = [j[1] for j in randPoints]
        if x[0] == x[1]:
            # skip vertical lines since infinity slope
            continue
        else:
            slope = (y[1]-y[0]) / (x[1]-x[0])
            intercept = y[0] - slope * x[0]

        inliers = []
        for point in points:
            # distance from point to line = |ax+by+c|/(a^2+b^2)^0.5 = |mx-1y+c|/(m^2+(-1)^2)^0.5
            distance = abs(slope*point[0]-1*point[1]+intercept) / math.sqrt((slope)**2+(-1)**2)
            if distance < distThresh:
                # append close point
                inliers.append(point)

        if len(inliers) >= inlierThresh:
            lines += 1
            # plot longest (min to max) white line on image with points
            cv2.line(imagePoints, min(inliers), max(inliers), (255, 255, 255), thickness=1)
            # plot longest (min to max) black line on normal image
            cv2.line(roadImage, min(inliers), max(inliers), (0, 0, 0), thickness=3)
            # remove used inliers
            for point in inliers:
                points.remove((point[0], point[1]))

    return imagePoints, roadImage
```

```python
# accumulator voting scheme
# help in understanding and implementing function from:
#   https://alyssaq.github.io/2014/understanding-hough-transform/
#   https://towardsdatascience.com/lines-detection-with-hough-transform-84020b3b1549
def hough(imagePoints, roadImage, maxLines=4):
    points = []
    for row in range(imagePoints.shape[0]):
        for col in range(imagePoints.shape[1]):
            # pixel is white point
            if imagePoints[row][col] > 0:
                # append point (col - x, row - y)
                points.append((col, row))

    # height, width, and diagonal calculations
    height = imagePoints.shape[0]
    width = imagePoints.shape[1]
    diagonal = int(np.ceil(np.sqrt(height**2 + width**2)))

    # thetas, cosines, and sines calculations
    thetas = np.deg2rad(np.arange(0, 180))
    cosines = np.cos(thetas)
    sines = np.sin(thetas)

    # accumulator H(θ,p) where p is [-diag, diag] and θ is [0, 180]
    accumulator = np.zeros((2*diagonal, len(thetas)))

    for point in points:
        x = point[0]
        y = point[1]

        for angleIdx in range(len(thetas)):
            # p = xcosθ + ysinθ (+diagonal for positive index offset)
            rho = int(x*cosines[angleIdx] + y*sines[angleIdx] + diagonal)
            # H(θ,p) = H(θ,p)+1 (+20 so points show better)
            accumulator[rho][angleIdx] += 20

    # save original accumulator since we will be removing points at local max
    accumulatorOriginal = np.copy(accumulator)

    lines = 0
    # keep running until 4 good lines found
    while lines < maxLines:
        localMax = 0

        # find value of (θ,p) where H(θ,p) is local max
        for i in range(accumulator.shape[0]):
            for j in range(accumulator.shape[1]):
                if accumulator[i][j] > localMax:
                    paramRho = i
                    paramTheta = j
                    localMax = accumulator[i][j]

        # remove points around localMax for distinct new line next iteration
        for i in range(-10, 10):
            for j in range(-10, 10):
                accumulator[paramRho + i][paramTheta + j] = 0

        # remove diagonal offset for actual value of rho
        paramRho = paramRho - diagonal
        # convert to radians for actual value of theta
        paramTheta = np.deg2rad(paramTheta)

        # line parameters to convert to y=ax+b
        a = np.cos(paramTheta)
        b = np.sin(paramTheta)
        x0 = a * paramRho
        y0 = b * paramRho

        # two points to span entire image
        pt1 = (int(x0 + 1000*(-b)), int(y0 + 1000*(a)))
        pt2 = (int(x0 - 1000*(-b)), int(y0 - 1000*(a)))

        # plot white line on image with points
        cv2.line(imagePoints, pt1, pt2, (255, 255, 255), thickness=1)
        # plot black line on normal image
        cv2.line(roadImage, pt1, pt2, (0, 0, 0), thickness=3)

        lines += 1

    return accumulatorOriginal, imagePoints, roadImage
```

```python
def main():
    # load image and convert to grayscale for safety
    image = cv2.imread("road.png")
    image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    print("Image Loaded")

    # calculate and apply gaussian filter to road and save it
    gausFilt = gaussianFilter(image)
    gausImage = convolute(image, gausFilt)
    cv2.imwrite("gaussFilter.png", gausImage)
    print("Gaussian Filter Applied")

    # apply sobel x to road and save it
    xSobelImage = convolute(gausImage, sobelX())
    cv2.imwrite("xSobelFilter.png", xSobelImage)
    print("Vertical Sobel Filter Applied")

    # apply sobel y to road and save it
    ySobelImage = convolute(gausImage, sobelY())
    cv2.imwrite("ySobelFilter.png", ySobelImage)
    print("Horizontal Sobel Filter Applied")

    # calculate hessian determinant and save it
    hessDet = hessian(gausImage)
    cv2.imwrite("hessianDet.png", hessDet)
    print("Hessian Detector Applied")

    # apply NMS to hessian determinant and save it
    hessSuppressed = nonMaximumSuppression(hessDet)
    cv2.imwrite("hessianSuppressed.png", hessSuppressed)
    print("Non-Maximum Suppression on Hessian Applied")

    # use ransac to find 4 best lines and save it
    ransacPoints, ransacImage = ransac(hessSuppressed, image)
    cv2.imwrite("ransacPoints.png", ransacPoints)
    cv2.imwrite("ransacRoad.png", ransacImage)
    print("RANSAC Applied")

    # use hough transform to find 4 best lines and save it
    accum, houghPoints, houghImage = hough(hessSuppressed, image)
    cv2.imwrite("accumulator.png", accum)
    cv2.imwrite("houghPoints.png", houghPoints)
    cv2.imwrite("houghRoad.png", houghImage)
    print("Hough Transformation Applied")

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

Pre-processing

A 5x5 Gaussian filter is calculated and applied to the image using convolution (sliding window technique): $h[m,n] = \Sigma\, g[k,l] * f[m+k,n+l]$

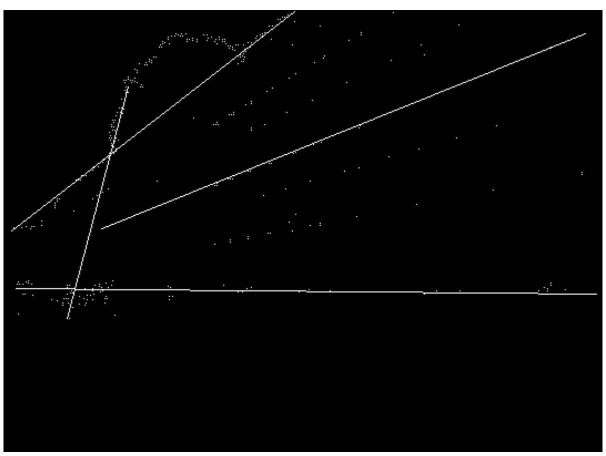
Sobel filters are applied to the image using convolution (left is x Sobel, right is y Sobel).

Hessian matrix is composed by taking the second partial derivatives using Sobel as a derivative operator. The determinant is then calculated, normalized, and thresholded to give the key points in the image. Non-maximum suppression is applied on this using 3x3 neighborhoods (left is before NMS, right is after NMS).

RANSAC

The NMS is traversed to store the pixel values of the key points. RANSAC proceeds as follows: two distinct random points are chosen, the slope and intercept of these two points are calculated, the distance of each point to the line is calculated and if it is below the distance threshold (mine is 2) it is counted as an inlier, if there is a significant number of inliers (above my inlier threshold of 28) the line is considered good and is plotted (from smallest inlier to largest inlier for longest line) and those inliers are removed so the next iteration finds a new distinct line. The algorithm runs in a while loop until 4 good lines are found.

Hough Transform

The NMS is traversed to store the pixel values of the key points. Hough transform starts with initializing and calculating the accumulator. The accumulator is initialized to a (2*diagonal, len(thetas)=181) image array and the algorithm for calculating the Hough space proceeds as follows: for each point, from theta of [0, 180] calculate the rho value and add that point to the accumulator.

With this Hough space image, the local max is found by iterating through the image and finding the pixel that holds the local max (the bright intersection points in the Hough space). Once the local max is found, the rho and theta at that local max is stored, the points around the local max are removed so the next iteration finds a new distinct local max, and then the rho and theta are converted back to point coordinates so they can be plotted on the image. The algorithm runs in a while loop until 4 good lines are found.