CS 558-A Computer Vision Assignment 1

Source Code

```
ogImage = cv.imread('road.png')
ogImage = cv.cvtColor(ogImage, cv.COLOR_BGR2GRAY)

############################################################################
################################################
                                                   # GAUSSIAN FILTER #
############################################################################
################################################
def createGaussian(kernalSize, sigma):
    temp = np.linspace(-(kernalSize // 2), kernalSize // 2, kernalSize) # If
size is 3, the array will be [-1, 0, 1]
    gaussian1dKernal = list()

    for i in temp:
        gaussian1dKernal.append((1/(np.sqrt(2*np.pi) * sigma))* (np.e ** ((-
np.square(i))/(2*np.square(sigma))))) # Creates the 1D Gaussian Kernal(a.k.a.
filter)

    gaussian2dKernal = np.outer(gaussian1dKernal, gaussian1dKernal) # Creates
the Gaussian 2D Kernal that we need
    kernal = gaussian2dKernal/(np.sum(gaussian2dKernal))

    return kernal

def applyFilter(image, kernal):
    padSize = len(kernal) // 2
    # cv.imwrite("OG_Image.png", image)
    paddedImage = np.pad(image, padSize, 'edge')
    # cv.imwrite("Padded_image.png", paddedImage)
    output = convolution(paddedImage, kernal, ogImage)
    return output

def convolution(paddedImage, kernal, ogImage):
    opRow, opCol = ogImage.shape
    outputImage = np.zeros((opRow, opCol))
    kernalSize = len(kernal)

    for i in range(opRow):
        for j in range(opCol):
            outputImage[i, j] = np.sum(kernal * paddedImage[i:i+kernalSize,
j:j+kernalSize])

    return outputImage
```

```
##############################################################################
#################################################
                                                    # SOBEL FILTER #
##############################################################################
#################################################

xAxisKernal = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]])
yAxisKernal = np.array([[1, 0, -1], [2, 0, -2], [1, 0, -1]])
padSize = len(xAxisKernal) // 2

def applyXAxisFilter(image):
    paddedImage = np.pad(image, padSize, 'edge')
    edgesXAxis = convolution(paddedImage, xAxisKernal, ogImage)

    return edgesXAxis

def applyYAxisFilter(image):
    paddedImage = np.pad(image, padSize, 'edge')
    edgesYAxis = convolution(paddedImage, yAxisKernal, ogImage)

    return edgesYAxis


##############################################################################
#################################################
                                                # NON-MAXIMUM SUPPRESSION #
##############################################################################
#################################################
def nonMaxSuppression (detHessian):
    paddedImage = np.pad(detHessian, 1, 'edge')
    angleRows, angleCols = paddedImage.shape
    angleMatrix = np.zeros_like(detHessian)

    for i in range(angleRows - 2):
        for j in range(angleCols - 2):
            window = paddedImage[i:i+3, j:j+3]
            horizontalCheck = np.abs(window[1, 0] - window[1, 2])
            verticalCheck = np.abs(window[0, 1] - window[2, 1])
            diagonalLTopToRBtmCheck = np.abs(window[0, 0] - window[2, 2]) #
↘↖ Direction
            diagonalLBtmToRTopCheck = np.abs(window[0, 2] - window[2, 0]) #
↗↙ Direction

            if(verticalCheck > max(horizontalCheck, diagonalLBtmToRTopCheck,
diagonalLTopToRBtmCheck)):
                angleMatrix[i, j] = 270 # ↑↓ Directon
            elif (horizontalCheck > max(verticalCheck,
diagonalLBtmToRTopCheck, diagonalLTopToRBtmCheck)):
                angleMatrix[i, j] = 180 # →← Direction
```

```
            elif (diagonalLTopToRBtmCheck > max(horizontalCheck,
verticalCheck, diagonalLBtmToRTopCheck)):
                angleMatrix[i, j] = 135 # ↘↖ Direction
            elif (diagonalLBtmToRTopCheck > max(horizontalCheck,
verticalCheck, diagonalLTopToRBtmCheck)):
                angleMatrix[i, j] = 225 # ↗↙ Direction

    edgeMatrix = np.zeros_like(detHessian)
    for i in range(angleRows - 2):
        for j in range(angleCols - 2):
            window = paddedImage[i:i+3, j:j+3]

            if(angleMatrix[i, j] == 270):
                if(window[1, 1] > max(window[0, 1], window[2, 1])):
                    edgeMatrix[i, j] = detHessian[i, j]
            elif(angleMatrix[i, j] == 180):
                if(window[1, 1] > max(window[1, 0], window[1, 2])):
                    edgeMatrix[i, j] = detHessian[i, j]
            elif(angleMatrix[i, j] == 135):
                if(window[1, 1] > max(window[0, 0], window[2, 2])):
                    edgeMatrix[i, j] = detHessian[i ,j]
            elif(angleMatrix[i, j] == 225):
                if(window[1, 1] > max(window[0, 2], window[2, 0])):
                    edgeMatrix[i, j] = detHessian[i, j]

    edgeMatrix[edgeMatrix>0] = 255

    # plt.imshow(edgeMatrix)
    # plt.title("edge_detection")
    # plt.show()

    return edgeMatrix

###############################################################################
#################################################
                                                    # HESSIAN DETECTOR #
###############################################################################
#################################################
gaussianKernal = createGaussian(3, 1)
gaussianImage = applyFilter(ogImage, gaussianKernal)

sobelHorizontal1 = applyXAxisFilter(gaussianImage)
sobelHorizontal2 = applyXAxisFilter(sobelHorizontal1)
sobelVertical1 = applyYAxisFilter(gaussianImage)
sobelVertical2 = applyYAxisFilter(sobelVertical1)
sobelXYAxis1 = applyYAxisFilter(gaussianImage)
sobelXYAxis2 = applyYAxisFilter(sobelXYAxis1)
```

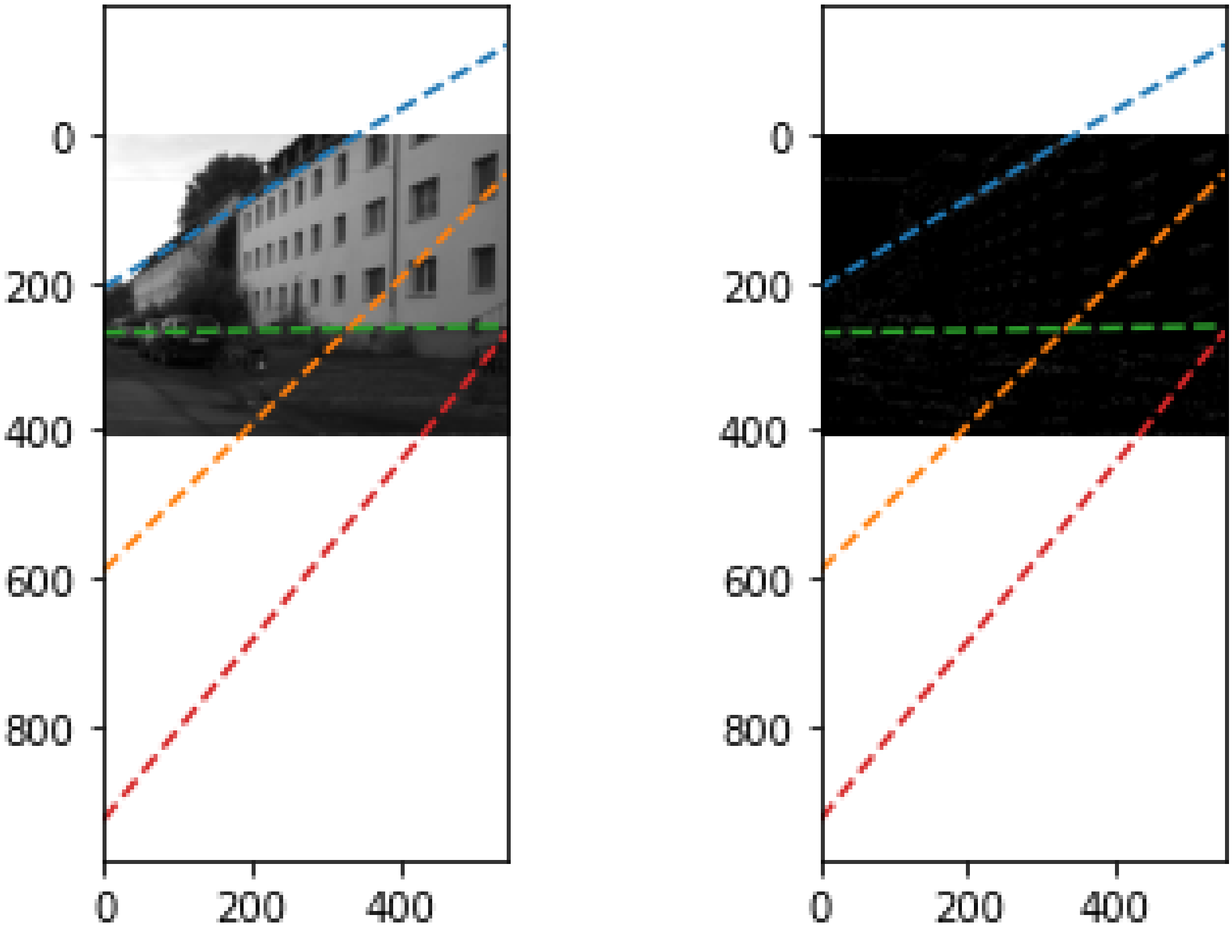
```
detHessian = np.multiply(sobelHorizontal1, sobelVertical2) -
np.power(sobelXYAxis2, 2)

# Threshold
threshold_low = 100
threshold_high = 0
detHessian[detHessian < threshold_low] = 0
if (threshold_high != 0 and threshold_high>threshold_low):
    detHessian[detHessian <= threshold_high] = 255
    detHessian[detHessian > threshold_high] = 0

fig, ax = plt.subplots(1, 2)
ax[0].imshow(ogImage, cmap = 'gray')
ax[1].imshow(detHessian, cmap = 'gray')
plt.show()

nmsImg = nonMaxSuppression(detHessian)


##############################################################################
#################################################
                                                        # RANSAC #
##############################################################################
#################################################
def applyRansac(image, iterations, threshold, ratio):

    pointCoordinates = np.argwhere(image == 255)
    optimalLines = []
    for i in range(0, 4):
        inliersUsed = np.array([])
        bestRatioOfInliers = 0
        bestLineSlope = 0
        bestLineIntercept = 0
        optimalInliers = np.asarray([])
        for i in range(iterations):

            pointsArray = np.random.randint(0, len(pointCoordinates), size =
2)
            points = np.asarray([pointCoordinates[pointsArray[0]],
pointCoordinates[pointsArray[1]]])

            slopeOfLine = (points[1, 1] - points[0, 1])/(points[1, 0] -
points[0, 0])
            lineIntercept = points[1, 1] - slopeOfLine * points[1, 0]
            count = 0

            yCoordinates, xCoordinates = pointCoordinates[:, 0],
pointCoordinates[:, 1]
```

```
            for i, (y, x) in enumerate(zip(yCoordinates,
xCoordinates)):
                point_x = (x + slopeOfLine*y - slopeOfLine*lineIntercept)/(1 +
slopeOfLine**2)
                point_y = (slopeOfLine*x + (slopeOfLine ** 2)*y - (slopeOfLine
** 2)*lineIntercept)/(1 + slopeOfLine**2) + lineIntercept
                dist = np.sqrt((point_x - x)**2 + (point_y - y)**2)
                if (dist <= threshold):
                    inliersUsed = np.append(inliersUsed, i)
                    count = count + 1

            inlierRatio =
count/(len(pointCoordinates)*len(pointCoordinates[0]))

            if (inlierRatio >= ratio):
                if (inlierRatio>bestRatioOfInliers):
                    bestRatioOfInliers = inlierRatio
                    bestLineSlope = slopeOfLine
                    bestLineIntercept = lineIntercept
                    optimalInliers = inliersUsed
                    bestRatio = inlierRatio

        #Adding the best lines to a list.
        optimalLines.append([bestLineSlope, bestLineIntercept, bestRatio])

        #Deleting all the used inliers.
        pointCoordinates = np.delete(pointCoordinates,
optimalInliers.astype(np.uint), axis = 0)

    return np.asarray(optimalLines)


ransacOutput = applyRansac(nmsImg, 1000, math.sqrt(3.84), 0.000)
# print(ransacOutput)
fig, ax = plt.subplots(1, 2)
ax[0].imshow(ogImage, cmap = 'gray')
ax[1].imshow(nmsImg, cmap = 'gray')

axes = plt.gca()
xValues = np.array(axes.get_xlim())
yValues = ransacOutput[0, 1] + ransacOutput[0, 0] * xValues
ax[0].plot(xValues, yValues, '--')
yValues = ransacOutput[1, 1] + ransacOutput[1, 0] * xValues
ax[0].plot(xValues, yValues, '--')
yValues = ransacOutput[2, 1] + ransacOutput[2, 0] * xValues
ax[0].plot(xValues, yValues, '--')
yValues = ransacOutput[3, 1] + ransacOutput[3, 0] * xValues
ax[0].plot(xValues, yValues, '--')
```

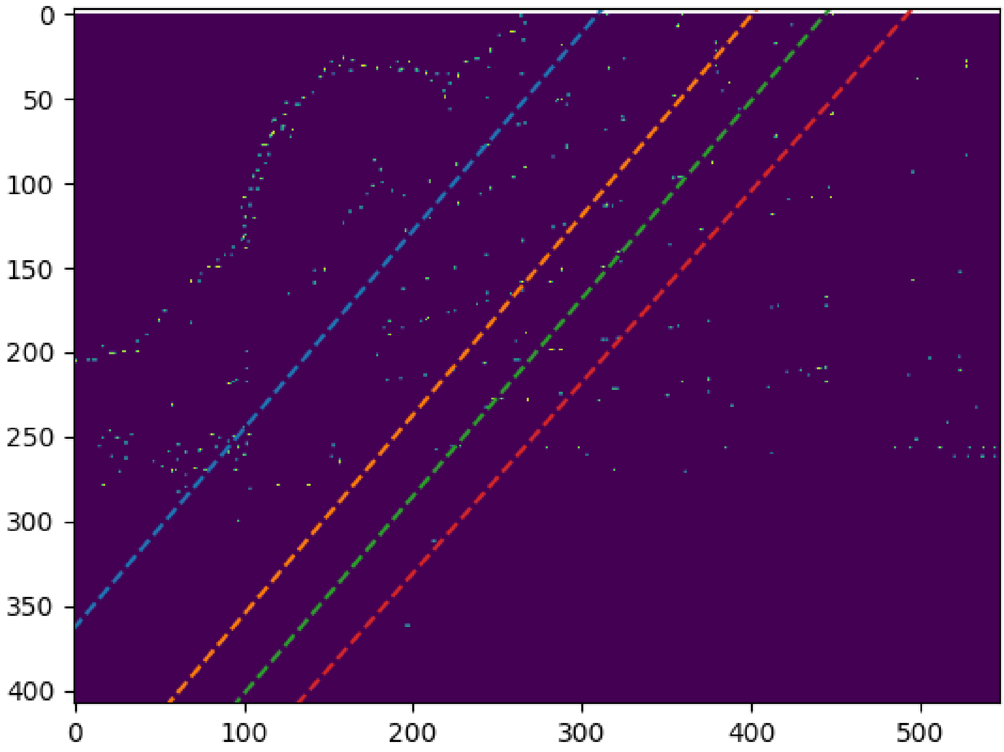
```
axes = plt.gca()
xValues = np.array(axes.get_xlim())
yValues = ransacOutput[0, 1] + ransacOutput[0, 0] * xValues
ax[1].plot(xValues, yValues, '--')
yValues = ransacOutput[1, 1] + ransacOutput[1, 0] * xValues
ax[1].plot(xValues, yValues, '--')
yValues = ransacOutput[2, 1] + ransacOutput[2, 0] * xValues
ax[1].plot(xValues, yValues, '--')
yValues = ransacOutput[3, 1] + ransacOutput[3, 0] * xValues
ax[1].plot(xValues, yValues, '--')

plt.show()


##############################################################################
################################################
                                                   # HOUGH TRANSFORM #
##############################################################################
################################################
def applyHough(nmsImg, threshold):

    #Making a matrix with values of all polar coordinate values.
    diagonalLen = np.sqrt(np.square(len(nmsImg)) + np.square(len(nmsImg[0])))
    angles = np.arange(0, 180)
    normalsArr = np.arange(-diagonalLen, diagonalLen)
    cosVal = np.cos(np.deg2rad(angles))
    sinVal = np.sin(np.deg2rad(angles))
    bins = np.zeros((len(normalsArr), len(angles)))
    coordinates = np.argwhere(nmsImg == 255)

    # Add polar coordinates to bins matrix.
    for i in range(len(coordinates)):
        polarCoordinates = coordinates[i]

        polarCoordinates[0] = polarCoordinates[0] - (len(nmsImg)/2)
        polarCoordinates[1] = polarCoordinates[1] - (len(nmsImg[0])/2)

        for j in range(len(angles)):
            normalValue = (polarCoordinates[1] * cosVal[j] +
polarCoordinates[0]*sinVal[j])
            angle = angles[j]

            minimumNormalAbs = np.abs(normalsArr - normalValue)
            minimumNormal = np.argmin(minimumNormalAbs)

            bins[minimumNormal, angle] += 1

    # Find points with most votes and make lines
    pointsWithMostVotes = []
```

```
    outerLoop = bins.shape[0]
    innerLoop = bins.shape[1]
    for j in range(outerLoop):
        for i in range(innerLoop):
            if (bins[j, i] >= threshold):
                normalValue = normalsArr[j]
                angle = angles[i]
                sinVal = np.sin(np.deg2rad(angle))
                cosVal = np.cos(np.deg2rad(angle))
                imageSize = len(nmsImg)/2
                imageSize0 = len(nmsImg[0])/2

                x1 = int((cosVal * normalValue) + imageSize0 + 1000 * (-
cosVal))
                y1 = int((sinVal * normalValue) + imageSize + 1000 * (sinVal))
                x2 = int((cosVal * normalValue) + imageSize0 - 1000 * (-
cosVal))
                y2 = int((sinVal * normalValue) + imageSize - 1000 * (sinVal))

                pointWithMostVotes = np.array([[x1, y1], [x2, y2], bins[j,
i]])
                pointsWithMostVotes.append(pointWithMostVotes)

    pointsWithMostVotes = np.array(pointsWithMostVotes)
    points = pointsWithMostVotes[np.argsort(pointsWithMostVotes[:, 2])]
    return points[-4:]


houghHighestVotes = applyHough(nmsImg, 15)

plt.imshow(ogImage)
plt.plot(houghHighestVotes[0, 0], houghHighestVotes[0, 1], '--')
plt.plot(houghHighestVotes[1, 0], houghHighestVotes[1, 1], '--')
plt.plot(houghHighestVotes[2, 0], houghHighestVotes[2, 1], '--')
plt.plot(houghHighestVotes[3, 0], houghHighestVotes[3, 1], '--')
plt.show()
```

RANSAC Output



HOUGH Transform



Explanation

First of all, for the pre-processing part, I made use of the Gaussian filter and Sobel filter from the previous assignment. Then I applied the determinant to get the Hessian result and thresholded the result. After that applied non-maximum suppression.

RANSAC: I took 4 values into the RANSAC function; the output from pre-processing, number of iterations, thresholding value, and ratio. Then I added all the pixels with intensity 255, to an array. In order to find the 4 optimal lines, I ran a for loop that had another for loop that ran for the number of iterations specified and got the best line needed and after saving that line in an array, I deleted the outliers for that best line found so that those points aren't considered again. Finally, I printed those lines.

HOUGH Transform: I took 2 values into the HOUGH transformation function; the output from pre-processing and thresholding value. I first made a Matrix of all polar coordinates. Then I put those coordinates into the bin matrix. Then I ran a for loop in order to find the points with the most votes and get lines based on the dimensions of the bins matrix. Then I returned only the best 4 lines that we need. Finally I plotted those lines.