Computer Vision Homework #3 Report

1. Read RGB image and convert to grayscale image:

```
if not(os.path.isdir(os.getcwd() + '/result_img')):
  os.makedirs(os.getcwd() + '/result_img')
img1 = cv2.imread('test_img/img1.png')
img2 = cv2.imread('test_img/img2.png')
img3 = cv2.imread('test_img/img3.png')

img1_gray = getGrayImg(img1)
img2_gray = getGrayImg(img2)
img3_gray = getGrayImg(img3)
```

```
def getGrayImg(img):
  img_gray = np.zeros((img.shape[0], img.shape[1]), dtype=np.uint8)
  for i in range(0, img.shape[0]):
    for j in range(0, img.shape[1]):
      img_gray[i, j] = 0.299 * img[i, j, 2] + 0.587 * img[i, j, 1] + 0.114 * img[i, j, 0]
  return img_gray
```

Read RGB image with cv2.imread() function and convert these image to grayscale within the getGrayImg() function.

2. Create Gaussian Filter:

```
def create_GaussianFilter(kernel_size, sigma):
  kernel = np.zeros((kernel_size, kernel_size), dtype=float)
  for i in range(0, kernel_size):
    for j in range(0, kernel_size):
      x = i - math.floor(kernel_size / 2)
      y = j - math.floor(kernel_size / 2)
      w = math.exp(-(x**2 + y**2) / (2 * sigma**2))/(2*math.pi*sigma**2)
      kernel[i, j] = w
  kernel = kernel/kernel.sum()
  return kernel
```

The values of pixels in Gaussian filter is according to the formula:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

The kernel size and sigma values for each image are as follows:

img1: kernel size=3x3 sigma=0.8

img2: kernel size=3x3 sigma=0.8

img2: kernel size=5x5 sigma=1

3. Create Canny Edge Detector:

```
class CannyEdgeDetector:
    def __init__(self):...
    def cal_gradient(self):...
    def non_maximum_suppression(self):...
    def Hysteresis(self, Tl, Th):...
    def Canny(self, img, Tl, Th):...
```

Create sobel operator:

```
def __init__(self):
    self.sobel_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
    self.sobel_y = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
```

Gradient calculation:

```
def cal_gradient(self):
    self.gradient_x = convolution(self.img, self.sobel_x, 1)
    self.gradient_y = convolution(self.img, self.sobel_y, 1)
    self.gradient_magnitude = np.sqrt(self.gradient_x**2 + self.gradient_y**2)
    self.gradient_magnitude = (255 * np.int32(self.gradient_magnitude > 255) + self.gradient_magnitude * np.int32(self.gradient_magnitude <= 255))
    self.gradient_direction = np.arctan2(self.gradient_y, self.gradient_x)
    self.gradient_direction += math.pi * np.int32(self.gradient_direction < 0)
```

After applying the Sobel operator in both vertical and horizontal directions, as well as convolving the blur image with sobel operator, obtain the gradient magnitude and gradient direction using the following formulas:

gradient magnitude: $G = ((G_x)^2 + (G_y)^2)^{1/2}$

gradient direction: $\tan^{-1}(G_y/G_x)$

Non-maximum suppression:

```
def non_maximum_suppression(self):
    self.suppressed_magnitude=np.zeros_like(self.gradient_magnitude)
    h, w = self.suppressed_magnitude.shape

    for i in range(1, h-1):
        for j in range(1, w-1):
            angle = self.gradient_direction[i, j]
            if (angle>=0 and angle<22.5) or (angle>=157.5 and angle<=180):
                q = self.gradient_magnitude[i, j+1]
                r = self.gradient_magnitude[i, j-1]
            elif (angle>=22.5 and angle<67.5):
                q = self.gradient_magnitude[i+1, j-1]
                r = self.gradient_magnitude[i-1, j+1]
            elif (angle>=67.5 and angle<112.5):
                q = self.gradient_magnitude[i+1, j]
                r = self.gradient_magnitude[i-1, j]
            elif (angle>=112.5 and angle<157.5):
                q = self.gradient_magnitude[i-1, j-1]
                r = self.gradient_magnitude[i+1, j+1]

            if self.gradient_magnitude[i, j] >= max(q, r):
                self.suppressed_magnitude[i, j] = self.gradient_magnitude[i, j]
            else:
                self.suppressed_magnitude[i, j] = 0
```

Going through each pixel in the gradient magnitude image and, based on the corresponding gradient direction, compares the magnitude of the current pixel with its neighbors along the gradient direction. If the magnitude of the current pixel is greater than or equal to the magnitudes of its neighbors, it is considered a local maximum and retained in the suppressed_magnitude image; otherwise, it is set to zero.

Double threshold and Edge Tracking by Hysteresis:

```
def Hysteresis(self, Tl, Th):
    self.result = np.zeros_like(self.suppressed_magnitude)
    for i in range(0, self.result.shape[0]):
        for j in range(0, self.result.shape[1]):
            if self.suppressed_magnitude[i, j]>=Tl:
                il = i-1
                ir = i+1
                jl = j-1
                jr = j+1
                if il<0: il = 0
                if jl<0: jl = 0
                if ir>=self.suppressed_magnitude.shape[0]: ir = self.suppressed_magnitude.shape[0]-1
                if jr>=self.suppressed_magnitude.shape[1]: jr = self.suppressed_magnitude.shape[1]-1
                pixels = self.suppressed_magnitude[il:ir, jl:jr]
                for x in range(0, pixels.shape[0]):
                    for y in range(0, pixels.shape[1]):
                        if pixels[x, y]>=Th:
                            self.result[i, j]=255
```

The method takes two threshold values, Tl and Th. It iterates through each pixel in the suppressed magnitude image. if the magnitude at a pixel is greater than Tl and any of the neighboring pixels have a magnitude greater than Th, the pixel in the result image corresponding to the current pixel position is set to 255.

4. Hough Transform

```
def houghLine(image):
    height = image.shape[0]
    width = image.shape[1]

    dist_max = int(np.round(np.sqrt(width ** 2 + height ** 2)))
    thetas = np.deg2rad(np.arange(-90, 90))
    rs = np.linspace(-dist_max, dist_max, 2*dist_max)
    accumulator = np.zeros((2 * dist_max, len(thetas)))
    for y in range(height):
        for x in range(width):
            if image[y,x] > 0:
                for k in range(len(thetas)):
                    r = x*np.cos(thetas[k]) + y * np.sin(thetas[k])
                    accumulator[int(r) + dist_max,k] += 1
    return accumulator, thetas
```

The Hough Transform accumulator array is then initialized with zeros. The function iterates through each pixel in the input image, and if the pixel value is greater than 0(indicating an edge or a line), it calculates the corresponding polar coordinates (r, theta) for each angle in the range.

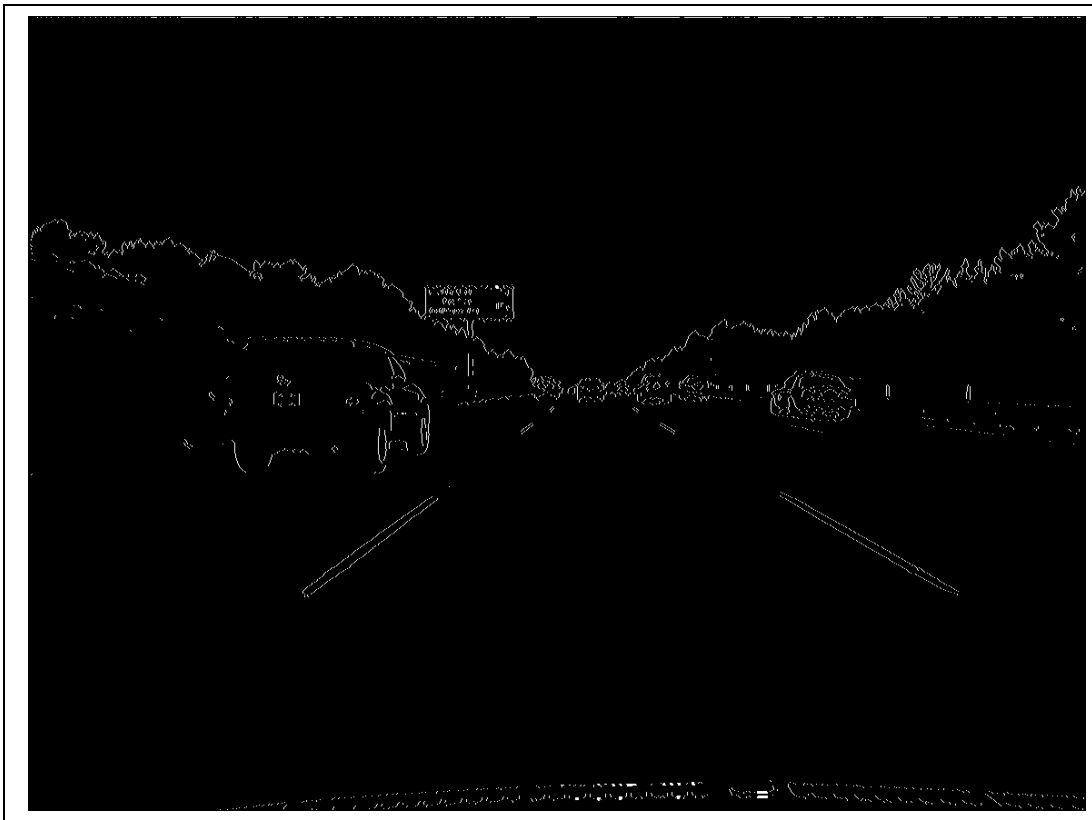
```python
def hough_peaks(accumulator, threshold):
    peaks = []
    while True:
        max_value = np.max(accumulator)

        if max_value >= threshold:
            idx = np.argmax(accumulator)
            rho_idx, theta_idx = np.unravel_index(idx, accumulator.shape)
            peaks.append((rho_idx - accumulator.shape[0] // 2, theta_idx))
            accumulator[rho_idx, theta_idx] = 0
        else:
            break
    return peaks
```

The function iteratively finds the maximum value in the accumulator, compares it to a given threshold, and if it exceeds the threshold, records the indices of the maximum value (interpreted as rho and theta values in Hough space). The process is repeated until no peak surpasses the threshold, and the function returns a list of identified peaks as (rho, theta) pairs.

5. Input images and output images

img1

img2



img3