Computer Vision

NTUT 資訊工程系 111598070 賴俊霖

Assignment 3

Q1. Gaussian Blur (result_img1)

Gaussian Filter：

Gaussian Filter

- Define Filter Size (3*3)
- Define Sigma σ

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

step1

$$\begin{bmatrix}(-1,-1) & (0,-1) & (1,-1)\\ (-1,0) & (0,0) & (1,0)\\ (-1,1) & (0,1) & (1,1)\end{bmatrix}$$

step2

$$\begin{bmatrix}0.045 & 0.122 & 0.045\\ 0.122 & 0.332 & 0.122\\ 0.045 & 0.122 & 0.045\end{bmatrix}$$

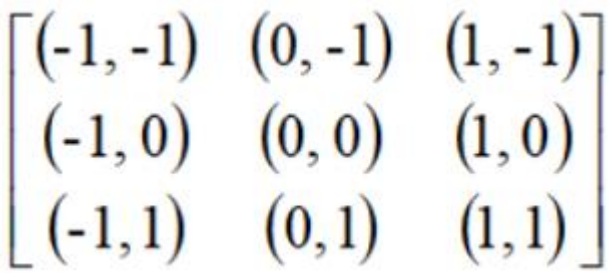
step3

Although the equation above has 2*pi* σ ,we can ignore the scale factor , and the equation will look like this $g(x) = e^{-\frac{x^2}{2\sigma^2}}$ to simplify our calculations.

And it will be like below.

```
x, y = np.mgrid[-1:2, -1:2]
gaussian_kernel = np.exp(-(x ** 2 + y ** 2))

# Normalization
gaussian_kernel = gaussian_kernel / gaussian_kernel.sum()
```

As you can see , np.mgrid[-1:2,-1:2] is used to gernerate the matrix

like this 

.

And we use np.exp(-(x**2+y**2)) which is the formula we simplified.

After using np.exp, we still need to normalize the matrix, and get the matrix below.

```
[[0.04491922 0.12210311 0.04491922]
 [0.12210311 0.33191066 0.12210311]
 [0.04491922 0.12210311 0.04491922]]
```

```
def gaussianBlur(img):

    x, y = np.mgrid[-1:2, -1:2]
    gaussian_kernel = np.exp(-(x ** 2 + y ** 2))

    # Normalization
    gaussian_kernel = gaussian_kernel / gaussian_kernel.sum()

    # print(gaussian_kernel)
    colized = convolution(gaussian_kernel,img)
```

Q2. Canny Edge Detection (result_img2)

Gradient Calculation :

```python
def gradientCal(gx,gy):
    G = np.hypot(gx,gy)
    G = G / G.max() * 255
    theta = np.arctan2(gy,gx)
    return (G,theta)
```

```python
def CannyEdgeDetect(img):
    Gx = np.array([[-1, 0, 1],
                   [-2, 0, 2],
                   [-1, 0, 1]])
    Gy = np.array([[-1, -2, -1],
                   [0, 0, 0],
                   [1, 2, 1]])
    img_gx = convolution(Gx,img)
    img_gy = convolution(Gy,img)

    mag = np.hypot(img_gx,img_gy)
    mag *=255.0 / np.max(mag)

    gradMat , theta = gradientCal(img_gx,img_gy)
```

We can convolve the image with sobel kernel to get the horizontalImg and verticalImg. And they can help us to get magnitude and directions for each pixel in the image.

G_x :

-1	0	1
-2	0	2
-1	0	1

G_y :

-1	-2	-1
0	0	0
1	2	1

Non-Maximum Suppression

There are some problems about the output of Gradient Calculation , some edges are thick , but we wanna thin edges , so we gonna use

nonmaximum suppression to fix them.

```
def nonmaxSuppression(img,D):
    m,n = img.shape
    copyed = np.zeros((m, n), dtype=np.int32)
    angle = D * 180 / np.pi
    angle[ angle<0 ] += 180

    for i in range(1,m-1):
        for j in range(1,n-1):
            try:
                q = 255
                r = 255

                if(0 <= angle[i,j] < 22.5) or (157.5 <= angle[i,j] <= 180):
                    q = img[i,j+1]
                    r = img[i,j-1]
                elif(22.5 <= angle[i,j] < 67.5):
                    q = img[i + 1, j - 1]
                    r = img[i - 1, j + 1]
                elif (67.5 <= angle[i, j] < 112.5):
                    q = img[i + 1, j]
                    r = img[i - 1, j]
                # angle 135
                elif (112.5 <= angle[i, j] < 157.5):
                    q = img[i - 1, j - 1]
                    r = img[i + 1, j + 1]

                if (img[i, j] >= q) and (img[i, j] >= r):
                    copyed[i, j] = img[i, j]
                else:
                    copyed[i, j] = 0

            except IndexError as e:
                pass

    return copyed
```

We create a matrix initialized to zero of the same size of the

gradient magnitude matrix.

And identify the edge direction based on the angle value from the

angle matrix. And we check if the pixel in the same direction has a higher magnitude than other pixels that is currently processed.

After processing, we get a same image but with thinner edges.

Double Threshold

Strong pixels are those with an intensity so high that we are sure they should be the final edge in the image.

And weak pixels may not enough to be considered as strong ones, but not small enough to be considered as null.

```
def threshold(img, lowthresholdRatio,highthresholdRatio):
    highRatio = img.max() * highthresholdRatio
    lowRatio = highRatio * lowthresholdRatio

    m,n = img.shape
    res = np.zeros((m,n),dtype=np.int32)

    weak = np.int32(25)
    strong = np.int32(255)

    strong_i, strong_j = np.where(img >= highRatio)

    weak_i, weak_j = np.where((img <= highRatio) & (img >= lowRatio))

    res[strong_i, strong_j] = strong
    res[weak_i, weak_j] = weak

    return res
```

```
# 1
#     thresholdImg = threshold(non,lowthresholdRatio=0.09,highthresholdRatio=0.5)
# 2
#     thresholdImg = threshold(non,lowthresholdRatio=0.03,highthresholdRatio=0.8)
# 3
#     thresholdImg = threshold(non,lowthresholdRatio=0.01,highthresholdRatio=0.1)
# 4
    thresholdImg = threshold(non,lowthresholdRatio=0.01,highthresholdRatio=0.8)
```

Pic above are the low / high threshold for every pics.

Edge Linking

After thresholding, the Edge Linking consists of transforming weak pixels into strong ones, if and only if at least one of the pixels around the one being processed is a strong one

```
def EdgeLinking(img, weak=25, strong=255):
    m, n = img.shape
    for i in range(1, m-1):
        for j in range(1, n-1):
            if (img[i,j] == weak):
                try:
                    if ((img[i+1, j-1] == strong) or
                        (img[i+1, j] == strong) or
                        (img[i+1, j+1] == strong) or
                        (img[i, j-1] == strong) or
                        (img[i, j+1] == strong) or
                        (img[i-1, j-1] == strong) or
                        (img[i-1, j] == strong) or
                        (img[i-1, j+1] == strong)):

                        img[i, j] = strong
                    else:
                        img[i, j] = 0
                except IndexError as e:
                    pass
    return img
```

Canny

```
def CannyEdgeDetect(img):
    Gx = np.array([[-1, 0, 1],
                   [-2, 0, 2],
                   [-1, 0, 1]])
    Gy = np.array([[-1, -2, -1],
                   [0, 0, 0],
                   [1, 2, 1]])
    img_gx = convolution(Gx,img)
    img_gy = convolution(Gy,img)

    mag = np.hypot(img_gx,img_gy)
    mag *=255.0 / np.max(mag)

    gradMat , theta = gradientCal(img_gx,img_gy)

    non = nonmaxSuppression(mag,theta)

# thresholdImg = threshold(non,lowthresholdRatio=0.05,highthresholdRatio=0.09)

# 1
# thresholdImg = threshold(non,lowthresholdRatio=0.09,highthresholdRatio=0.5)
    thresholdImg = threshold(non,lowthresholdRatio=0.01,highthresholdRatio=0.5)

    imgEdgeLinking = EdgeLinking(thresholdImg)

    cv2.imwrite('result_img2.jpg', imgEdgeLinking)

    return imgEdgeLinking
```

Q3. Hough Transform(result_img3)

First we need to convert the traditional Cartesian coordinate system to polar coordinates, so that the computer will not encounter infinite size when computing.

Then we have to define the scope of Rho and Theta.

```
thetas = np.deg2rad(np.arange(-90.0, 90.0, angle_step))
m, n = img.shape
diagLen = int(round(math.sqrt(m * m + n * n)))
rhos = np.linspace(-diagLen, diagLen, diagLen * 2)
```

And we create a 2D matrix initialized to zero and the rows and columns are equal to the number of ros values and the number of thetas

The values higher than threshold should be indexes to edges.

```
accumulator = np.zeros((2 * diagLen, num_thetas), dtype=np.uint8)

edges = img > value_threshold if 1 else img < value_threshold

y_idxs, x_idxs = np.nonzero(edges)
```

Voting in the accumulator, for each edge point and for each theta value, find the nearest rho value and increment that index in the accumulator.

```
for i in range(len(x_idxs)):
    x = x_idxs[i]
    y = y_idxs[i]

    for t_idx in range(num_thetas):
        rho = diagLen + int(round(x * cos_t[t_idx] + y * sin_t[t_idx]))
        accumulator[rho, t_idx] += 1

m = accumulator.shape[0]
n = accumulator.shape[1]
accumulator = accumulator.reshape((m * n))
```

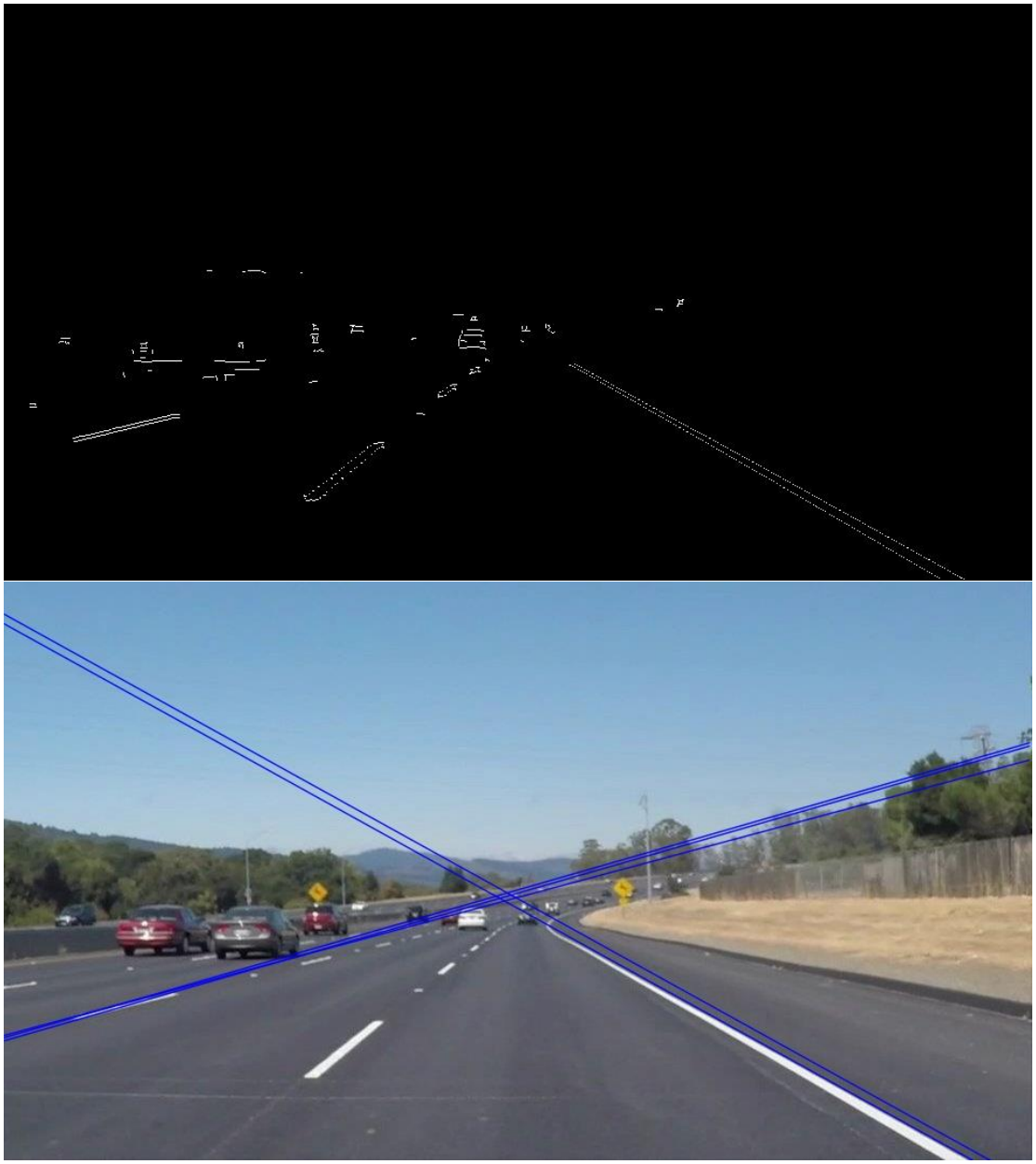
Local maximum in the accumulator indicates the parameters of the most prominent lines in the images we input. And the maximum maybe the possible lines for every index in accumulator matrix.

```
def peak_vote(acc,thetas,rhos):
    idx = np.argmax(acc)
    rho = rhos[int(idx / n)]
    theta = thetas[idx % n]

    return idx,theta,rho
```

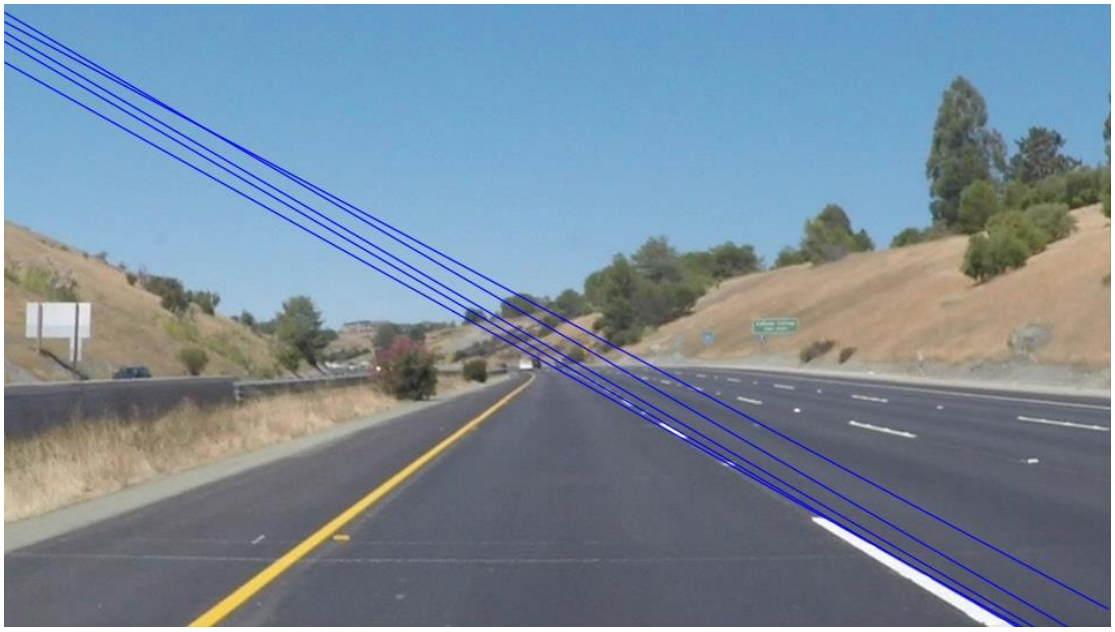
```python
def hough_line(img, angle_step=1, value_threshold=120):

    thetas = np.deg2rad(np.arange(-90.0, 90.0, angle_step))
    m, n = img.shape
    diagLen = int(round(math.sqrt(m * m + n * n)))
    rhos = np.linspace(-diagLen, diagLen, diagLen * 2)

    cos_t = np.cos(thetas)
    sin_t = np.sin(thetas)
    num_thetas = len(thetas)

    accumulator = np.zeros((2 * diagLen, num_thetas), dtype=np.uint8)

    edges = img > value_threshold if 1 else img < value_threshold

    y_idxs, x_idxs = np.nonzero(edges)

    for i in range(len(x_idxs)):
        x = x_idxs[i]
        y = y_idxs[i]

        for t_idx in range(num_thetas):
            rho = diagLen + int(round(x * cos_t[t_idx] + y * sin_t[t_idx]))
            accumulator[rho, t_idx] += 1

    m = accumulator.shape[0]
    n = accumulator.shape[1]
    accumulator = accumulator.reshape((m * n))

    def peak_vote(acc,thetas,rhos):
        idx = np.argmax(acc)
        rho = rhos[int(idx / n)]
        theta = thetas[idx % n]

        return idx,theta,rho
```

```
for i in range(5):
    idx , theta , rho = peak_vote(accumulator,thetas,rhos)

    y1 = int((rho) / math.sin(theta))
    x2 = img.shape[1]
    y2 = int((rho - (x2 * math.cos(theta))) / math.sin(theta))

    point1 , point2 = (0,y1) , (x2,y2)

    cv2.line(img2,point1,point2, (255, 0, 0), 1, cv2.LINE_AA)

    np.delete(accumulator, [0])
    accumulator = np.delete(accumulator, np.where(accumulator == accumulator[idx]))

cv2.imwrite('result_img3.jpg', img2)
```

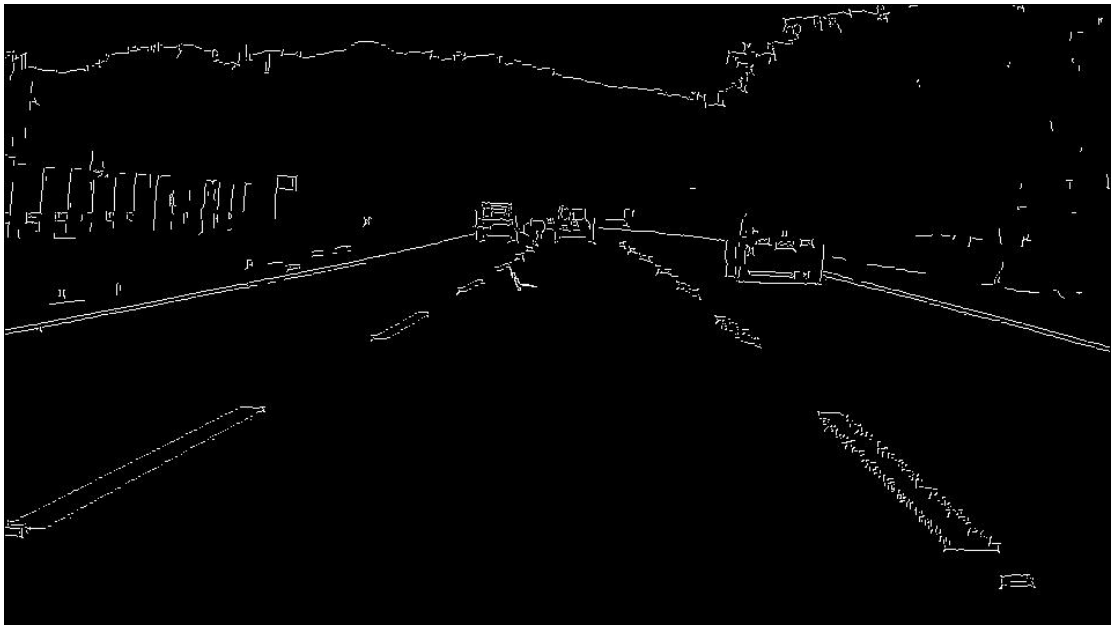
Output of Pic 1

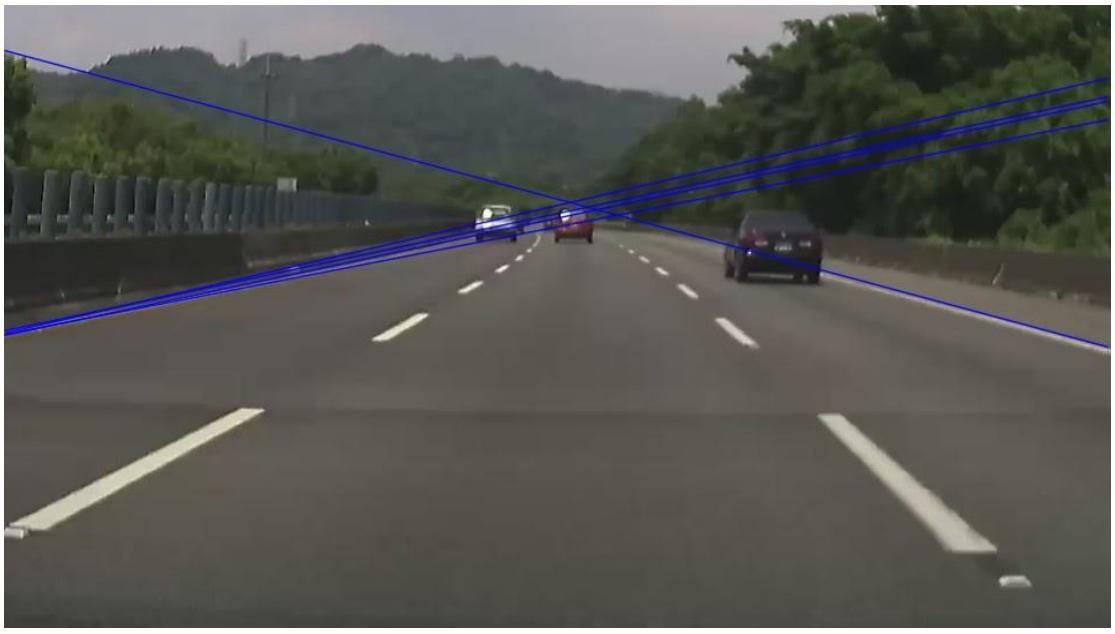


Output of Pic 2



Output of Pic 3





Output of Pic 4