**Program 2 Report**

清華研 張睿

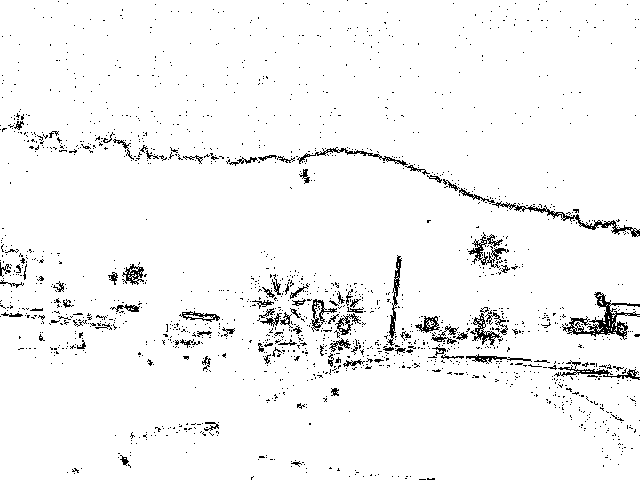
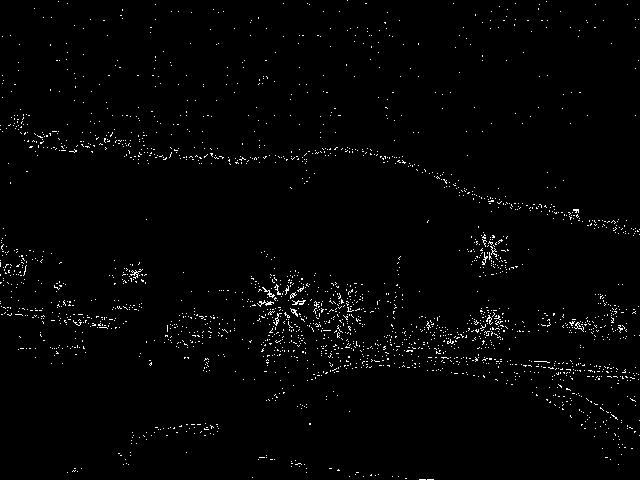
交大學號：A081605

**p1im3.bmp**

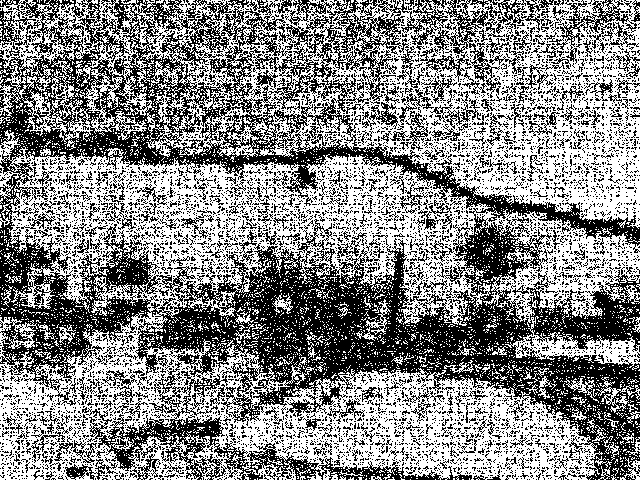
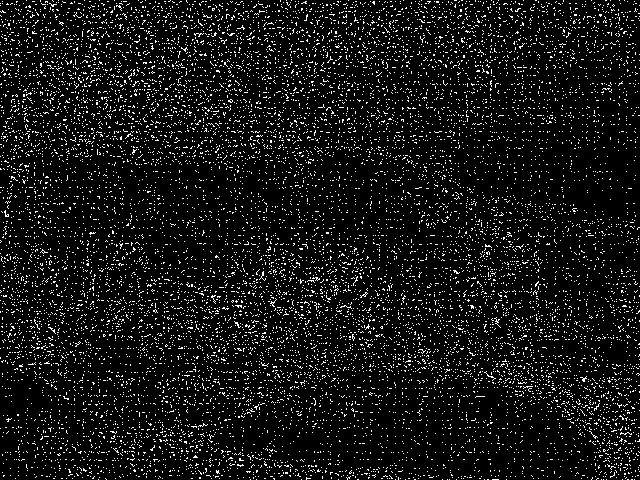
**Task #1 – Edge Detection:**

* LoG filter:

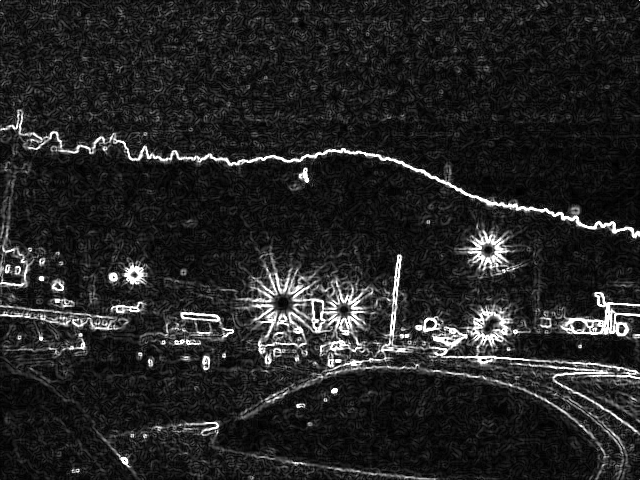
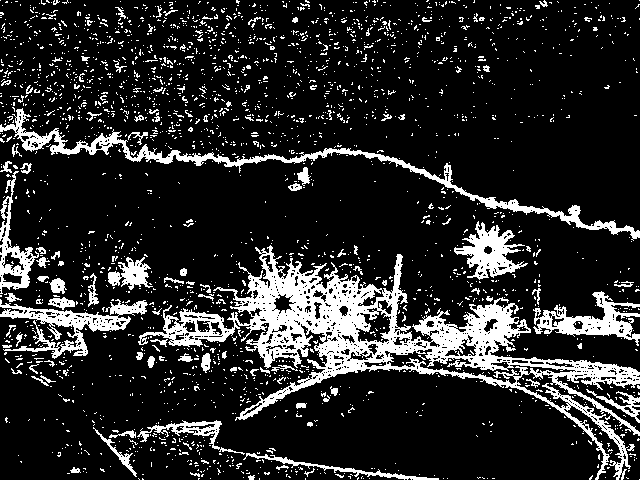
To perform edge detection with LoG, I took the idea of zero-cross so I highlight the pixels (set its value to 255) whose values of LoG are close to zero (left figure below). However, the pixels in the flat regions in the original image also appear to be close to zero after LoG filtering. To find the edges only, I highlight the pixels whose values are in a range slightly above zero, e.g. [3, 5]. (Right figure below). Possibly because of the sensitivity to noise, the edges are always broken.

** **

* Laplacian filter has the effects similar to LoG. Furthermore, Laplacian is much more sensitive to noise. (The two images below are corresponded to the two images above, the only difference is that they are from Laplacian filter)

* The Sobel magnitude map (left image below) looks good on visualizing edges, but our goal is to select exact edge pixels for segmentation. Therefore, we need to threshold out those pixels to be the selected pixels (if we are not performing Canny edge detection later). I choose a threshold (0.3 \* Otsu’s threshold of the Sobel magnitude map) which can just remain the as much edges as possible. (Right image below) Unfortunately, by this threshold, lots of noises are also remained. Furthermore, the edges are thick.

* If we do some preprocessing (gamma correction & adaptive local noise reduction filter), the noise can be reduced but the edges get thicker.



**Task #2 – Canny edge detector:**

* Comparing with the result above, the NMS and double thresholding let the edge thinner and eliminate the noises in the sky.



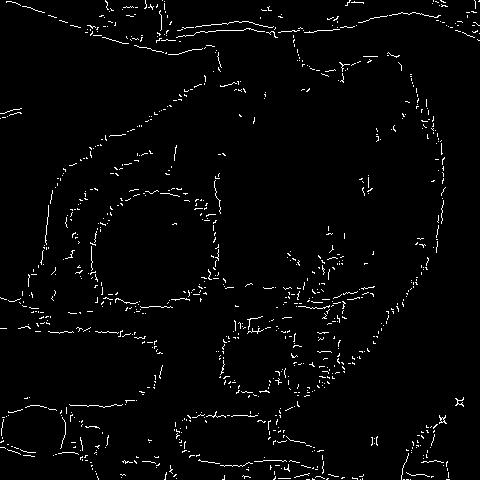
* The double thresholding parameters are as follows:

(Otsu’s threshold/2, Otsu’s threshold/3, iter=1, Square structure element size=11)

* The implementation and the threshold-choosing strategy are same as the discussion in **p1im5**.

**p1im6.bmp**

* Applied method and result:
  + Preprocessing: adaptive local noise reduction filter (filter size = 9, estimate noise = 100)
  + Gradient filter: Sobel filter (size = 7)
  + Non-maximum suppression
  + Double thresholds: (Otsu’s threshold, Otsu’s threshold / 2)



* Discussion:

|  |  |  |
| --- | --- | --- |
| Pre-  process | Gaussian filter | Adaptive local noise reduction filter |
| Final result |  |  |
| discussion | Detected edges are more broken then the right if the edges are not so noisy in the original image | Detected edges are more broken then the left if the edges are much noisy in the original image |

|  |  |  |
| --- | --- | --- |
| Grad. filter | Sobel (size=3) | Sobel (size=7) |
| Final result |  |  |
| discussion | Possibly because of the low resolution of the original image, there are many detected edges with “double horizontal/vertical lines”. |  |

|  |  |  |
| --- | --- | --- |
| Higher thres. in the double thres. | Otsu’s threshold | Otsu’s threshold \* 1.3 |
| Final result |  |  |
| discuss |  | Even though the lower threshold remains the same, if the higher threshold is too high, the result image misses some edge. It’s because we **start from the strong edges** which gradient magnitude are higher than the higher threshold. |

**p1im5.bmp**

* Applied method and result:
  + Sobel filter (size=3)
  + Non-maximum suppression
  + Double-thresholding
    - (Otsu’s threshold, Otsu’s threshold / 2)
    - Edge tracking: Select weak edge points with the dilation of the already selected edge points.
      * Square structure element size = 11
      * iter = 7

p.s. My edge tracking algorithm is roughly as follows:

selected = strong  
for \_ in range(iter):  
 mask = dilation(selected, np.ones((SEsize, SEsize))  
 selectedWeak = np.logical\_and(weak, mask)  
 selected = np.logical\_or(selected, selectedWeak)  
return selected

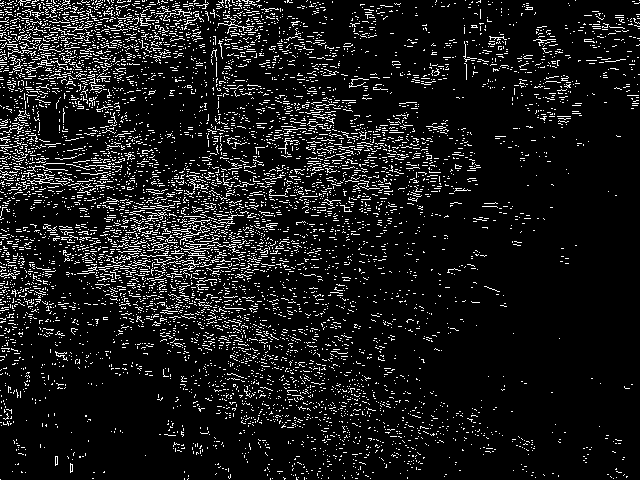


* Discussion
  + The edges are relatively clear so I didn’t preprocess this image.
  + For choosing the two thresholds:
    - The goal is to remain as much edge details of the objects on the ground as possible without including the edge points in the sky.
    - Otsu’s threshold is a good choice to be the higher threshold because the edge points in the sky are eliminated.
    - For the smaller threshold, Otsu’s/2 is better than Otsu’s/3 because the sky edges are sparser in the former and they are less possible to be reached by the edge tracking algorithm.

|  |  |
| --- | --- |
| thresholds | Points pass the thresholds |
| Otsu’s |  |
| Otsu’s / 2 |  |
| Otsu’s / 3 |  |

**p1im1**

* To this image, preprocessing (right image below) significantly improves the edge detection result comparing to the image without preprocessing (left image below) because there are some dark regions with edge information.
* Preprocessing: gamma correction & adaptive local noise reduction filter (following the method in my program 1)

**Code listing**

**spatialFiltering.py** (p10)

correlation

gaussianFilter

sobelFilter

LoGFilter

adaptiveLocalNoiseReductionFilter

bilateralFilter

**intensityTransformation.py (p19)**

histogramEqual

gammaCorrelation

normalizeFullDomain i.e. map to [0, 255]

**cannyRelated.py (p)**

nms

otsu

doubleThresholds

**p1im1.py**

**spatialFiltering.py**

import numpy as np  
import math  
import pickle  
  
filters2d = {  
 "sobel\_vertical\_3": np.array([[-1, 0, 1],  
 [-2, 0, 2],  
 [-1, 0, 1]]),  
 "sobel\_vertical\_5": np.array([[-2/8, -1/5, 0, 1/5, 2/8],  
 [-2/5, -1/2, 0, 1/2, 2/5],  
 [-2/4, -1/1, 0, 1/1, 2/4],  
 [-2/5, -1/2, 0, 1/2, 2/5],  
 [-2/8, -1/5, 0, 1/5, 2/8]]),  
 "sobel\_vertical\_7": np.array([[-3/18, -2/13, -1/10, 0, 1/10, 2/13, 3/18],  
 [-3/13, -2/8, -1/5, 0, 1/5, 2/8, 3/13],  
 [-3/10, -2/5, -1/2, 0, 1/2, 2/5, 3/10],  
 [-3/9, -2/4, -1/1, 0, 1/1, 2/4, 3/9],  
 [-3/10, -2/5, -1/2, 0, 1/2, 2/5, 3/10],  
 [-3/13, -2/8, -1/5, 0, 1/5, 2/8, 3/13],  
 [-3/18, -2/13, -1/10, 0, 1/10, 2/13, 3/18]]),  
 "laplacian\_3": np.array([[0, -1, 0],  
 [-1, 4, -1],  
 [0, -1, 0]]),  
 "laplacian\_5": np.array([[-4, -1, 0, -1, -4],  
 [-1, 2, 3, 2, -1],  
 [0, 3, 4, 3, 0],  
 [-1, 2, 3, 2, -1],  
 [-4, -1, 0, -1, -4]]),  
 "laplacian\_7": np.array([[-10, -5, -2, -1, -2, -5, -10],  
 [-5, 0, 3, 4, 3, 0, -5],  
 [-2, 3, 6, 7, 6, 3, -2],  
 [-1, 4, 7, 8, 7, 4, -1],  
 [-2, 3, 6, 7, 6, 3, -2],  
 [-5, 0, 3, 4, 3, 0, -5],  
 [-10, -5, -2, -1, -2, -5, -10]])  
}  
  
  
def correlation(img, filter, repeat=True):  
 *"""  
 valid input depths:  
 img: 1, filter: 1  
 img: 3, filter: 3  
 img: 3, filter: 1  
 """* ### filter must be a square with odd H and W  
 if filter.shape[0] != filter.shape[1] or filter.shape[0] % 2 == 0 or filter.shape[0] % 2 == 0:  
 print("correlation: invalid filter!!")  
 return  
 reshape = False  
 if len(img.shape) == 2:  
 img = img[:, :, np.newaxis]  
 reshape = True  
 if len(filter.shape) == 2:  
 filter = filter[:, :, np.newaxis]  
 if img.shape[2] == 3 and filter.shape[2] == 1:  
 filter = np.repeat(filter, 3, axis=2)  
  
 pad = int((filter.shape[0] - 1) / 2)  
 size = filter.shape[0]  
  
 result = np.zeros(img.shape)  
 paddedImg = np.zeros((img.shape[0] + 2 \* pad, img.shape[1] + 2 \* pad, img.shape[2]))  
 paddedImg[pad: -pad, pad: -pad, :] = img  
 if repeat:  
 paddedImg[:pad, pad: -pad, :] = img[0, :, :].reshape((1, img.shape[1], img.shape[2]))  
 paddedImg[-pad:, pad: -pad, :] = img[-1, :, :].reshape((1, img.shape[1], img.shape[2]))  
 paddedImg[pad: -pad, :pad, :] = img[:, 0, :].reshape((img.shape[0], 1, img.shape[2]))  
 paddedImg[pad: -pad, -pad:, :] = img[:, -1, :].reshape((img.shape[0], 1, img.shape[2]))  
  
 for i in range(img.shape[0]):  
 for j in range(img.shape[1]):  
 result[i, j, :] = np.sum(paddedImg[i: i + size, j: j + size, :] \* filter, axis=(0, 1))  
  
 if reshape:  
 img = np.reshape(img, (img.shape[0], img.shape[1]))  
 result = np.reshape(result, (result.shape[0], result.shape[1]))  
  
 return result  
  
  
def calculateGaussianFilter(size, version, sd=5):  
 *"""* ***:param*** *sd: the standard deviation, is only used in version 0  
 """* if version == 0:  
 centerIndex = size // 2  
  
 xIndex = np.tile(np.arange(size), (size, 1))  
 xDiff = xIndex - centerIndex # x diff with the center  
 yIndex = np.tile(np.arange(size).reshape(-1, 1), (1, size))  
 yDiff = yIndex - centerIndex # y diff with the center  
 numerator = xDiff \*\* 2 + yDiff \*\* 2  
  
 filter2d = (1 / (2 \* math.pi \* sd \*\* 2)) \* np.exp(-(numerator / (2 \* sd \*\* 2)))  
  
 filter2d = filter2d / np.sum(filter2d)  
  
 return filter2d  
  
 N = (size - 1) / 2  
  
 filter1d = np.arange(-N, N + 1)  
 numerator = (3 \* filter1d / N) \*\* 2  
 filter1d = np.exp(-(numerator / 2))  
 filter1d = filter1d / np.sum(filter1d)  
  
 filter2d = np.tile(filter1d, (size, 1)) \* np.tile(filter1d.reshape(-1, 1), (1, size))  
  
 return filter2d  
  
  
def gaussianFilter(img, size, version=0, sd=5):  
 *"""* ***:param*** *img: could be gray scale or 3-channel image  
 """* filter = calculateGaussianFilter(size, version, sd)  
 return correlation(img, filter)  
  
  
def sobelFilter(img, size):  
 *"""* ***:param*** *img: could be gray scale or 3-channel BGR image* ***:param*** *size: must be either 3, 5, or 7  
 """* print("sobel filter")  
  
 global filters2d  
  
 if len(img.shape) == 3:  
 if img.shape[2] == 3:  
 img = 0.11 \* img[:, :, 0] + 0.59 \* img[:, :, 1] + 0.3 \* img[:, :, 2]  
  
 filterV = filters2d["sobel\_vertical\_" + str(size)]  
 filterH = filterV.T  
 xGrad = correlation(img, filterV)  
 yGrad = correlation(img, filterH)  
 mag = np.abs(xGrad) + np.abs(yGrad)  
 dir = np.arctan2(yGrad, xGrad)  
 return mag, dir  
  
  
def LoGFilter(img, size, version=0, sd=5, withoutGaussian=False):  
 *"""* ***:param*** *img: could be gray scale or 3-channel BGR image* ***:param*** *size: must be either 3, 5, or 7  
 """* print("LoG filter")  
  
 global filters2d  
  
 if len(img.shape) == 3:  
 if img.shape[2] == 3:  
 img = 0.11 \* img[:, :, 0] + 0.59 \* img[:, :, 1] + 0.3 \* img[:, :, 2]  
  
 filter = filters2d["laplacian\_" + str(size)]  
 if not withoutGaussian:  
 filter = filter \* calculateGaussianFilter(size, version, sd)  
 return correlation(img, filter)  
  
  
# def prewittFilter(img):  
  
  
def adaptiveLocalNoiseReductionFilter(img, size, estNoise, imgName="imgName"):  
 *""" imgName is for .pkl saving """* pklName = "alnrf\_" + str(imgName) + "\_" + str(size) + ".pkl"  
 try:  
 diffFromAvg = pickle.load(open(pklName, "rb"))  
 except (OSError, IOError) as e:  
  
 tmpShape = list(img.shape)  
 tmpShape.append(size \* size)  
 tmpShape = tuple(tmpShape)  
 diffFromAvg = np.zeros(tmpShape)  
  
 for i in range(size \* size):  
 print("iter: ({}/{})".format(str(i), str(size \* size)))  
 subFilter = np.zeros(size \* size)  
 subFilter[i] = 1  
 subFilter = np.reshape(subFilter, (size, size))  
 filter = subFilter - np.ones((size, size)) / (size \* size)  
 diffFromAvg[..., i] = correlation(img, filter)  
  
 pickle.dump(diffFromAvg, open(pklName, "wb"))  
  
 localVarImg = np.sum(diffFromAvg \*\* 2, axis=-1) / (size \* size)  
  
 # print(localVarImg[:10, :10, 0])  
  
 localMean = correlation(img, np.ones((size, size)) / (size \* size))  
  
 factor = estNoise / (localVarImg + 0.00001)  
 factor = np.clip(factor, a\_min=None, a\_max=1)  
  
 return img - factor \* (img - localMean)  
  
  
def bilateralFilter(img, size, sdg, imgName="imgName"):  
 *"""  
 valid input depths:  
 img: 1, filter: 1  
 img: 3, filter: 3  
 img: 3, filter: 1  
 imgName is for .pkl saving  
 """* filter = calculateGaussianFilter(size, 0)  
 reshape = False  
 if len(img.shape) == 2:  
 img = img[:, :, np.newaxis]  
 reshape = True  
 if len(filter.shape) == 2:  
 filter = filter[:, :, np.newaxis]  
 if img.shape[2] == 3 and filter.shape[2] == 1:  
 filter = np.repeat(filter, 3, axis=2)  
  
 print("var:")  
 print(np.var(img[:, :, 0]))  
 print(np.var(img[:, :, 1]))  
 print(np.var(img[:, :, 2]))  
  
 pklName = "bilateral\_" + str(imgName) + "\_" + str(size) + ".pkl"  
 try:  
 weightMul = pickle.load(open(pklName, "rb"))  
 except (OSError, IOError) as e:  
  
 ### calculate the weight multiplier for each pixel  
 weightMul = np.zeros((img.shape[0], img.shape[1], size, size, img.shape[2]))  
 subFilterSubtractor = np.zeros((size, size))  
 subFilterSubtractor[size // 2, size // 2] = 1  
 for i in range(size):  
 for j in range(size):  
 print("iter: ({}/{})".format(str(i \* size + j), str(size \* size)))  
 subFilter = np.zeros((size, size))  
 subFilter[i, j] = 1  
 tmpFilter = subFilter - subFilterSubtractor  
 weightMul[:, :, i, j, :] = correlation(img, tmpFilter)  
  
 pickle.dump(weightMul, open(pklName, "wb"))  
  
 weightMul = np.exp(-1 \* weightMul \*\* 2 / (2 \* sdg))  
  
 pad = int((filter.shape[0] - 1) / 2)  
 size = filter.shape[0]  
  
 result = np.zeros(img.shape)  
 paddedImg = np.zeros((img.shape[0] + 2 \* pad, img.shape[1] + 2 \* pad, img.shape[2]))  
 paddedImg[pad: -pad, pad: -pad, :] = img  
  
 for i in range(img.shape[0]):  
 for j in range(img.shape[1]):  
 weight = filter \* weightMul[i, j, ...]  
 weightDivisor = np.sum(weight, axis=(0, 1))  
 weightDivisor = weightDivisor[np.newaxis, np.newaxis, :]  
 weightDivisor = np.repeat(weightDivisor, size, axis=0)  
 weightDivisor = np.repeat(weightDivisor, size, axis=1)  
 result[i, j, :] = np.sum(paddedImg[i: i + size, j: j + size, :] \* weight / weightDivisor, axis=(0, 1))  
  
 # if i < 1 and j < 1:  
 # print((weight / weightDivisor)[:2, :2, 0])  
  
 if reshape:  
 img = np.reshape(img, (img.shape[0], img.shape[1]))  
 result = np.reshape(result, (result.shape[0], result.shape[1]))  
  
 return result

**intensityTransformation.py**

import numpy as np  
  
  
def histogramEqual(img, lowerb=0, upperb=255):  
 grayImg = 0.3 \* img[:, :, 2] + 0.59 \* img[:, :, 1] + 0.11 \* img[:, :, 0]  
 grayImg = grayImg.astype("uint8")  
  
 grayHistogram = np.zeros(256)  
  
 ### histogram of the gray-scale image  
 for i in range(255):  
 grayHistogram[i] = (grayImg == i).sum()  
  
 ### normalize the area to 1  
 grayHistogram = grayHistogram / grayHistogram.sum()  
  
 ### mapping to [0, 255]  
 # grayMappingFunc = np.cumsum(grayHistogram) \* 255  
 grayMappingFunc = np.cumsum(grayHistogram) \* (upperb - lowerb) + lowerb  
  
 grayResult = grayMappingFunc[grayImg]  
 mappingMultiplier = grayResult / grayImg  
 return img \* np.repeat(mappingMultiplier[:, :, np.newaxis], 3, axis=2)  
  
  
def gammaCorrection(img, gamma, respectively=True):  
 *""" the input image must has 3 channels """* if respectively:  
 imgFloat = img.astype("float64")  
 for i in range(0, imgFloat.shape[2]):  
 imgFloat[:, :, i] = imgFloat[:, :, i] / 255  
  
 c = 1  
 correct = lambda t: c \* t \*\* gamma  
 vfunc = np.vectorize(correct)  
 result = vfunc(imgFloat)  
  
 for i in range(0, imgFloat.shape[2]):  
 result[:, :, i] = result[:, :, i] \* 255  
  
 return result  
  
 grayImg = 0.3 \* img[:, :, 2] + 0.59 \* img[:, :, 1] + 0.11 \* img[:, :, 0]  
 grayImg = np.clip(grayImg, a\_min=None, a\_max=255)  
 normalizedGrayImg = grayImg / 255  
  
 c = 1  
 correct = lambda t: c \* t \*\* gamma  
 vfunc = np.vectorize(correct)  
 grayImgResult = vfunc(normalizedGrayImg)  
 grayImgResult = grayImgResult \* 255  
 ratio = grayImgResult / (grayImg + 0.00001)  
  
 return np.clip(img \* np.repeat(ratio[:, :, np.newaxis], 3, axis=2), a\_min=None, a\_max=255)  
  
  
def normalizeFullDomain(img, respectively=False):  
 *""" the input image must has 3 channels """* if respectively:  
 result = np.zeros(img.shape)  
 for i in range(img.shape[2]):  
 result[:, :, i] = (img[:, :, i] - img[:, :, i].min()).astype('int') \* 255 / (  
 img[:, :, i].max() - img[:, :, i].min() + 0.00001)  
 # print((img[:, :, i] - img[:, :, i].min()) \* 255)  
 # print((img[:, :, i] - img[:, :, i].min()) \* 256)  
 return result  
  
 grayImg = 0.3 \* img[:, :, 2] + 0.59 \* img[:, :, 1] + 0.11 \* img[:, :, 0]  
 grayImg = np.clip(grayImg, a\_min=None, a\_max=255)  
 noralizedGrayImg = (grayImg - grayImg.min()) \* 255 / (grayImg.max() - grayImg.min() + 0.00001)  
 ratio = noralizedGrayImg / grayImg  
 return np.clip(img \* np.repeat(ratio[:, :, np.newaxis], 3, axis=2), a\_min=None, a\_max=255)

**cannyRelated.py**

import cv2  
import numpy as np  
import math  
from scipy import ndimage  
  
def nms(mag, dir):  
 *"""* ***:param*** *mag: 2D array* ***:param*** *dir: [-pi, pi], 2D map* ***:return****: 2D array  
 """* ### round the grad direction  
 # to 0~360 deg  
 dir = (dir + math.pi) \* 360 / (2 \* math.pi)  
 dir8 = dir // 45 + 1 \* (dir % 45 > 22.5)  
 dir8 = dir8 - 1 \* (dir8 == 8)  
  
 angleDiff = np.zeros((mag.shape[0], mag.shape[1], 8))  
 paddedMag = np.zeros((mag.shape[0] + 2, mag.shape[1] + 2))  
 paddedMag[1: -1, 1: -1] = mag  
 angleDiff[:, :, 0] = mag - paddedMag[1: -1, 2:]  
 angleDiff[:, :, 1] = mag - paddedMag[: -2, 2:]  
 angleDiff[:, :, 2] = mag - paddedMag[: -2, 1: -1]  
 angleDiff[:, :, 3] = mag - paddedMag[: -2, : -2]  
 angleDiff[:, :, 4] = mag - paddedMag[1: -1, : -2]  
 angleDiff[:, :, 5] = mag - paddedMag[2:, : -2]  
 angleDiff[:, :, 6] = mag - paddedMag[2:, 1: -1]  
 angleDiff[:, :, 7] = mag - paddedMag[2:, 2:]  
  
 # if each position of dir8 corresponds to 0, 1, 2, or 3 (the 3rd index), should we remain or suppress  
 remain = np.zeros((mag.shape[0], mag.shape[1], 4))  
 for i in range(4):  
 remain[:, :, i] = np.logical\_and(angleDiff[:, :, i] > 0, angleDiff[:, :, i+4] > 0)  
 lookRemain = dir8 % 4  
 mask = np.zeros((mag.shape[0], mag.shape[1]), dtype='bool')  
 for i in range(4):  
 mask = np.logical\_or(mask, np.logical\_and(remain[:, :, i], lookRemain == i))  
  
 return mag \* mask  
  
  
def otsu(map):  
 *"""* ***:param*** *map: 2D np array, all element >= 0* ***:return****: the threshold  
 """* map = map.astype('int')  
  
 # hist = np.histogram(map, bins=map.max()+1, range=(0, map.max()))  
 numBin = map.max() + 1  
 hist = np.zeros(numBin)  
 for i in range(numBin):  
 hist[i] = np.sum(map == i)  
 hist = hist / map.size  
  
 histCumuSum = np.zeros(numBin)  
 histCumuSum[0] = hist[0]  
 for i in range(1, numBin):  
 histCumuSum[i] = histCumuSum[i - 1] + hist[i]  
  
 histCumuMean = np.zeros(numBin)  
 histCumuMean[0] = 0  
 for i in range(1, numBin):  
 histCumuMean[i] = histCumuMean[i - 1] + hist[i] \* i  
  
 globalMean = histCumuMean[-1]  
  
 varBtw = np.zeros(numBin)  
 for i in range(1, numBin):  
 varBtw[i] = (globalMean \* histCumuSum[i] - histCumuMean[i])\*\*2 / ((histCumuSum[i]) \* (1 - histCumuSum[i]))  
  
 return np.argmax(varBtw)  
  
  
def doubleThresholds(mag, thres1, thres2, returnStrong=False, iter=1, seSize=3, cross=False):  
 *"""* ***:param*** *mag: 2D array storing gradient magnitude of each pixels* ***:param*** *thres1: larger threshold* ***:param*** *thres2: smaller threshold* ***:return****: 2D bool array  
 """* strong = mag >= thres1  
 if returnStrong:  
 return strong  
 weak = np.logical\_and(mag < thres1, mag >= thres2)  
 cv2.imwrite("outputs/p1im6/sobel\_weak.bmp", weak \* 255)  
  
 selected = strong  
 for \_ in range(iter):  
 if cross:  
 mask = ndimage.morphology.binary\_dilation(selected, np.ones((seSize, 1), dtype='bool'))  
 mask = np.logical\_or(mask, ndimage.morphology.binary\_dilation(selected, np.ones((1, seSize), dtype='bool')))  
 else:  
 mask = ndimage.morphology.binary\_dilation(selected, np.ones((seSize, seSize), dtype='bool'))  
 selectedWeak = np.logical\_and(weak, mask)  
 selected = np.logical\_or(selected, selectedWeak)  
 return selected