

Program 2 Report

清華研 張睿
交大學號：A081605

Introduction

In this program, we compare the effects of different gradient filters for edge detection. After the edge detection, we perform NMS and double-thresholding to make the edge thinner and link the separated edge points. Furthermore, we study the effects of some preprocessing methods.

Methods

- Preprocessing
 - Normalize to the whole domain (i.e. map to $[0, 255]$, such that $\min=0$, $\max=255$)
 - Gamma correlation
 - Adaptive local noise reduction filter
 - Gaussian filter
- Gradient filters
 - Laplacian filter
 - LoG filter
 - Different sizes of Sobel filter
- Canny edge detector related functions
 - Non maximum suppression
 - Otsu's threshold
 - Double thresholding

p1im3.bmp

Task #1 – Edge Detection:

- LoG filter:

To perform edge detection with LoG, I took the idea of zero-crossing 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:

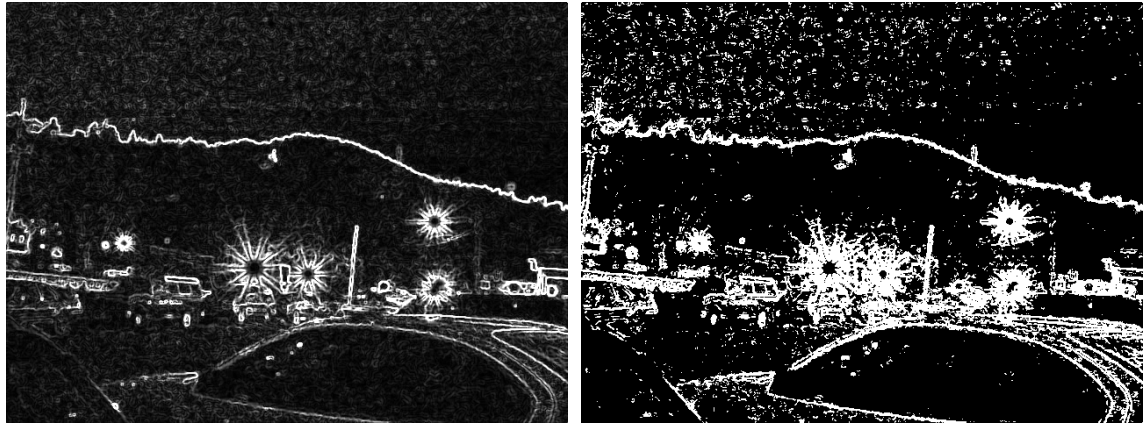
Laplacian filter has the effects similar to LoG. Furthermore, Laplacian is much more sensitive to noise. (The two images below are corresponded to the two images above, the only difference is that they are from Laplacian filter)



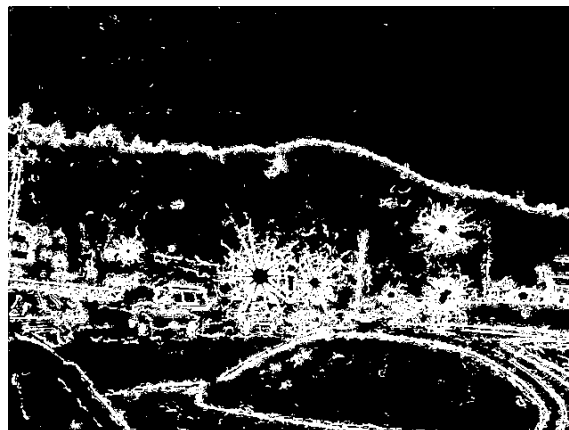
- Sobel filter:

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 ($0.3 * \text{Otsu's threshold of the Sobel magnitude map}$) which can just remain as much meaningful 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 (normalize full domain & gamma correction & adaptive local noise reduction filter), the noise can be reduced but the edges get thicker. (Normalize full domain: normalize the image's pixel values such that min=0 and max=255)



Task #2 – Canny edge detector:

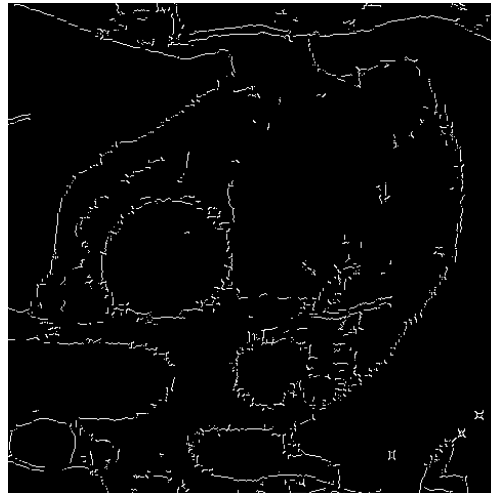
- Comparing with the result above, the NMS and double thresholding let the edge thinner and eliminate the noises in the sky.



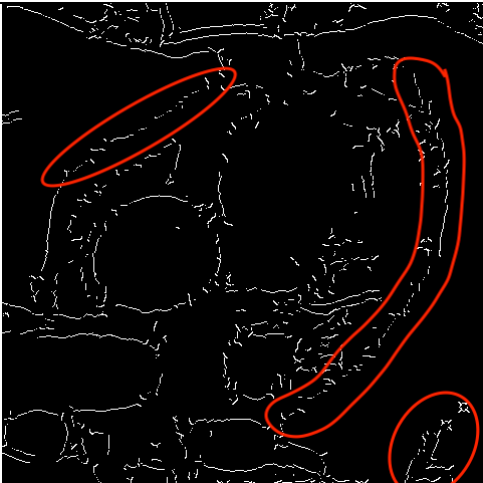
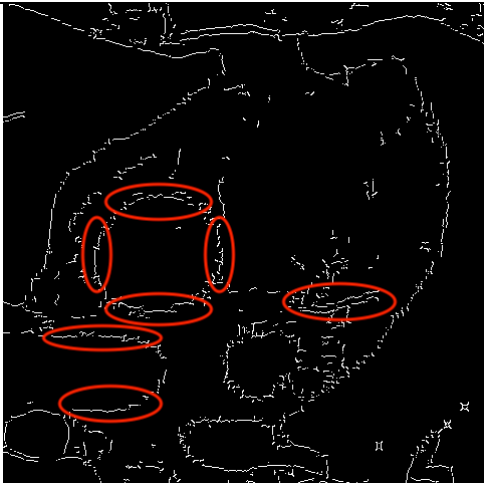
- The double thresholding parameters are as follows:
(Otsu's threshold/2, Otsu's threshold/3, iter=1, Square structure element size=11)
- The implementation and the threshold-choosing strategy are same as the discussion in **p1im5**.


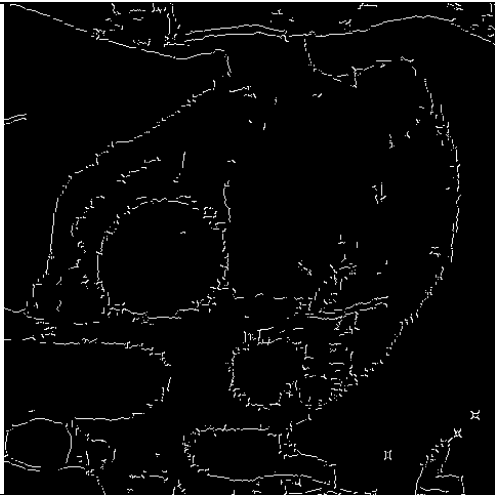
p1im6.bmp

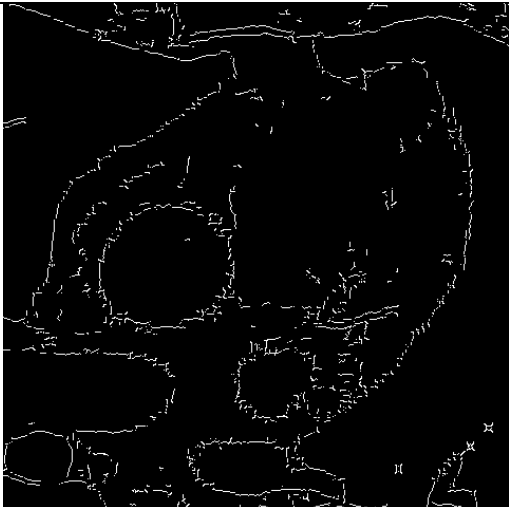
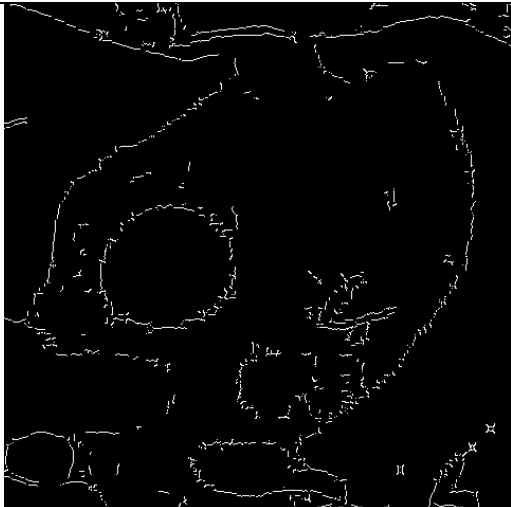
- Applied method and result:
 - Preprocessing: adaptive local noise reduction filter (filter size = 9, estimate noise = 100)
 - Gradient filter: Sobel filter (size = 7)
 - Non-maximum suppression
 - Double thresholds: (Otsu's threshold, Otsu's threshold / 2)



- 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	 The image shows a cityscape with various buildings and trees. The edges of these objects are highlighted in white against a black background. The sky is completely black, indicating that the edges in the sky have been successfully removed. The buildings and trees show clear, well-defined edges, suggesting that the thresholding process effectively preserved the structural details of the ground objects while eliminating background noise.

Otsu's / 2

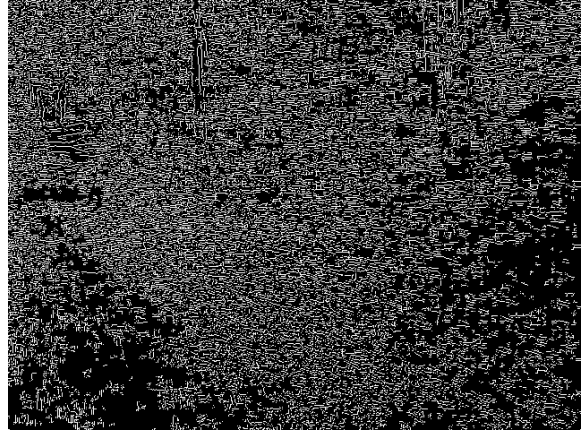
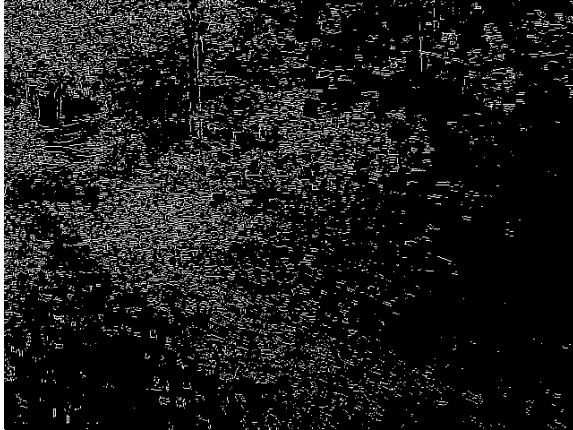


Otsu's / 3



p1im1

- Applied method and result:
 - Preprocessing: normalize full domain & gamma correlation (gamma=0.5)
 - Gradient filter: Sobel filter (size = 3)
 - Non-maximum suppression
 - Double thresholds: (Otsu's threshold, Otsu's threshold / 2)
- 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 in the original image.



Code listing

spatialFiltering.py

- correlation
- gaussianFilter
- sobelFilter
- LoGFilter
- adaptiveLocalNoiseReductionFilter

intensityTransformation.py

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

cannyRelated.py

```
nms
otsu
doubleThresholds
```

p1im3.py

p1im6.py

p1im5.py

p1im1.py

spatialFiltering.py[illegible]

```

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, :] = 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 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)

```

intensityTransformation.py

```

import numpy as np

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

histCumSum = np.zeros(numBin)
histCumSum[0] = hist[0]
for i in range(1, numBin):
    histCumSum[i] = histCumSum[i - 1] + hist[i]

histCumMean = np.zeros(numBin)
histCumMean[0] = 0
for i in range(1, numBin):
    histCumMean[i] = histCumMean[i - 1] + hist[i] * i

globalMean = histCumMean[-1]

varBtw = np.zeros(numBin)
for i in range(1, numBin):
    varBtw[i] = (globalMean * histCumSum[i] -
histCumMean[i])*2 / ((histCumSum[i]) * (1 - histCumSum[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

```

p1im3.py

```

import cv2
import numpy as np
import math
from spatialFiltering import gaussianFilter, sobelFilter,
LoGFilter, adaptiveLocalNoiseReductionFilter
import matplotlib.pyplot as plt
from cannyRelated import nms, doubleThresholds, otsu
from intensityTransformation import normalizeFullDomain,
gammaCorrection
import os

if __name__ == '__main__':
    inputFolder = "data"
    fileName = "p1im3.bmp"
    inputPath = os.path.join(inputFolder, fileName)
    fileNameSplit = os.path.splitext(fileName)[0]
    img = cv2.imread(inputPath)
    outputFolder = os.path.join("outputs", fileNameSplit + "t2")
    if not os.path.exists(outputFolder):
        os.mkdir(outputFolder)
    cv2.imwrite(os.path.join(outputFolder, fileName), img)

    ##### preprocessing
    img = normalizeFullDomain(img, False)
    img = gammaCorrection(img, 0.5, True)
    img = adaptiveLocalNoiseReductionFilter(img, 11, 50,
fileNameSplit)

    ##### edge detection

    for size in [3]:
        mag, dir = sobelFilter(img, size)
        cv2.imwrite(os.path.join(outputFolder,
"sobel_{}_mag.bmp".format(size)), mag)

    ##### Canny edge detection start from magnitude
    & direction maps
    nmsMag = nms(mag, dir)
    cv2.imwrite(os.path.join(outputFolder, "nmsMag.bmp"), nmsMag)

```

```

# plt.hist(mag.ravel(), bins=100)
# plt.show()

thres = otsu(nmsMag)
print("otsu thres:", thres)

iter = 1
seSize = 7
# douThres = doubleThresholds(nmsMag, thres * 1.3, thres * i,
False, iter, seSize)
douThres = doubleThresholds(nmsMag, thres / 2, thres / 3,
False, iter, seSize)
douThresImg = douThres * 255
cv2.imwrite(os.path.join(outputFolder,
"sobel_douThres_{}_{}.bmp".format(seSize, iter)), douThresImg)

```

p1im6.py

```

import cv2
import numpy as np
import math
from spatialFiltering import gaussianFilter, sobelFilter,
LoGFilter, adaptiveLocalNoiseReductionFilter
import matplotlib.pyplot as plt
from cannyRelated import nms, doubleThresholds, otsu
import os

if __name__ == '__main__':
    inputFolder = "data"
    fileName = "p1im6.bmp"
    inputPath = os.path.join(inputFolder, fileName)
    fileNameSplit = os.path.splitext(fileName)[0]
    img = cv2.imread(inputPath)
    outputFolder = os.path.join("outputs/", fileNameSplit)
    if not os.path.exists(outputFolder):
        os.mkdir(outputFolder)
    cv2.imwrite(os.path.join(outputFolder, fileName), img)

    ##### preprocessing
    img = adaptiveLocalNoiseReductionFilter(img, 9, 100,
fileNameSplit)

    ##### edge detection
    size = 7
    mag, dir = sobelFilter(img, size)
    cv2.imwrite(os.path.join(outputFolder,
"sobel_{}_mag_ada7.bmp".format(size)), mag)

```

```

##### Canny edge detection start from magnitude
& direction maps
nmsMag = nms(mag, dir)
cv2.imwrite(os.path.join(outputFolder, "nmsMag_ada7.bmp"),
nmsMag)

# plt.hist(mag.ravel(), bins=100)
# plt.show()

thres = otsu(nmsMag)
print("otsu thres:", thres)

strong = doubleThresholds(nmsMag, thres, thres, True)
strongImg = strong * 255
cv2.imwrite(os.path.join(outputFolder,
"sobel_strong_ada7.bmp"), strongImg)

for i in [0.5]:
    iter = 1
    seSize = 5
    douThres = doubleThresholds(nmsMag, thres, thres * i,
False, iter, seSize)
    douThresImg = douThres * 255
    cv2.imwrite(os.path.join(outputFolder,
"sobel_douThres_{ }_{ }_ada7.bmp".format(iter, seSize, i)),
douThresImg)

```

p1im5.py

```

import cv2
import numpy as np
import math
from spatialFiltering import gaussianFilter, sobelFilter,
LoGFilter, adaptiveLocalNoiseReductionFilter
import matplotlib.pyplot as plt
from cannyRelated import nms, doubleThresholds, otsu
from intensityTransformation import normalizeFullDomain
import os

if __name__ == '__main__':
    inputFolder = "data"
    fileName = "p1im5.bmp"
    inputPath = os.path.join(inputFolder, fileName)
    fileNameSplit = os.path.splitext(fileName)[0]
    img = cv2.imread(inputPath)
    outputFolder = os.path.join("outputs", fileNameSplit)
    if not os.path.exists(outputFolder):
        os.mkdir(outputFolder)

```

```

cv2.imwrite(os.path.join(outputFolder, fileName), img)

##### preprocessing

##### edge detection
size = 3
mag, dir = sobelFilter(img, size)
cv2.imwrite(os.path.join(outputFolder,
"sobel_{}_mag.bmp".format(size)), mag)

##### Canny edge detection start from magnitude
& direction maps
nmsMag = nms(mag, dir)
cv2.imwrite(os.path.join(outputFolder, "nmsMag.bmp"), nmsMag)

# plt.hist(mag.ravel(), bins=100)
# plt.show()

thres = otsu(nmsMag)
print("otsu thres:", thres)

for i in [7]:
    for j in [11]:
        iter = i
        seSize = j
        douThres = doubleThresholds(nmsMag, thres, thres / 2,
False, iter, seSize, cross=True)
        douThresImg = douThres * 255
        cv2.imwrite(os.path.join(outputFolder,
"sobel_douThres_{}_{}_cross.bmp".format(seSize, iter)),
douThresImg)

```

p1im1.py

```

import cv2
import numpy as np
import math
from spatialFiltering import gaussianFilter, sobelFilter,
LoGFilter, adaptiveLocalNoiseReductionFilter
import matplotlib.pyplot as plt
from cannyRelated import nms, doubleThresholds, otsu
from intensityTransformation import normalizeFullDomain,
gammaCorrection
import os

if __name__ == '__main__':
    inputFolder = "data"
    fileName = "p1im1.bmp"
    inputPath = os.path.join(inputFolder, fileName)
    fileNameSplit = os.path.splitext(fileName)[0]
    img = cv2.imread(inputPath)

```

```

outputFolder = os.path.join("outputs", fileNameSplit + "t2")
if not os.path.exists(outputFolder):
    os.mkdir(outputFolder)
cv2.imwrite(os.path.join(outputFolder, fileName), img)

##### preprocessing
img = normalizeFullDomain(img, False)
img = gammaCorrection(img, 0.5, True)

##### edge detection
for size in [3]:
    mag, dir = sobelFilter(img, size)
    cv2.imwrite(os.path.join(outputFolder,
"sobel_{}_mag.bmp".format(size)), mag)

# ##### Canny edge detection start from
# magnitude & direction maps
nmsMag = nms(mag, dir)
cv2.imwrite(os.path.join(outputFolder, "nmsMag.bmp"), nmsMag)

# plt.hist(mag.ravel(), bins=100)
# plt.show()

thres = otsu(nmsMag)
print("otsu thres:", thres)

for dividend in [1]:
    strong = doubleThresholds(nmsMag, thres / dividend, None,
True)
    strongImg = strong * 255
    cv2.imwrite(os.path.join(outputFolder,
"sobel_strong_{}.bmp".format(dividend)), strongImg)
    for i in [0.5]:
        iter = 1
        seSize = 5
        douThres = doubleThresholds(nmsMag, thres, thres * i,
False, iter, seSize)
        douThresImg = douThres * 255
        cv2.imwrite(os.path.join(outputFolder,
"sobel_douThres_{}_{}_{}.bmp".format(iter, seSize, i)),
douThresImg)

```