谭树杰 11849060

Lab11 Report

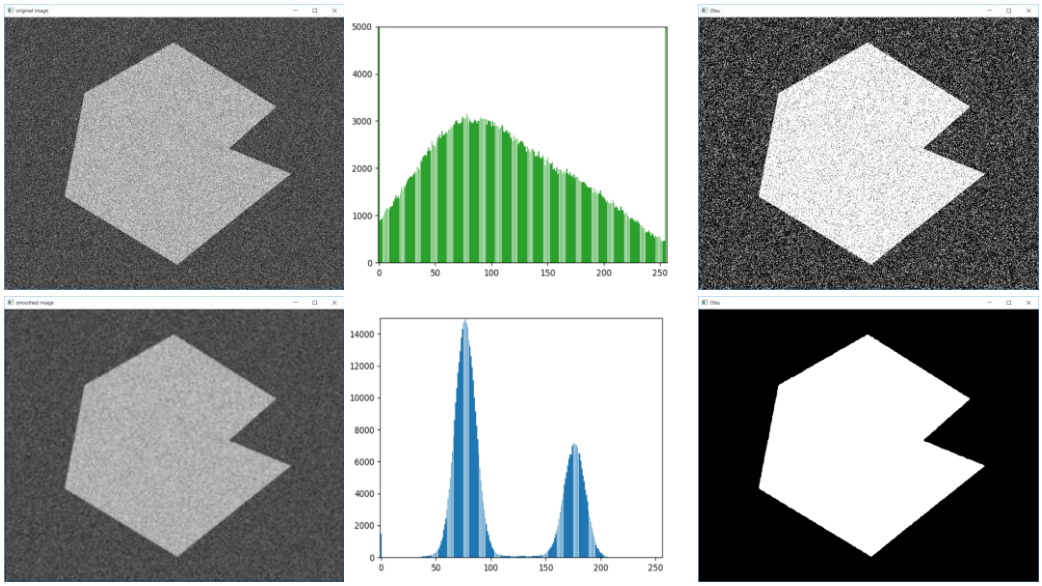
1. **Use Otus's method on original image large_septagon_gaussian_noise_mean_0_std_50_added.pgm and its 5x5 smoothed image to perform segmentation to output binary images.**

Results:

(1) The first row below are the original image and its histogram and Result obtained using Otsu's method respectively. (threshold = 108)

The second row below are the smoothed image, its histogram and Result obtained using Otsu's method respectively. (threshold = 125)

The smooth mask is 5 * 5 averaging



Analysis:

Every black point in the white region and every white point in the black region is a thresholding error, so the segmentation was highly unsuccessful.

To solve this problem, we smooth the noisy image with an averaging mask of size 5x5. The improvement in the shape of the histogram due to smoothing is evident, and we would expect thresholding of the smoothed image to be nearly perfect. As final picture shows, this indeed was the case.The slight distortion of the boundary between object and background in the segmented, smoothed image was caused by the blurring of the boundary. In fact, the more aggressively we smooth an image, the more boundary errors we should anticipate in the segmented result.

The implementation code is shown below:

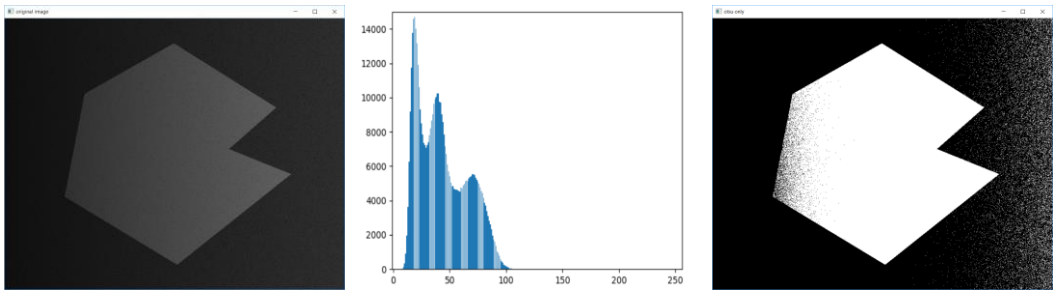
```python
import numpy as np
import cv2
import matplotlib.pyplot as plt


def hist_image(img):
    bw_data = np.array(img).astype('int32')
    bins = np.array(range(0,257))
    counts, pixels =np.histogram(bw_data, bins)
    pixels = pixels[:-1]
    plt.bar(pixels, counts, align='center')
    plt.xlim(-1, 256)
    plt.ylim(-1, 15000)
    plt.show()

    total_counts = np.sum(counts)
    counts = counts.astype(np.float64)
    counts *= 1.0 / total_counts
    return bins, counts, pixels, bw_data, total_counts



def otsu(img):
    bins, hist, pixels, bw_data, total_counts = hist_image(img)
    sumB, wB, maximum = (0, 0, 0.0)
    sum1 = np.sum(pixels * hist)  # m_g
    total = np.sum(hist)  # total = 1
    level = 10
    for i in range(1, 255):
        wB = wB + hist[i] # P1
        wF = total - wB   # P2
        if wB == 0 or wF == 0:
            continue
        sumB = sumB + i * hist[i] # m(k)
        mB = sumB / wB    # m1(k)
        mF = (sum1 - sumB) / wF # m2(k)
        between = wB * wF * (mB - mF) ** 2
        if between >= maximum:
            level = i
            maximum = between
    dst = threshold(img, level)
    return dst
```

```
def threshold(src, th):
    dst = np.zeros_like(src, dtype=float)
    dst[src <= th] = 0
    dst[src > th] = 1
    return dst


def arithmetic_mean_filter(src, size=3):
    margin = int(size / 2)
    dst = np.zeros_like(src, dtype=np.uint8)
    img_padded = np.pad(src, ((margin, margin), (margin, margin)), 'constant')
    for r in range(margin, img_padded.shape[0] - (margin+1)):
        for c in range(margin, img_padded.shape[1] - (margin + 1)):
            filter_window = np.copy(img_padded[r - margin:r + margin + 1, c - margin:c + margin + 1])
            dst[r - margin, c - margin] = np.sum(filter_window) / (size * size)
    return dst


path_in = './in/large_septagon_gaussian_noise_mean_0_std_50_added.pgm'
img = cv2.imread(path_in, 0)
# cv2.imshow("original image", img)

dst = otsu(img)
cv2.imshow("Otsu", dst)


smoothed = arithmetic_mean_filter(img, size=5)
smoothed_int = np.copy(smoothed)
# cv2.imshow("smoothed image", smoothed_int)
dst = otsu(smoothed_int)
cv2.imshow("Otsu", dst)
```

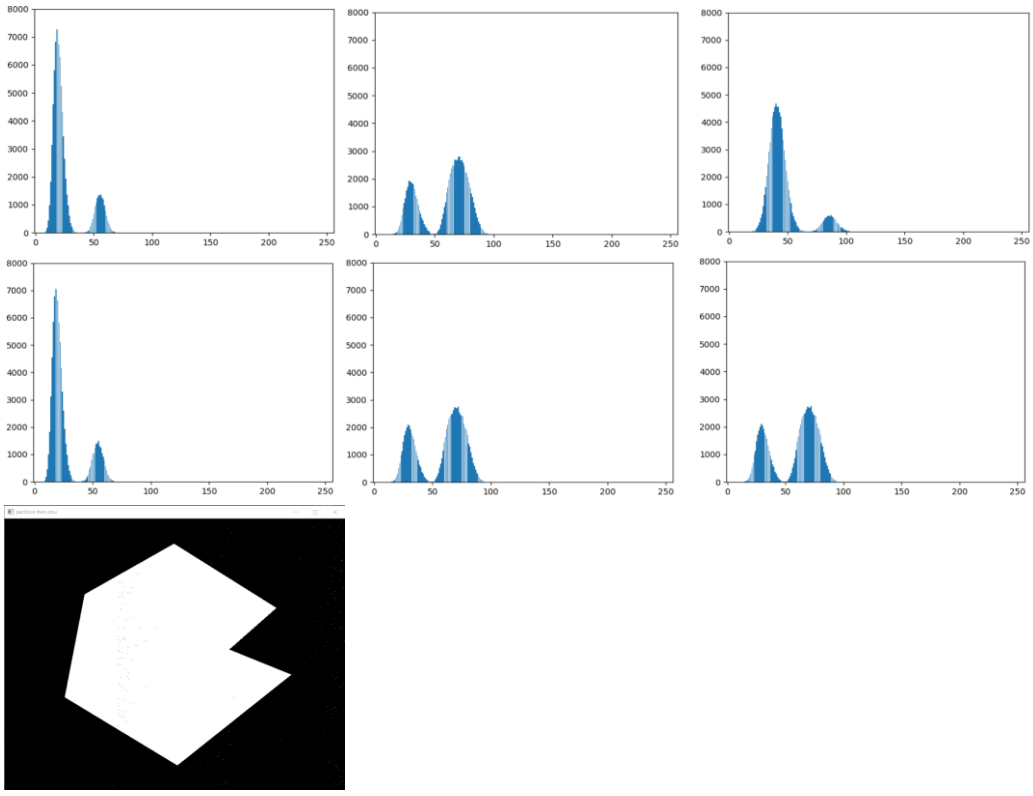
2. Partition method first and then Otsu's method to segment septagon_noisy_shaded.pgm.

Results:

(1) The septagon_noisy_shaded.pgm, its histogram and result image by directly applying Otsu's algorithm are shown below.



(2) The histogram of six parts of the original image and the result image by partitioning first then applying Otsu's algorithm are shown below.



Analysis:

We can observe that the original image has three peak values, therefore the Otsu's method can not be used directly, as we can see from the result image.

We can observe that every subimage have only one distinct valley, hence we can use Otsu's method on each subimage separately, then assemble these subimages to get the final result image. We can observe the improvement clearly.

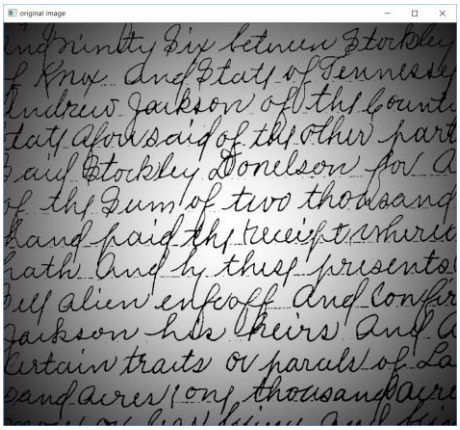
The implementation code is shown below:

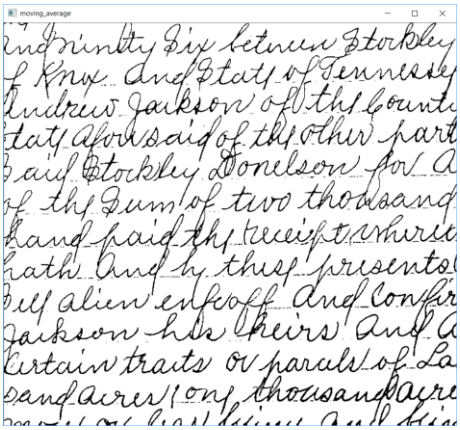
```
# 2. partition first
path_in = './in/septagon_noisy_shaded.pgm'
img = cv2.imread(path_in, 0)
cv2.imshow("original image", img)
dst = otsu(img)
cv2.imshow("otsu only", dst)
hight, width = img.shape
r1 = int(hight / 2)
c1 = int(width / 3)
c2 = c1 * 2
r2,c3 = hight, width
d11 = otsu(img[0:r1, 0:c1])
d12 = otsu(img[0:r1, c1:c2])
d13 = otsu(img[0:r1, c2:c3])
d21 = otsu(img[r1:r2, 0:c1])
d22 = otsu(img[r1:r2, c1:c2])
d23 = otsu(img[r1:r2, c2:c3])
block1 = np.concatenate([d11,d12,d13], axis=1)
block2 = np.concatenate([d21,d22,d23], axis=1)
dst = np.concatenate([block1, block2], axis=0)
cv2.imshow("partition then otsu", dst)
```

3. Use moving average thresholding to segment spot_shaded_text_image.pgm

Result:

The spot_shaded_text_image.pgm and thresholded image are shown below.





Analysis:

Because using the Otsu global thresholding method could not overcome the intensity variation. We choose local thresholding using moving averages. We can see that the handwriting characters of the image is more legible after processing. In general, thresholding

based on moving averages works well when the objects of interest are small (or thin) with respect to the image size, a condition satisfied by images of typed or handwritten text.

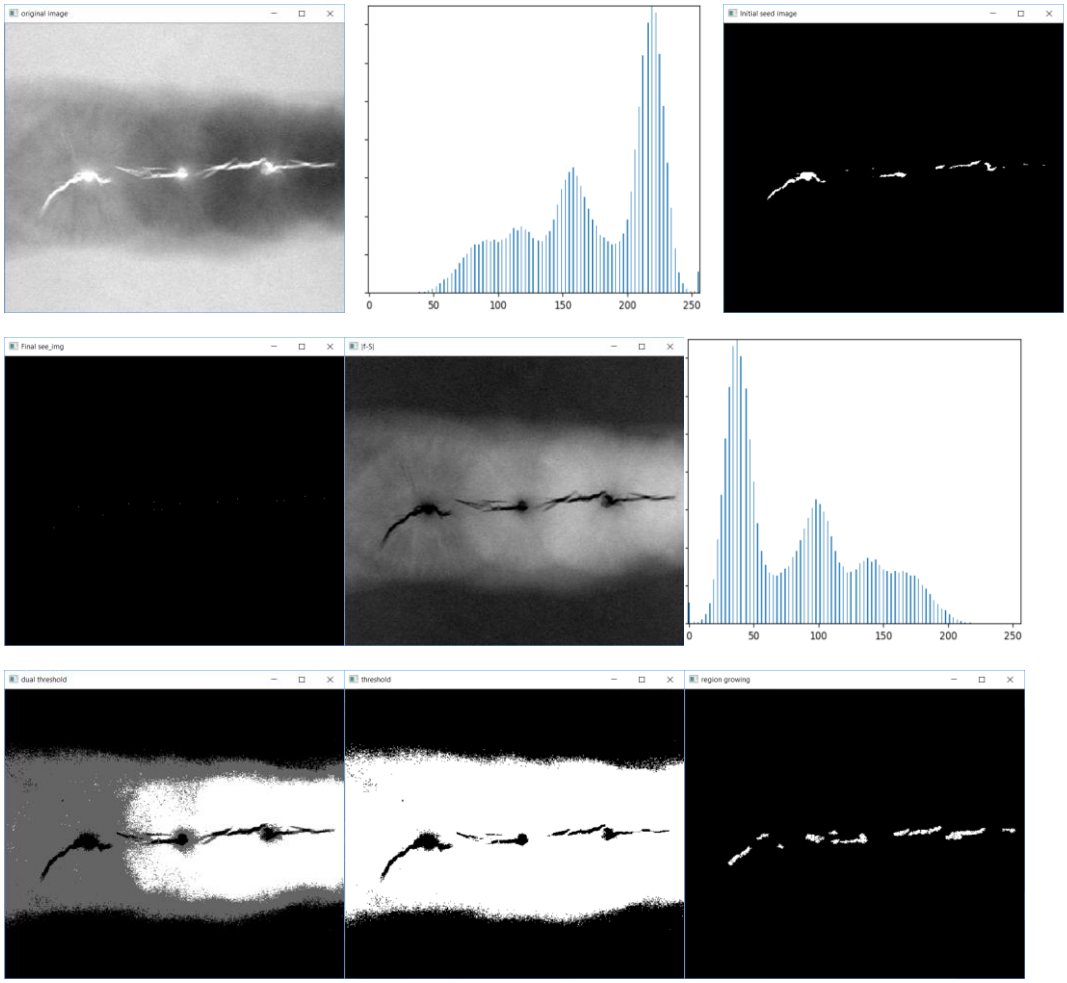
The implementation code is shown below:

```
# 3. moving average
import numpy as np
from queue import Queue
import cv2


def moving_average(img, n, b):
    hight, width = img.shape
    thres = np.zeros_like(img)
    dst = np.zeros_like(img)
    q = Queue()
    m = 0.
    for i in range(n):
        q.put(0.)
    flag = 1
    for i in range(hight):
        for j in range(0, width):
            if flag == -1:
                j = width-1 - j
            head = q.get()
            tail = float(img[i, j])
            q.put(tail)
            m = m + (tail - head) / n
            # thres[i, j] = m
            if img[i,j] > b * m:
                dst[i,j] = 255
            # print("img[i,j]= "+str(img[i,j])+"  b*m= "+str(b*m))
        flag = -flag
    return dst

path_in = './in/spot_shaded_text_image.pgm'
img = cv2.imread(path_in, 0)
cv2.imshow("original image", img)
dst = otsu(img)
cv2.imshow("Otsu", dst)

dst = moving_average(img, 20, 0.5)
cv2.imshow("moving_average", dst)
```

4. Use region growing method to perform segmentation on defective_weld.pgm and noisy_region.pgm

Result:

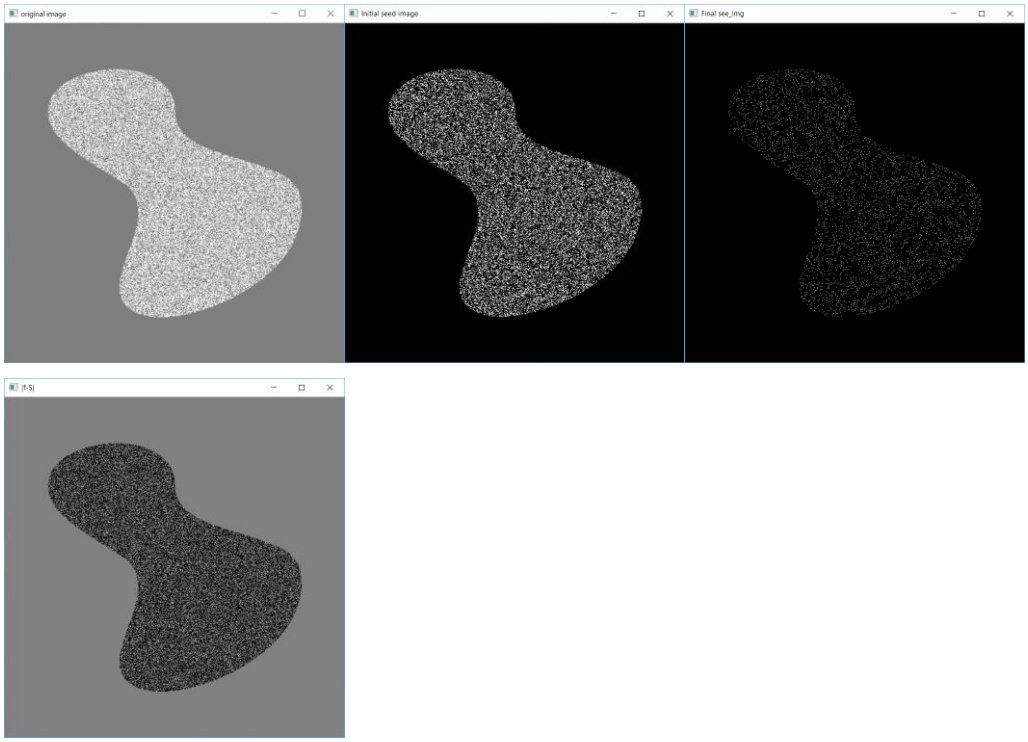
(1)The defective_weld.pgm and related images(as in textbook) are shown below.



a b c
d e f
g h i

FIGURE (a) X-ray image of a defective weld. (b) Histogram. (c) Initial seed image. (d) Final seed image (the points were enlarged for clarity). (e) Absolute value of the difference between (a) and (c). (f) Histogram of (e). (g) Difference image thresholded using dual thresholds. (h) Difference image thresholded with the smallest of the dual thresholds. (i) Segmentation result obtained by region growing. (Original image courtesy of X-TEK Systems, Ltd.)

(2) The noisy_region.pgm and related images as previous are shown below.



Analysis:

We illustrate the use of region growing by segmenting the defective weld regions. These regions could be used in applications such as weld inspection, for inclusion in a database of historical studies, or for controlling an automated welding system. The result in Fig. (g) shows that the problem of segmenting the defects cannot be solved using dual thresholds, even though the thresholds are in the main valleys. Fig(i) shows, this step resulted in the correct segmentation, indicating that the use of connectivity was a fundamental requirement in this case.

The implementation code is shown below:

```python
import numpy as np
from matplotlib import pyplot as plt
import sys
import cv2
from copy import deepcopy
from collections import Counter
from random import choice


def getset(img):
    img_set = set()
    for r in range(img.shape[0]):
        for c in range(img.shape[1]):
            if img[r, c] != 0:
                img_set.add((r, c))
    return img_set


def get_component(img_set, point):
    result = set()
    queue = {point}
    tmp_set = deepcopy(img_set)
    while queue != set():
        p = queue.pop()
        result.add(p)
        for r in range(-1, 2):
            for c in range(-1, 2):
                a_point = (p[0]+r, p[1]+c)
                if a_point in tmp_set:
                    queue.add(a_point)
                    tmp_set = tmp_set.difference({a_point})
    return result


def erode(img):
    img_set = getset(img)
    dst = np.zeros_like(img)
    while img_set != set():
        p = choice(list(img_set))
        dst[p] = 255
        com = get_component(img_set, p)
        img_set = img_set.difference(com)
    return dst
```

```
def set_to_img(src_set, img):
    test_img = np.zeros_like(img)
    for p in src_set:
        test_img[p] = 255
    return test_img


def hist_image(img):
    bw_data = np.array(img).astype('int32')
    bins = np.array(range(0,257))
    counts, pixels =np.histogram(bw_data, bins)
    pixels = pixels[:-1]
    plt.bar(pixels, counts, align='center')
    plt.xlim(-1, 256)
    plt.ylim(-1, 15000)
    plt.show()

    total_counts = np.sum(counts)
    counts = counts.astype(np.float64)
    counts *= 1.0 / total_counts
    return counts, pixels


def growing(img, seed):
    neighbors = [ ]
    for i in range(-1, 2):
        for j in range(-1, 2):
            if not (i == 0 and j == 0):
                neighbors.append((i, j))
    region_threshold = 68
    region_size = 1
    intensity_difference = 0.
    neighbor_point_list = []
    neighbor_intensity_list = []
    # Mean of the segmented region
    region_mean = img[seed]
    height, width = img.shape
    image_size = height * width
```

```
    segmented_img = np.zeros_like(img, np.uint8)
    while (intensity_difference < region_threshold) and (region_size < image_size):
        for i in range(8):
            x_new = seed[0] + neighbors[i][0]
            y_new = seed[1] + neighbors[i][1]
            check = (0 <= x_new < height) and (0 <= y_new < width)
            if check:
                if segmented_img[x_new, y_new] == 0:
                    neighbor_point_list.append([x_new, y_new])
                    neighbor_intensity_list.append(img[x_new, y_new])
                    segmented_img[x_new, y_new] = 255
            # Add pixel with intensity neareat to the mean to the region
            distance = np.abs(neighbor_intensity_list - np.array([region_mean]))
            pixel_distance = min(distance)
            index = np.where(distance == pixel_distance)[0][0]
            segmented_img[seed[0], seed[1]] = 255
            region_size += 1
            # New region mean
            region_mean = (region_mean * region_size + neighbor_intensity_list[index]) / (region_size+1)
            # Update the seed value
            seed = neighbor_point_list[index]
            # Remove the value from the neighborhood lists
            neighbor_intensity_list[index] = neighbor_intensity_list[-1]
            neighbor_point_list[index] = neighbor_point_list[-1]
    seg_set = getset(segmented_img)
    return seg_set


def region_growing(img, thres):
    thres_set = getset(thres)
    dst_set = set()
    while thres_set != set():
        seed = thres_set.pop()
        dst_set = dst_set.union(growing(img, seed))
    dst = set_to_img(dst_set, img)
    return dst


def threshold(src, th):
    dst = np.zeros_like(src)
    dst[src > th] = 255
    return dst
```

```
if __name__ == '__main__':
    # path_in = './in/defective_weld.pgm'
    path_in = './in/noisy_region.pgm'
    img = cv2.imread(path_in, 0)
    cv2.imshow("original image", img)
    # counts, pixels = hist_image(img)
    thres = threshold(img, 254)
    # cv2.imshow("Initial seed image", thres)

    seed_img = erode(thres)
    cv2.imshow("Final see_img", seed_img)

    diff = np.abs(thres - img)
    diff = diff.astype(np.uint8)
    # cv2.imshow("|f-S|", diff)
    # counts, pixels = hist_image(diff)

    dual_img = np.zeros_like(diff)
    dual_img[diff>126] = 255
    dual_img[(diff<=126) * (diff > 68)] = 100
    # cv2.imshow("dual threshold", dual_img)

    dual = threshold(diff, 68)
    # cv2.imshow("threshold", dual)

    dst = region_growing(img, seed_img)
    cv2.imshow('region growing', dst)
    cv2.waitKey()
    cv2.destroyAllWindows()
```