Lab 10

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.
2. Partition method first and then Otsu's method to segment septagon_noisy_shaded.pgm
3. Use moving average thresholding to segment spot_shaded_text_image.pgm
4. Use region growing method to perform segmentation on defective_weld.pgm and noisy_region.pgm

Algorithm:

1. Otus's

The assumption of the OTSU algorithm is that there exists a threshold, TH, which divides all pixels of the image into two classes, C1 (pixels less than TH) and C2 (pixels greater than TH). The mean values of these two classes are denoted as m1 and m2, respectively, while the global mean of the image is represented as mG. Additionally, the probabilities of pixels belonging to class C1 and class C2 are denoted as p1 and p2, respectively.

$$p1 * m1 + p2 * m2 = mG \quad (1)$$

$$p1 + p2 = 1 \quad (2)$$

The variance between classes:

$$\sigma^2 = p1(m1 - mG)^2 + p2(m2 - mG)^2 \quad (3)$$

Simplify the equation and substitute (1) into equation (3)

$$\sigma^2 = p1p2(m1 - m2)^2$$

In fact, the grayscale k that maximizes the above equation is the OTSU threshold, and many blogs do the same.

$$p1 = \sum_{i=0}^{k} p_i \quad (5)$$

$$m1 = \frac{1}{p1} * \sum_{i=0}^{k} ip_i \quad (6)$$

$$m2 = \frac{1}{p2} * \sum_{i=k+1}^{L-1} ip_i \quad (6)$$

According to the formula, go through 0~255 gray levels and find the largest k of equation (4)

2. Partition

1) Compute the histogram of the image.

2) Initialize variables:

- bestThreshold = 0
- bestVariance = 0

3) Iterate over all possible threshold values:

for threshold in range(0, 255):

Compute the probabilities of pixels in classes C1 (less than threshold) and C2 (greater than or equal to threshold):

- p1 = sum(histogram[0:threshold]) / totalPixels
- p2 = sum(histogram[threshold+1:255]) / totalPixels

Compute the mean values of pixels in classes C1 and C2:
- m1 = sum(pixelIntensity * histogram[pixelIntensity] for pixelIntensity in range(0, threshold+1)) / (totalPixels * p1)
- m2 = sum(pixelIntensity * histogram[pixelIntensity] for pixelIntensity in range(threshold+1, 256)) / (totalPixels * p2)

Compute the between-class variance:
- variance = p1 * p2 * (m1 - m2)^2

Check if the computed variance is greater than the bestVariance:
if variance > bestVariance:
- bestVariance = variance
- bestThreshold = threshold

4) Apply the bestThreshold to the image to create a binary segmentation result:
- segmentedImage = createBinaryImage(image, bestThreshold)

Return segmentedImage.

3. Moving Average Thresholding

movingAverageThresholding(image, windowSize, threshold):
Iterate over each pixel in the image:
for each pixel (x, y) in image:
Compute the average intensity within the local window centered at (x, y):
- window = extractLocalWindow(image, (x, y), windowSize)
- averageIntensity = sum(pixelIntensity for pixelIntensity in window) / windowSize^2

Check if the average intensity is greater than or equal to the threshold:
if averageIntensity >= threshold:
- resultImage[x, y] = 255 # Set the pixel to white in the result image
else:
- resultImage[x, y] = 0 # Set the pixel to black in the result image

Return resultImage.

4. Region Growing

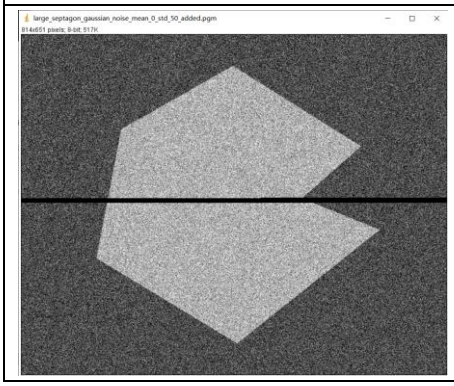
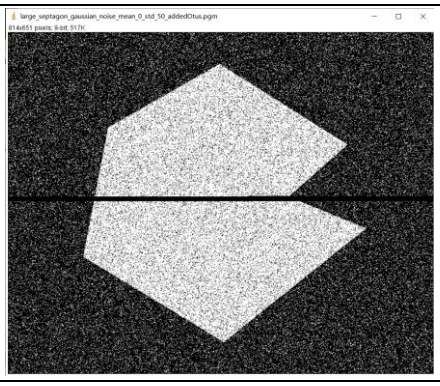
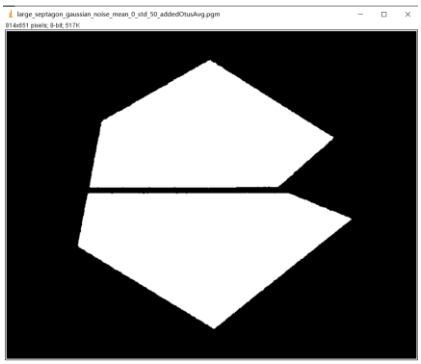
regionGrowing(image, seedPoint, threshold):
Initialize an empty queue.
Push the seedPoint onto the queue.
Get the grayscale intensity of the seedPoint:
- seedIntensity = image[seedPoint.x, seedPoint.y]
Initialize a visited matrix to keep track of visited pixels and set all elements to false.
While the queue is not empty:
Pop the front pixel from the queue:
- currentPoint = queue.pop()
Mark the currentPoint as visited:
- visited[currentPoint.x, currentPoint.y] = true
Set the currentPoint as part of the segmented region:
- resultImage[currentPoint.x, currentPoint.y] = 255 # Set the pixel to white in the result image

Iterate over the neighbors of the currentPoint:
for each neighborPoint in getNeighborPoints(currentPoint):
Check if the neighborPoint is within the image boundaries and has not been visited:
if isWithinImageBoundaries(neighborPoint) and not visited[neighborPoint.x, neighborPoint.y]:
Get the grayscale intensity of the neighborPoint:
- neighborIntensity = image[neighborPoint.x, neighborPoint.y]
Check if the absolute difference between seedIntensity and neighborIntensity is less than or equal to the threshold:
if abs(seedIntensity - neighborIntensity) <= threshold:
- queue.push(neighborPoint) # Add the neighborPoint to the queue
- visited[neighborPoint.x, neighborPoint.y] = true # Mark the neighborPoint as visited

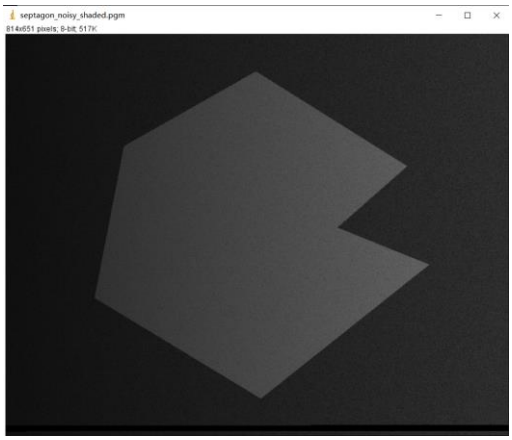
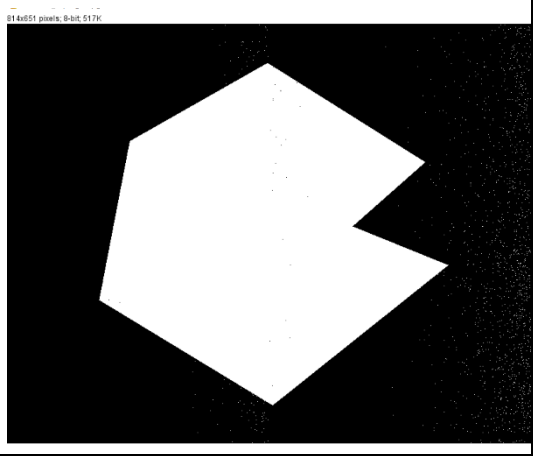
Return resultImage.

Results (compare the results with the original image):

1. Otus's

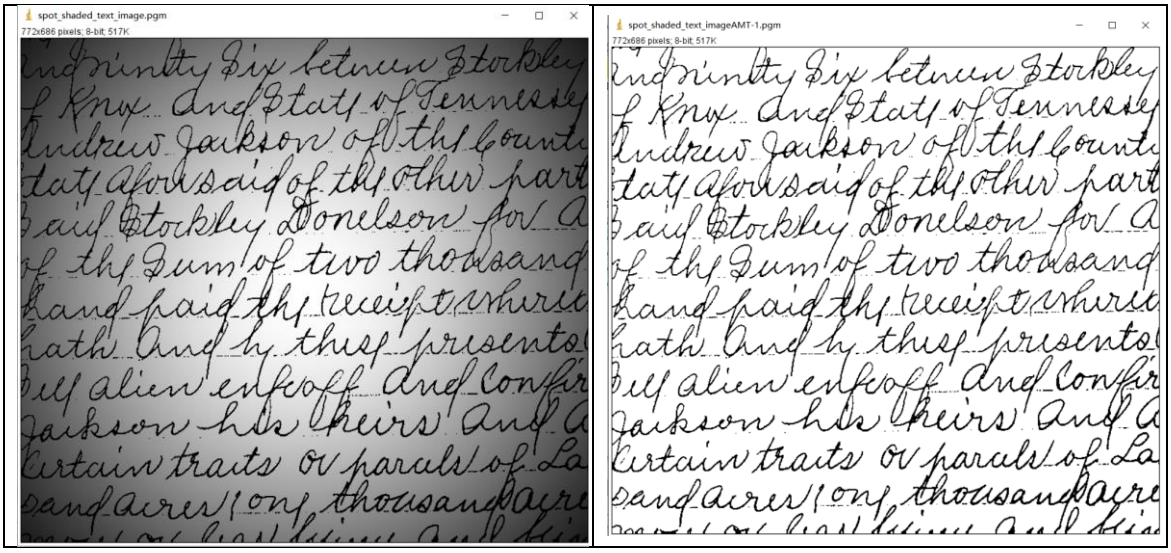
Original	Otus's	Otus's after Average Filter
		

2. Partition

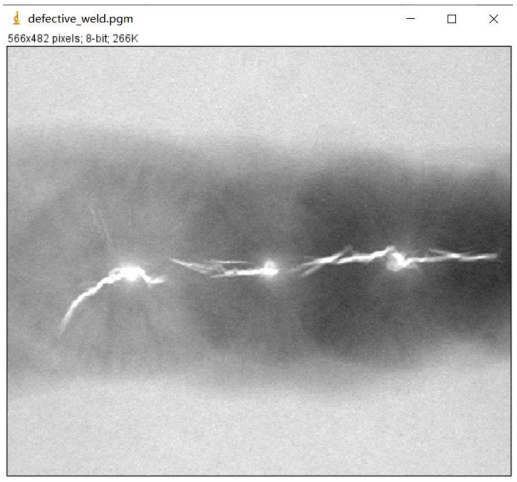
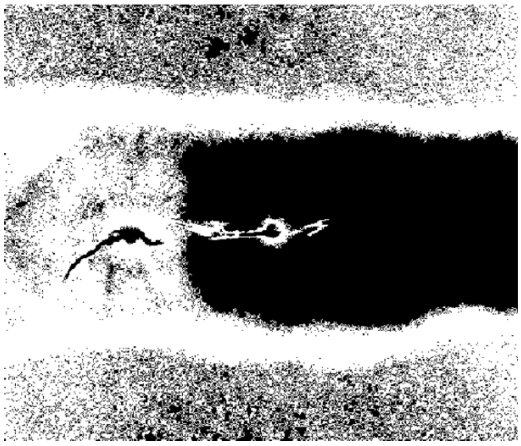
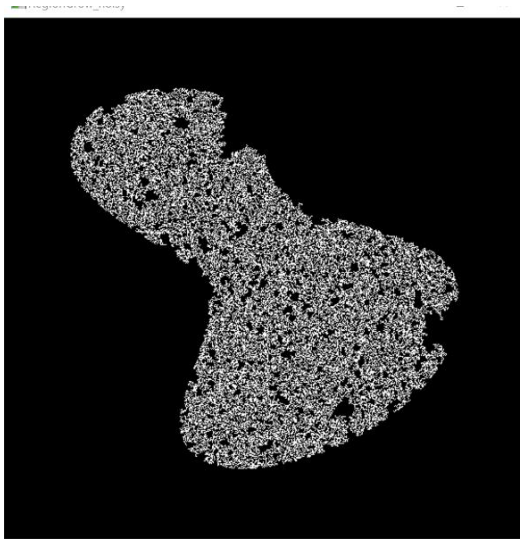
Original	Otus's after Partition
	

3. Moving average thresholding

Original	Moving average thresholding



4. Region growing method

Original	Region growing method
	
	

Discussion:

1. The significance of Otsu's binarization lies in the ability to automatically select the best threshold in order to achieve the best target and background segmentation in the image. Based on the principle of maximum inter-class variance, it calculates inter-class variance under different thresholds and finds the threshold that makes the maximum inter-class variance as the best segmentation threshold.
2. Partition method is an image segmentation method based on a threshold value. By dividing image pixels into different areas or categories, the image is divided into sub-areas with different characteristics or attributes. The partition method can help separate objects or areas of interest in an image from the background, thus providing a better basis for object detection and recognition. By setting an appropriate threshold, the target and background can be effectively separated, so that the subsequent target detection algorithm or feature extraction algorithm can analyze and process the region of interest more accurately.
3. Moving average thresholding can smooth the image and reduce the influence of noise by calculating the average value of the neighborhood around the pixel. By using the average of the neighborhood around the pixel as a threshold, isolated noise points or areas of small noise can be eliminated, resulting in a cleaner and smoother binary image.
4. The region growing method can separate the target or area of interest in the image from the background. Pixels with similar characteristics can be aggregated into a region by selecting suitable growth criteria, such as pixel similarity or distance between pixels. This helps to segment the target area and provide for subsequent target identification, analysis and measurement.

Codes:

1. Otus's

```
Findhistogram(image->Height, image->Width, image, histogram);
Findpi(image->Height, image->Width, histogram, pi);
mg = m(255, pi);

for (int i = 0; i < 256; i++) {
    var[i] = var(i, mg, pi);
}

max = var[0];
count = 1;
k = 0;
for (int i = 0; i < 256; i++) {
    if (abs(var[i], max) < 1e-10) {
        k += i;
        count++;
    }
    else if (var[i] > max) {
        max = var[i];
        count = 1;
        k = i;
    }
}
k_star = k / count;
printf("%lf\n", k_star);
for (int i = 0; i < image->Height; i++) {
    for (int j = 0; j < image->Width; j++) {
        if (tempin[i * image->Width + j] < k_star)
            tempout[i * image->Width + j] = 0;
        else
            tempout[i * image->Width + j] = 255;
    }
}
return(outimage);
```

2. Otus's after partition

```
mg = 0;
for (i = 0; i < 256; i++) {
    mg += i * p[i];
}

m1 = (int*)malloc(sizeof(int) * 256);
for (int i = 0; i < 256; i++) {
    for (int j = 0; j <= i; j++) {
        m1[i] += j * p[j];
    }
    if (p1[i] != 0) {
        m1[i] /= p1[i];
    }
    else {
        m1[i] = 0;
    }
}

m2 = (double*)malloc(sizeof(double) * 256);

for (int i = 0; i < 256; i++) {
    for (int j = i + 1; j < 256; j++) {
        m2[i] += j * p[j];
    }
    if ((areaS - p1[i]) != 0) {
        m2[i] /= (areaS - p1[i]);
    }
    else {
        m2[i] = 0;
    }
}

var = (double*)malloc(sizeof(double) * 256);
for (i = 0; i < 256; i++) {

    var[i] = p1[i] * pow(m1[i] - mg, 2) + (1 - p1[i]) * pow(m2[i] - mg, 2);
}

max = 0, min = 255;
for (i = 0; i < areaS; i++) {
    if (tempin[i] > max) {
        max = tempin[i];
    }
    if (tempin[i] < min) {
        min = tempin[i];
    }
}

int k = min + 1;
for (int i = min + 1; i < max - 1; i++) {
    if (var[i] > var[k]) {
        k = i;
    }
}
for (int m = 0; m < areaH; m++) {
    for (int n = 0; n < areaW; n++) {
        int temp = tempin[(i * areaH + m) * image->Width + j * areaW + n];
        if (temp > k) {
            tempout[(i * areaH + m) * image->Width + j * areaW + n] = 255;
        }
        else {
            tempout[(i * areaH + m) * image->Width + j * areaW + n] = 0;
        }
    }
}
```

3. Moving Average Thresholding

```
for (int i = 0; i < image->Height; i++) {
    for (int j = 0; j < image->Width; j++) {
        diff= 0.0;

        int i * image->Width + j = i * image->Width + j;
        if (i * image->Width + j < n + 1) {
            diff = tempin[i * image->Width + j];
        }
        else {
            diff = tempin[i * image->Width + j] - tempin[i * image->Width + j - n - 1];
        }

        diff *= 1 / n;
        m1 = m0 + diff;
        m0 = m1;

        if (tempin[i * image->Width + j] > round(m1 * c)) {
            tempout[i * image->Width + j] = 255;
        }
        else {
            tempout[i * image->Width + j] = 0;
        }
    }
}
```

4. Region growing method

```
for (int i = 0; i < image->Heigh; ++i) {
    for (int j = 0; j < image-Width; ++j) {
        if (pts[i * image-Width + j].pix <= 0) continue;
        if (pts[i * image-Width + j].pnum > -1) continue;
        if (seed.empty() == true) seed.push(pts[i * image-Width + j]);

        while (!seed.empty()) {
            top = seed.top();
            seed.pop();
            poly.push_back(top);
            pts[(int)(top.x * image-Width + top.y)].pnum = polys.size();

            for (int X = -1; X <= 1; ++X) {

                for (int Y = -1; Y <= 1; ++Y) {

                    if (top.x + X < 0 || top.x + X >= image->Heigh || top.y + Y < 0 || top.y + Y >= image-Width) continue;
                    if (pts[((int)top.x + X) * image-Width + ((int)top.y + Y)].pnum > -1) continue;
                    if (abs(pts[((int)top.x + X) * image-Width + ((int)top.y + Y)].pix - top.pix) <= threshold) {
                        seed.push(pts[((int)top.x + X) * image-Width + ((int)top.y + Y)]);
                        pts[((int)top.x + X) * image-Width + ((int)top.y + Y)].pnum = polys.size();
                    }
                }
            }
        }
        polys.push_back(poly);
        poly.clear();
    }
}
```