10. Otus, Moving Average, and Region Growing

1. Otus's method

Algorithm:

Here are the algorithm execution steps of Otus:

1) Compute the normalized histogram of the input image, use $p_i = \frac{n_i}{MN}, i = 1,2,3, \dots, L-1$ to represent each component of the histogram. And n_i is the number of pixels whose intensity is i.

2) Calculate the cumulative sum $p_1(k)$ for $k = 0,1,2, \dots, L-1$:

$$p_1(k) = \sum_{i=0}^{k} p_i$$

3) Calculate the cumulative mean $m(k)$ for $k = 0,1,2, \dots, L-1$:

$$m_{(k)} = \sum_{i=0}^{k} ip_i$$

4) Calculate the mean of the global intensity:

$$m_G = \sum_{i=0}^{L-1} ip_i$$

5) Calculate the interclass variance $\sigma B^2(k)$ for $k = 0,1,2, \dots, L-1$:

$$\sigma_B^2(k) = \frac{[m_G p_1(k) - m(k)]^2}{p_1(k)[1 - p_1(k)]}$$

6) Obtain the best threshold $k*$, if the maximum value is not unique, then use the average of the corresponding detected maximum values k to obtain $k*$:

$$\sigma_B^2(k^*) = \max_{0 \le k \le L-1} \sigma_B^2(k)$$

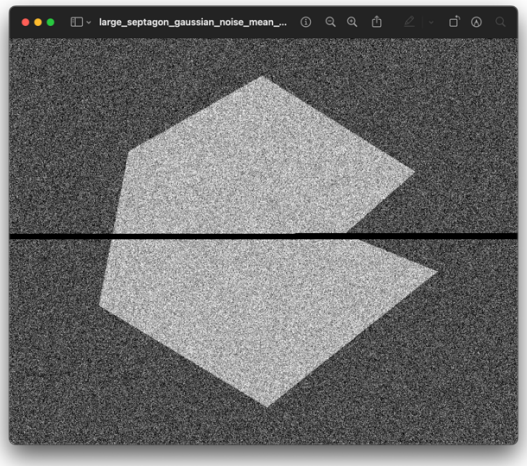
7) Binarization:

$$g(x,y) = \begin{cases} 1, f(x,y) > k^* \\ 0, f(x,y) \le k^* \end{cases}$$

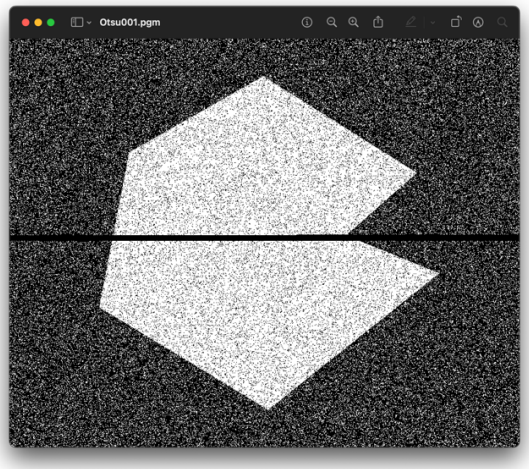
Results (including pictures):

Process result of "large_septagon_gaussian_noise_mean_0_std_50_added.pgm"

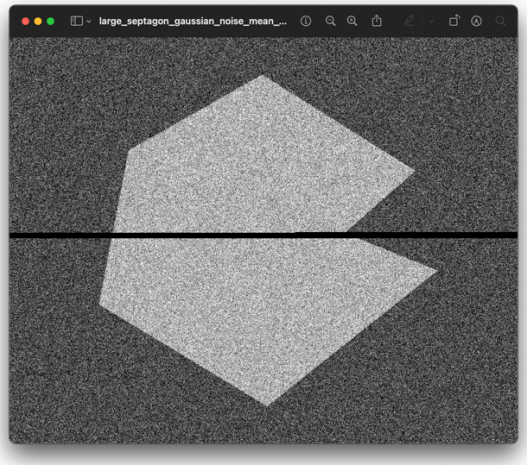
Source image: Otus's result (Without smoothing):

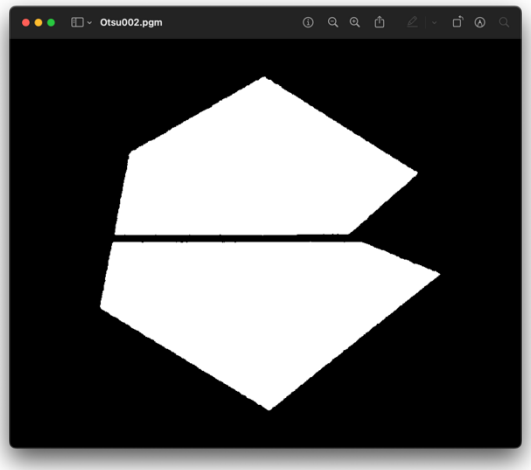


Source image:



Otus's result (With smoothing):





Discussion:

From the results, the segmentation effect of the Otus algorithm is much better than the previous global threshold algorithm. The criterion of the Otsu algorithm to measure the optimal segmentation threshold is the maximum inter-class variance, which is optimal when the inter-class variance is the largest. However, the algorithm is very sensitive to noise and target size, and noise reduction processing is required to achieve considerable results.

Codes:

(1) Otus Algorithm code:

```
258 unsigned char* Otsu_Algorithm(unsigned char *tempin, int width, int height) {
259     unsigned char *tempout;
260     int size = width * height;
261     tempout = (unsigned char *) malloc(size);
262     float histogram[256], ratio[256], sigma[256];
263 
264     for(int i = 0; i < 256; i++) {
265         histogram[i] = 0;
266     }
```

```
    for(int i = 0; i < size; i++) {
        int temp = tempin[i];
        histogram[temp] += 1;
    }
    for(int i = 0; i < 256; i++) {
        ratio[i] = histogram[i] / (float)size;
    }

    float mg = sum(256, ratio);

    for(int i = 0; i < 256; i++) {
        sigma[i] = calculate_sigma(i, mg, ratio);
    }

    float max = sigma[0];
    int max_count = 1, k = 0;
    for(int i = 1; i < 256; i++) {
        if(abs(sigma[i] - max) < 1e-10) {
            k += i;
            max_count++;
        }
        else if(sigma[i] > max) {
            max = sigma[i];
            max_count = 1;
            k = i;
        }
    }
    float k_final = (float)k / (float)max_count;

    for(int i = 0; i < size; i++) {
        if(tempin[i] <= k_final) tempout[i] = 0;
        else tempout[i] = 255;
    }
    return tempout;
```

(2) Calculate sigma function and sum function:

```
float calculate_sigma(int k, float mg, float ratio[]){
    float p1k = 0.0;
    for(int i = 0; i < k; i++) {
        p1k += ratio[i];
    }
    float mk = sum(k, ratio);
    if(p1k < 1e-10 || (1 - p1k) < 1e-10) return 0.0;
    else return pow(mg * p1k - mk, 2) / (p1k * (1 - p1k));
}

float sum(int k, float ratio[]) {
    float sum = 0.0;
    for(int i = 0; i < k; i++) {
        sum += i * ratio[i];
    }
    return sum;
}
```

(3) Otus main code:

```
void Otsu(Image *image) {
    unsigned char *tempin, *tempout, *out;
    Image *outimage;
    outimage = CreateNewImage(image, (char*)"#testing Function");
    tempin = image->data;
    out = outimage->data;
    tempout = Otsu_Algorithm(tempin, image->Width, image->Height);

    for(int i = 0; i < image->Width * image->Height; i++) {
        out[i] = tempout[i];
    }
    SavePNMImage(outimage, (char*)"Otsu.pgm");
}
```

2. Partition Otus's method

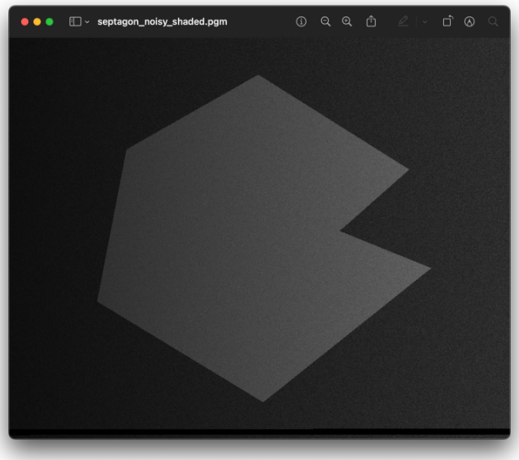
Algorithm:

Divide the image into non-overlapping rectangles, ensure that the lighting of each rectangle is approximately uniform, and then perform the Otus algorithm on each rectangle separately. The Otus algorithm is the same as in the first part of this report.

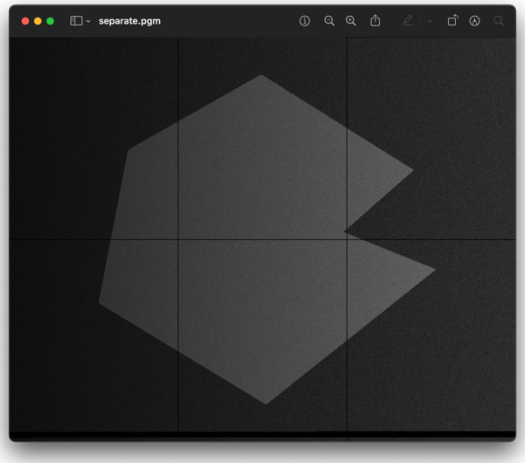
Results (including pictures):

Process result of "septagon_noisy_shaded.pgm"

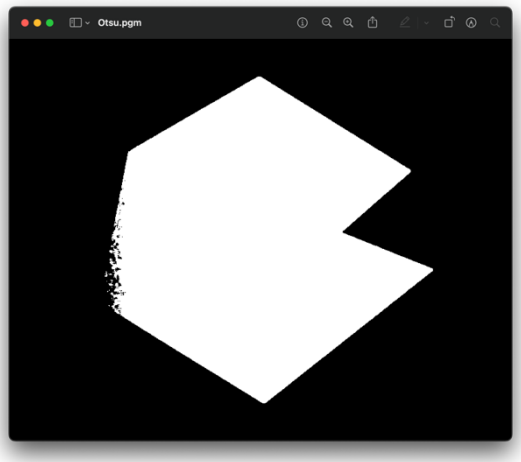
Source image:



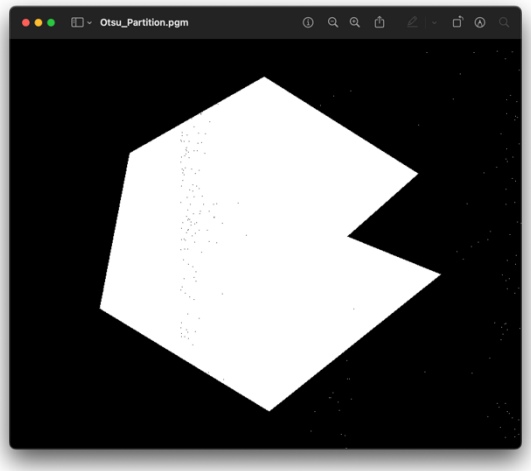
Splitted image:



Global Otus' s result:



Partition Otus' s result:



Discussion:

Partition Otus's method belongs to variable thresholding. Because noise and uniform lighting play an important role in the performance of the thresholding algorithm. For example, in our image with lighting changes this time, the segmentation error with global Otus is obvious.. But although the original image is divided into 6 parts, there are still some little error in the result, because there are two modes in the histogram and two modes have an obvious trough. Therefore, Partition segmentation works better when objects and backgrounds occupy

reasonably comparable sized areas. But if the segmented image only contains objects or backgrounds, it will fail.

Codes:

```
void Otsu_Partition(Image *image) {
    unsigned char *tempin, *tempout;
    int y = (float)image->Height / 2.0 + 1;
    int x = (float)image->Width / 3.0 + 1;
    int size_partition = x * y;
    Image *outimage;
    outimage = CreateNewImage(image, (char*)"#testing Function");
    tempin = image->data;
    tempout = outimage->data;
    unsigned char inPart1[size_partition], inPart2[size_partition], inPart3[size_partition], inPart4[size_partition],
        inPart5[size_partition], inPart6[size_partition];
    unsigned char *outPart1, *outPart2, *outPart3, *outPart4, *outPart5, *outPart6;

    for(int i = 0; i < image->Height; i++) {
        for(int j = 0; j < image->Width; j++){
            if(j - 2*x >= 0 && i - y >= 0) {
                inPart6[x * (i-y) + (j-2*x)] = tempin[image->Width * i + j];
            }
            else if(j - x < 0 && i - y < 0) {
                inPart1[x * i + j] = tempin[image->Width * i + j];
            }
            else if(j - 2*x >= 0 && i - y < 0) {
                inPart3[x * i + (j-2*x)] = tempin[image->Width * i + j];
            }
            else if(j - x < 0 && i - y >= 0) {
                inPart4[x * (i-y) + j] = tempin[image->Width * i + j];
            }
            else if(j >= x && j < 2*x && i - y >= 0) {
                inPart5[x * (i-y) + (j-x)] = tempin[image->Width * i + j];
            }
            else if(j >= x && j < 2*x && i - y < 0) {
                inPart2[x * i + (j-x)] = tempin[image->Width * i + j];
            }
        }
    }

    outPart1 = Otsu_Algorithm(inPart1, x, y);
    outPart2 = Otsu_Algorithm(inPart2, x, y);
    outPart3 = Otsu_Algorithm(inPart3, x, y);
    outPart4 = Otsu_Algorithm(inPart4, x, y);
    outPart5 = Otsu_Algorithm(inPart5, x, y);
    outPart6 = Otsu_Algorithm(inPart6, x, y);

    for(int i = 0; i < image->Height; i++) {
        for(int j = 0; j < image->Width; j++){
            if(j - 2*x >= 0 && i - y >= 0) {
                tempout[image->Width * i + j] = outPart6[x * (i-y) + (j-2*x)];
            }
            else if(j - x < 0 && i - y < 0) {
                tempout[image->Width * i + j] = outPart1[x * i + j];
            }
            else if(j - 2*x >= 0 && i - y < 0) {
                tempout[image->Width * i + j] = outPart3[x * i + (j-2*x)];
            }
            else if(j - x < 0 && i - y >= 0) {
                tempout[image->Width * i + j] = outPart4[x * (i-y) + j];
            }
            else if(j >= x && j < 2*x && i - y >= 0) {
                tempout[image->Width * i + j] = outPart5[x * (i-y) + (j-x)];
            }
            else if(j >= x && j < 2*x && i - y < 0) {
                tempout[image->Width * i + j] = outPart2[x * i + (j-x)];
            }
        }
    }
    SavePNMImage(outimage, (char*)"Otsu_Partition.pgm");
}
```

3. Moving Average Thresholding

Algorithm:

Assume there is a 5x5 image is shown below, aij represents the intensity at (i,j).

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{pmatrix}$$

And then follow the zigzag linear scan, so it needs to turn the two-dimensional matrix into a one-dimensional row matrix:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{25} & a_{24} & a_{23} & a_{22} & a_{21} & . & . & . & a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{pmatrix}$$

After that, we insert n zeros at the beginning of the array to calculate the average at each point with:

$$m(k+1) = \frac{1}{n} \sum_{i=k+2-n}^{k+1} z_i \,,$$

Where n is the number of points used to calculate the average, and any position out of index will be taken as 0. And then we obtain the segmentation threshold:

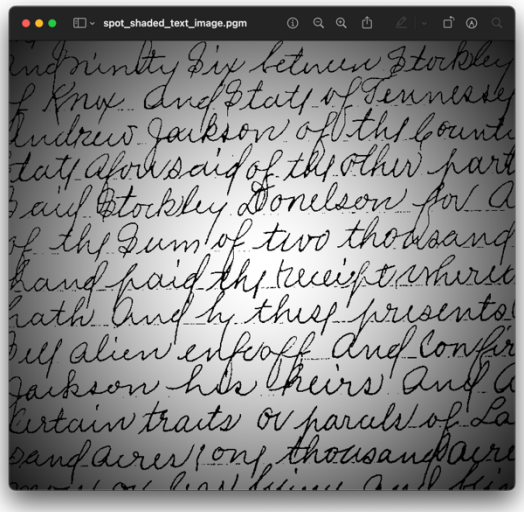
$T_{xy} = bm_{xy}, b$ is a constant.

Finally binarize the image to obtain our results.

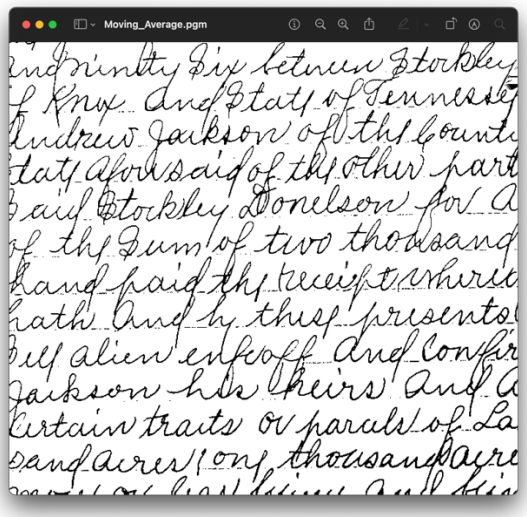
Results (including pictures):

Process result of "Moving_Average.pgm"

Source image:

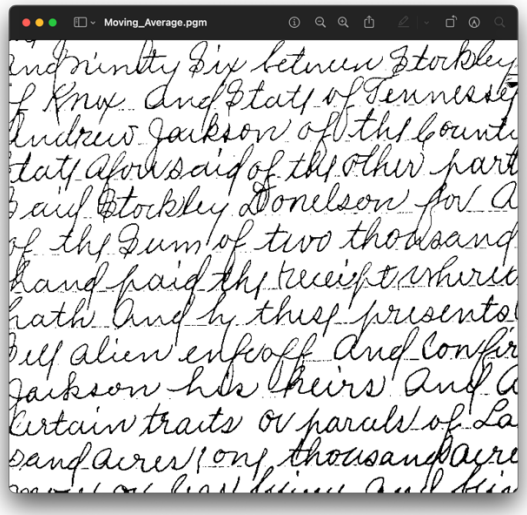


Moving average result:



Result of Otus:

Moving average result:



Discussion:

The moving average method is a kind of variable threshold processing. The moving average method scans the entire image linearly in a zigzag shape, which to reduce the deviation of the lighting in the image. This solves the problem of intensity changes that cannot be handled by the global threshold. As a rule of thumb, the algorithm works well when the size of objects and images is relatively small or thin, such as text images obscured by blobs or sinusoidal brightness.

Codes:

```
139 void movingAverage(Image *image) {
140     unsigned char *tempin, *tempout, *Z;
141     Image *outimage;
142     int size = image->Width * image->Height;
143     float T[size];
144     int T_inverse[size];
145     outimage = CreateNewImage(image, (char*)"#testing Function");
146     tempin = image->data;
147     tempout = outimage->data;
148     Z = (unsigned char *) malloc(size+20);
149 
150     for(int i = 0; i < 20; i++) {
151         Z[i] = 0;
152     }
153 
154     int index = 20;
155     for(int i = 0; i < image->Height; i++) {
156         for(int j = 0; j < image->Width; j++){
157             Z[index++] = tempin[image->Width * i + j];
158         }
159         i++;
160         for(int j = image->Width-1; j <= 0; j++){
161             Z[index++] = tempin[image->Width * i + j];
162         }
163     }
164 
165     for(int i = 0; i < size; i++) {
166         float sum = 0.0;
167         for(int j = 0; j < 20; j++) {
168             sum += Z[i-j+20];
169         }
170         T[i] = sum / 20.0 * 0.5;
171     }
172 
173     index = 0;
```

```
    for(int i = 0; i < image->Height; i++) {
        for(int j = 0; j < image->Width; j++){
            T_inverse[index++] = T[image->Width * i + j];
        }
        i++;
        for(int j = image->Width-1; j <= 0; j++){
            T_inverse[index++] = T[image->Width * i + j];
        }
    }

    for(int i = 0; i < size; i++) {
        if(tempin[i] > T_inverse[i]) tempout[i] = 255;
        else tempout[i] = 0;
    }
    SavePNMImage(outimage, (char*)"Moving_Average.pgm");
}
```

4. Region Growing

Algorithm:

The steps for a region growing implementation (my implementation) are as follows:

1) Calculate the histogram of each intensity in the image
2) Take the pixels with the highest 5% intensity as the "seed" and mark them
3) With each seed as the center, consider the 8-neighbor pixels, if they meet the growth criterion (greater than the set intensity or the difference value is small enough), then mark the neighbor valid pixels
4) Repeat step 3 until every point in the image is traversed and the growth is over.

In addition, the **Deep First Search** is implemented in my algorithms. So the time complexity of my program will be $O(image\_size)$, since each pixel in the image just needs to be traversed and marked once.

Results (including pictures):

Process result of “defective_weld.pgm” and “noisy_region.pgm”

Source image:

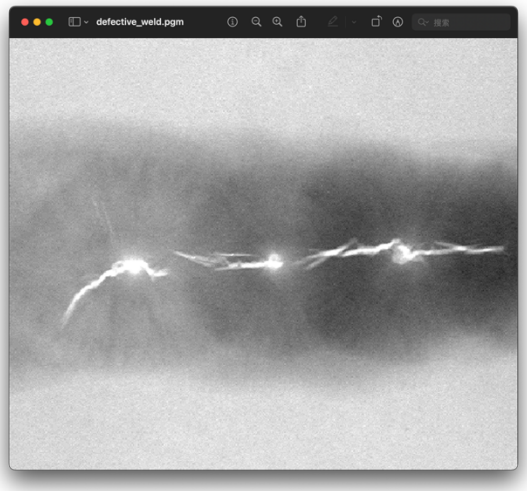
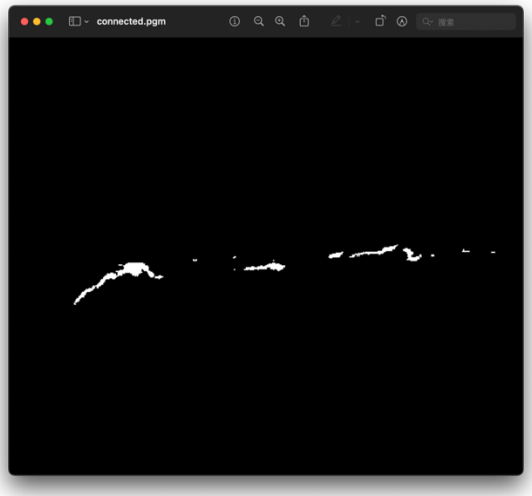
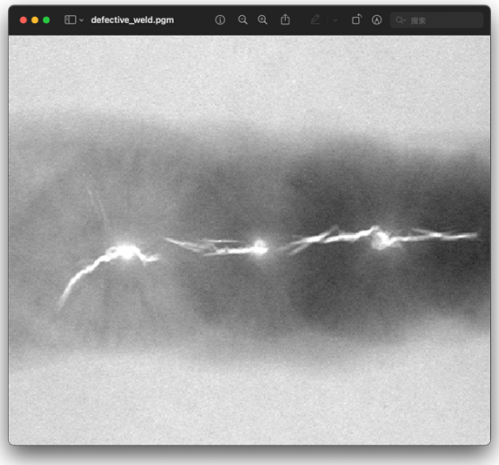


Image after thresholding:



Source image:

Region growing result:



Source image:

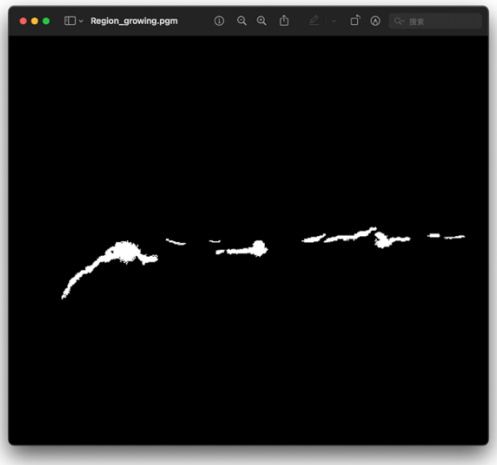
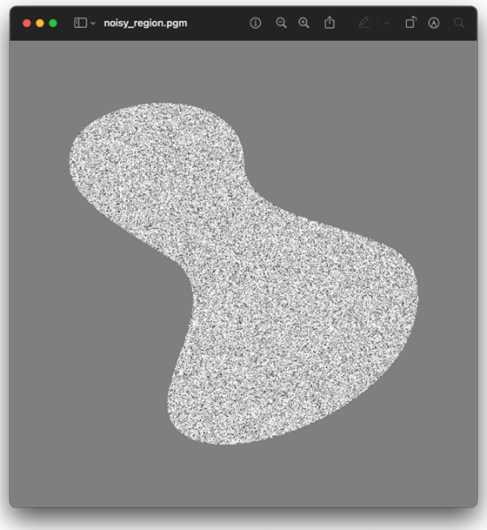
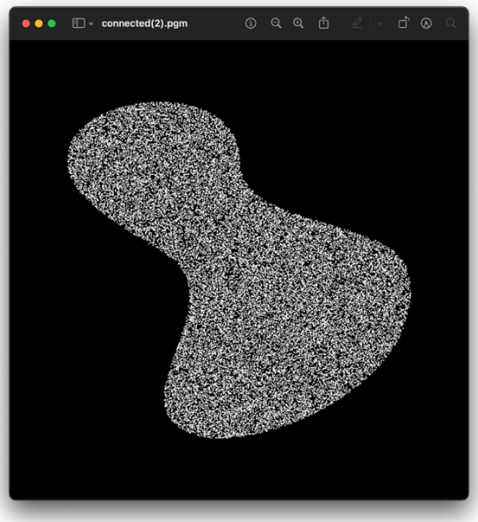


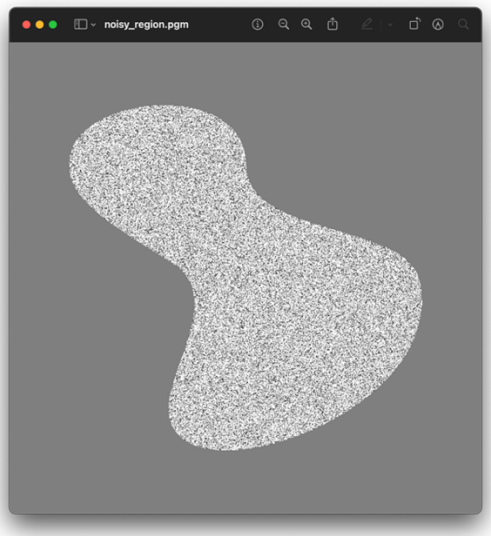
Image after thresholding:

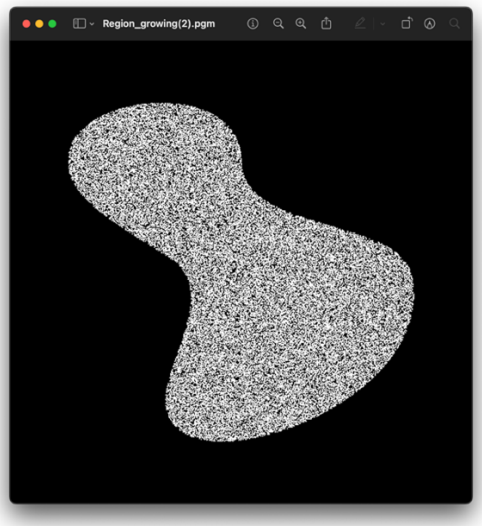


Source image:



Region growing result:





Discussion:

Region growing number is the process of aggregating pixels or subregions into larger regions according to predefined criteria. The basic idea is to start from a set of growing points, and merge adjacent pixels or regions with similar properties to the growing points.

Region growing has these advantages:

1) The connected regions with the same characteristics can be segmented.
2) It can provide good boundary information and segmentation results.
3) Growth criteria can be freely specified during the growth process, and multiple criteria can be selected at the same time.

Its disadvantages:

1) The computational cost is high **(Solved by DFS algorithm)**.
2) Noise and intensity inhomogeneity can lead to holes and over-segmentation.
3) Not good for shadows in the image.

Codes:

(1) Main part of Region Growing

```
// Algorithms Code:
void region_growing(Image *image) {
    unsigned char *tempin, *tempout1, *tempout2;
    Image *outimage1, *outimage2;
    int size = image->Width * image->Height;
    int checkBoard[size];
    float histogram[256];
    outimage1 = CreateNewImage(image, (char*)"#testing Function");
    outimage2 = CreateNewImage(image, (char*)"#testing Function");
    tempin = image->data;
    tempout1 = outimage1->data;
    tempout2 = outimage2->data;

    for(int i = 0; i < 256; i++) {
        histogram[i] = 0;
    }
    for(int i = 0; i < size; i++) {
        int temp = tempin[i];
        histogram[temp] += 1;
    }

    int T = 256;
    int light_count = 0;
    while(light_count < (float)size * 0.004) {
        T--;
        light_count += histogram[T];
    }
    for(int i = 0; i < size; i++) {
        if(tempin[i] >= T) tempout1[i] = 255;
        else tempout1[i] = 0;
    }
    SavePNMImage(outimage1, (char*)"connected.pgm");

    for(int i = 0; i < size; i++) {
        if(tempout1[i] == 255) checkBoard[i] = 1;
        else checkBoard[i] = 2;
    }
    for(int i = 0; i < image->Height; i++) {
        for(int j = 0; j < image->Width; j++){
            DFS_MarkPixels(image, checkBoard, j, i);
        }
    }
```

```
    for(int i = 0; i < size; i++) {
        if(checkBoard[i] == 3) tempout2[i] = 255;
        else tempout2[i] = 0;
    }
    SavePNMImage(outimage2, (char*)"Region_growing.pgm");
}
```

(2) Mark Pixels through Deep First Search

```
void DFS_MarkPixels(Image *image, int *checkBoard, int x, int y) {
    unsigned char *tempin;
    tempin = image->data;
    int currPosition = image->Width * y + x;
    // Base Case:
    if(x < 0 || y < 0 || x >= image->Width || y >= image->Height || checkBoard[currPosition] != 1) {
        return;
    }
    // Recursive Steps:
    checkBoard[currPosition] = 3;
    // use 8-adjacent marking:
    for(int m = -1; m <= 1; m++) {
        for(int n = -1; n <= 1; n++) {
            if(checkBoard[image->Width * (y+n) + x+m] != 2) continue;
            if(tempin[image->Width * (y+n) + x+m] > 200) {
                checkBoard[image->Width * (y+n) + x+m] = 1;
                DFS_MarkPixels(image, checkBoard, x+m, y+n);
            }
            else checkBoard[image->Width * (y+n) + x+m] = 0;
        }
    }
}
```