**Lab 10**

Use two images for each operation to do the following operations and write down their advantages and disadvantages and explain your results:

1. **Otsu with 5×5 filter**

**(large\_septagon\_gaussian\_noise\_mean\_0\_std\_50\_added.pgm):**

**Algorithm:**

1. Calculate the normalized histogram of the input image. Use pi, i = 0,1,2,..., L-1 to represent each component of the histogram.

2. Use formula (10.3-4) to calculate the cumulative sum P1(k) for k = 0,1,2,..., L-1.



3. Use formula (10.3-8) to calculate the cumulative mean m (k) for k = 0,1,2,..., L-1.



4. Calculate the global gray mean MC with equation (10.3-9)



5. Use formula (10.3-17) to calculate the inter class variance (k) for k = 0,1,2,..., L-1.



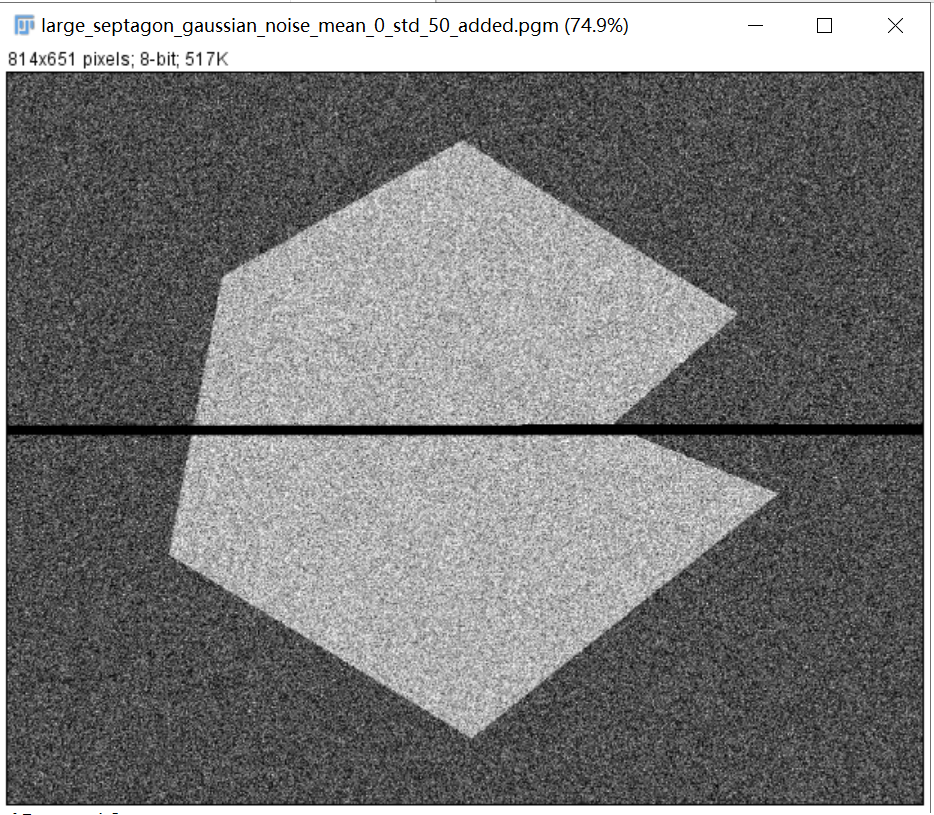
6. Get the Otsu threshold value K\*, that is to get the K value with the largest (k). If the maximum value is not unique, K\* is obtained by averaging the corresponding detected maximum values K.

7. Calculate formula (10.3-16) at k = k\* to obtain separability measure n\*.

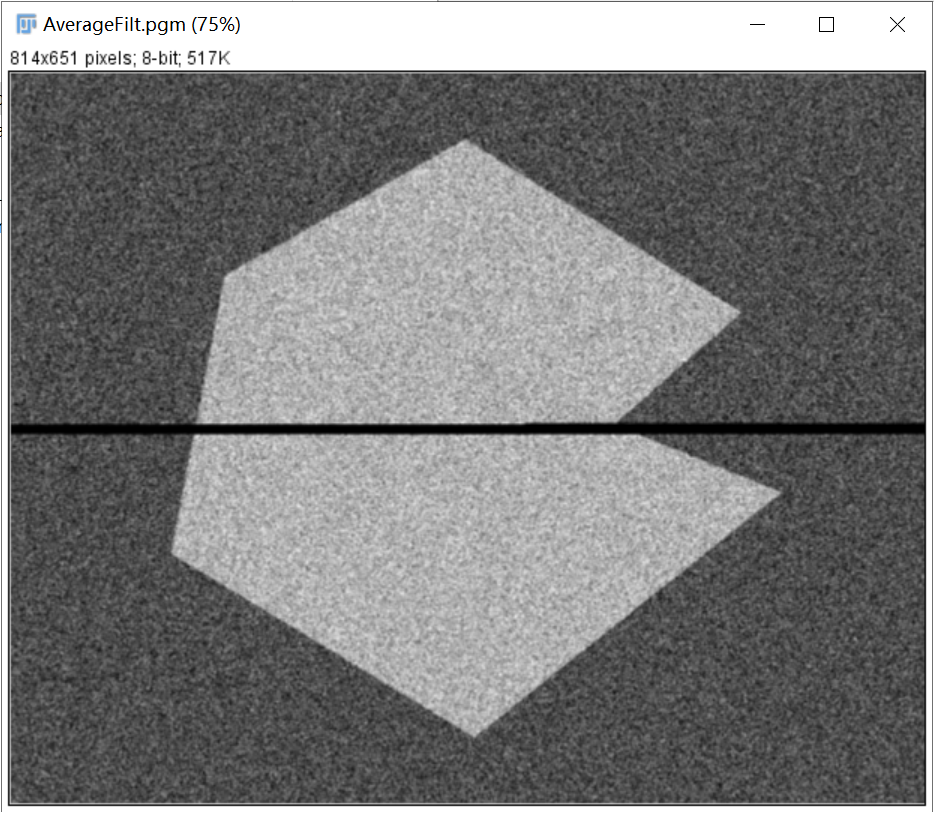


**Results (including pictures):**

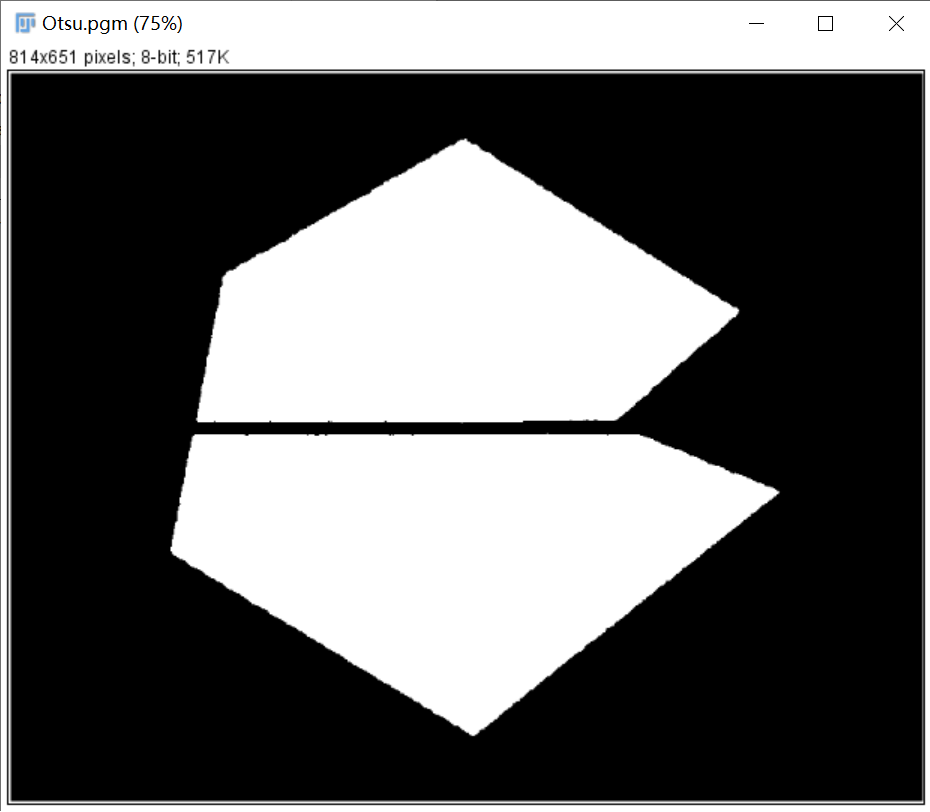
Source:



Result（5×5 average filter）:



Result(otsu after filter):



**Discussion:**

Size 5 before passing × In the mean template smoothing noise processing of 5, each black spot in the white area and each white spot in the black area are the error of threshold processing, so the segmentation is very unsuccessful. Use a size of 5 × After smoothing the noise image with the mean template of 5, the histogram shape is obviously improved due to the smoothing process. Therefore, we expect that the result obtained by threshold processing on the smoothed image will be close to perfect. In the smoothed and segmented image, the boundary between the object and the background is slightly distorted, which is caused by the blur of the boundary. In fact, we should expect that the more we smooth an image, the greater the boundary error in the segmented result.

**Codes:**

Image \*AverageFilt(Image \*image){

unsigned char \*tempin, \*tempout;

int size, i, j, k, t;

Image \*outimage;

outimage = CreateNewImage(image, "#testing AverageFilt");

tempin = image->data;

tempout = outimage->data;

if (image->Type == GRAY) size = image->Width \* image->Height;

else if (image->Type == COLOR) size = image->Width \* image->Height \* 3;

for (i = 2; i < image->Height - 2; ++i){

for (j = 2; j < image->Width - 2; ++j){

int sum = 0;

for (k = -2; k < 3; ++k)

for (t = -2; t < 3; ++t)

sum += tempin[image->Width \* (i + k) + (j + t)];

tempout[image->Width \* i + j] = sum / 25;

}

}

return(outimage);

}

Image \*Otsu(Image\* image) {

int width, height;

width = image->Width;

height = image->Height;

Image\* outimage;

unsigned char\* tempin, \*tempout;

const char\* comment = "Otsu";

outimage = CreateNewImage(image, comment);

tempin = image->data;

tempout = outimage->data;

unsigned long histogram[GRAY\_LEVELS] = {};

double pi[GRAY\_LEVELS] = {};

double sigma[GRAY\_LEVELS] = {};

double mg;

short max\_count = 0;

short k = 0;

double max\_value = 0.0;

double k\_star;

calculate\_histogram(height, width, tempin, histogram);

calculate\_pi(height, width, histogram, pi);

mg = m(GRAY\_LEVELS - 1, pi);

for (int i = 0; i < GRAY\_LEVELS; i++)

sigma[i] = calculate\_sigma(i, mg, pi);

max\_value = sigma[0];

max\_count = 1;

k = 0;

for (int i = 1; i < GRAY\_LEVELS; i++){

if (dabs(sigma[i], max\_value) < 1e-10){

k += i;

max\_count++;

}

else if (sigma[i] > max\_value){

max\_value = sigma[i];

max\_count = 1;

k = i;

}

}

k\_star = k / max\_count;

for (int i = 0; i < height; i++){

for (int j = 0; j < width; j++){

if (tempin[i \* width + j] <= k\_star)

tempout[i \* width + j] = 0x00;

else

tempout[i \* width + j] = 0xff;

}

}

return (outimage);

}

double dabs(double a, double b)

{

if (a < b)

return b - a;

else

return a - b;

}

void calculate\_histogram(long height, long width, unsigned char \*image, unsigned long histogram[])

{

short k;

for (int i = 0; i < height; i++){

for (int j = 0; j < width; j++){

k = image[i \* width + j];

histogram[k] = histogram[k] + 1;

}

}

}

void calculate\_pi(long height, long width, unsigned long histogram[], double pi[])

{

for (int i = 0; i < GRAY\_LEVELS; ++i){

pi[i] = (double)histogram[i] / (double)(height \* width);

}

}

double p1(int k, double pi[])

{

double sum = 0.0;

for (int i = 0; i <= k; i++){

sum += pi[i];

}

return sum;

}

double m(int k, double pi[])

{

double sum = 0.0;

for (int i = 0; i <= k; i++){

sum += i \* pi[i];

}

return sum;

}

double calculate\_sigma(int k, double mg, double pi[])

{

double p1k = p1(k, pi);

double mk = m(k, pi);

if (p1k < 1e-10 || (1 - p1k) < 1e-10)

return 0.0;

else

return pow(mg \* p1k - mk, 2) / (p1k \* (1 - p1k));

}

1. **Partition method then Otsu(septagon\_noisy\_shaded.pgm):**

**Algorithm:**

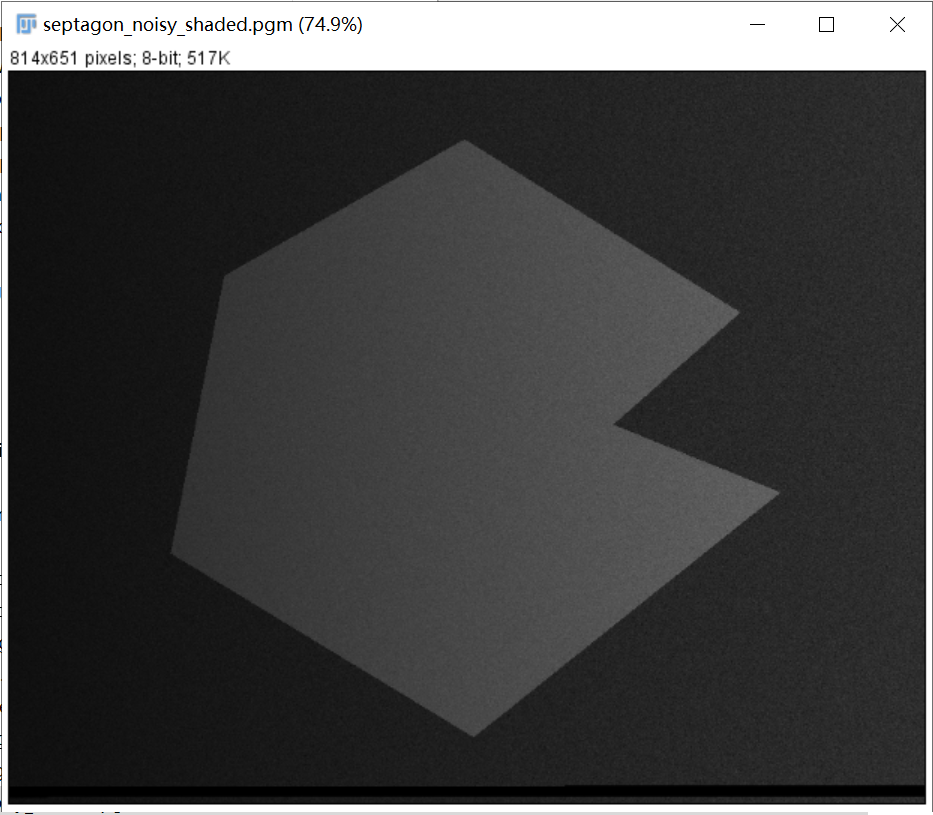
Subdivide an image into small nonoverlapping rectangles and then use Otsu on each rectangle.

The algorithm of Otsu is illustrated above.

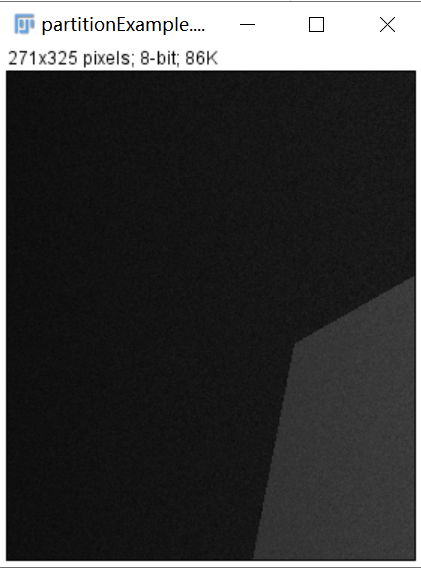
In terms of how to segment, I just simply divide the picture into six pieces and save it as a new picture. Because it can't be divided equally, there will be a difference of 1 in width and height between the pictures, but it doesn't matter

**Results (including pictures):**

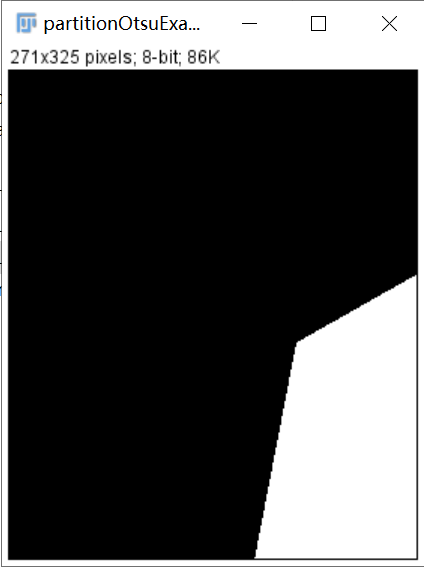
Source:



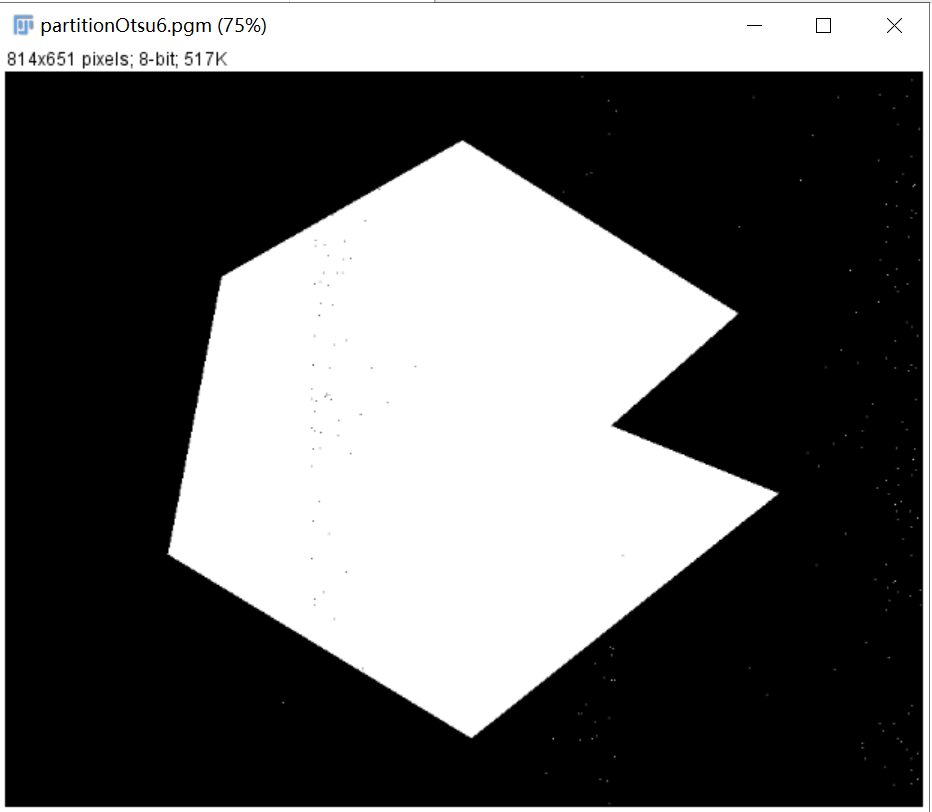
Result(one of the rectangles after subdividing):



Result(use Otsu on one of the rectangles after subdividing):



Result(After recombining):



**Discussion:**

Subdivide an image into small nonoverlapping rectangles so that the illumination of each rectangle is uniform

The method compensates the non-uniformities in illumination and/or reflectance

**Codes:**

Image \*partitionOtsu6(Image\* image) {

int width, height;

width = image->Width;

height = image->Height;

Image\* outimage, \*tempimage1, \*tempimage2, \*tempimage3, \*tempimage4, \*tempimage5, \*tempimage6;

unsigned char\* tempin, \*tempout, \*temp1, \*temp2, \*temp3, \*temp4, \*temp5, \*temp6;

const char\* comment = "partitionOtsu";

outimage = CreateNewImage(image, comment);

tempimage1 = CreateNewSizeImage(image, comment, 271, 325);

tempimage2 = CreateNewSizeImage(image, comment, 271, 325);

tempimage3 = CreateNewSizeImage(image, comment, 272, 325);

tempimage4 = CreateNewSizeImage(image, comment, 271, 326);

tempimage5 = CreateNewSizeImage(image, comment, 271, 326);

tempimage6 = CreateNewSizeImage(image, comment, 272, 326);

tempin = image->data;

tempout = outimage->data;

temp1 = tempimage1->data;

temp2 = tempimage2->data;

temp3 = tempimage3->data;

temp4 = tempimage4->data;

temp5 = tempimage5->data;

temp6 = tempimage6->data;

for (int y = 0; y < 325; y++){

for (int x = 0; x < 271; x++){

temp1[y\*271 + x] = tempin[y\*width + x];

}

}

for (int y = 0; y < 325; y++){

for (int x = 271; x < 542; x++){

temp2[y\*271 + x-271] = tempin[y \* width + x];

}

}

for (int y = 0; y < 325; y++){

for (int x = 542; x < 814; x++){

temp3[y\*272 + x-542] = tempin[y \* width + x];

}

}

for (int y = 325; y < 651; y++){

for (int x = 0; x < 271; x++){

temp4[(y - 325) \* 271 + x] = tempin[y \* width + x];

}

}

for (int y = 325; y < 651; y++){

for (int x = 271; x < 542; x++){

temp5[(y - 325) \* 271 + x-271] = tempin[y \* width + x];

}

}

for (int y = 325; y < 651; y++){

for (int x = 542; x < 814; x++){

temp6[(y - 325) \* 272 + x-542] = tempin[y \* width + x];

}

}

SavePNMImage(tempimage1, "..\\output\\partitionExample.pgm");

tempimage1 = Otsu(tempimage1);

tempimage2 = Otsu(tempimage2);

tempimage3 = Otsu(tempimage3);

tempimage4 = Otsu(tempimage4);

tempimage5 = Otsu(tempimage5);

tempimage6 = Otsu(tempimage6);

SavePNMImage(tempimage1, "..\\output\\partitionOtsuExample.pgm");

temp1 = tempimage1->data;

temp2 = tempimage2->data;

temp3 = tempimage3->data;

temp4 = tempimage4->data;

temp5 = tempimage5->data;

temp6 = tempimage6->data;

for (int y = 0; y < 325; y++){

for (int x = 0; x < 271; x++){

tempout[y\*width + x] = temp1[y \* 271 + x];

}

}

for (int y = 0; y < 325; y++){

for (int x = 271; x < 542; x++){

tempout[y\*width + x] = temp2[y \* 271 + x - 271];

}

}

for (int y = 0; y < 325; y++){

for (int x = 542; x < 814; x++){

tempout[y\*width + x] = temp3[y \* 272 + x - 542];

}

}

for (int y = 325; y < 651; y++){

for (int x = 0; x < 271; x++){

tempout[y\*width + x] = temp4[(y - 325) \* 271 + x];

}

}

for (int y = 325; y < 651; y++){

for (int x = 271; x < 542; x++){

tempout[y\*width + x] = temp5[(y - 325) \* 271 + x - 271];

}

}

for (int y = 325; y < 651; y++){

for (int x = 542; x < 814; x++){

tempout[y\*width + x] = temp6[(y - 325) \* 272 + x - 542];

}

}

return(outimage);

}

Image \*Otsu(Image\* image) {

int width, height;

width = image->Width;

height = image->Height;

Image\* outimage;

unsigned char\* tempin, \*tempout;

const char\* comment = "Otsu";

outimage = CreateNewImage(image, comment);

tempin = image->data;

tempout = outimage->data;

unsigned long histogram[GRAY\_LEVELS] = {};

double pi[GRAY\_LEVELS] = {};

double sigma[GRAY\_LEVELS] = {};

double mg;

short max\_count = 0;

short k = 0;

double max\_value = 0.0;

double k\_star;

calculate\_histogram(height, width, tempin, histogram);

calculate\_pi(height, width, histogram, pi);

mg = m(GRAY\_LEVELS - 1, pi);

for (int i = 0; i < GRAY\_LEVELS; i++)

sigma[i] = calculate\_sigma(i, mg, pi);

max\_value = sigma[0];

max\_count = 1;

k = 0;

for (int i = 1; i < GRAY\_LEVELS; i++){

if (dabs(sigma[i], max\_value) < 1e-10){

k += i;

max\_count++;

}

else if (sigma[i] > max\_value){

max\_value = sigma[i];

max\_count = 1;

k = i;

}

}

k\_star = k / max\_count;

for (int i = 0; i < height; i++){

for (int j = 0; j < width; j++){

if (tempin[i \* width + j] <= k\_star)

tempout[i \* width + j] = 0x00;

else

tempout[i \* width + j] = 0xff;

}

}

return (outimage);

}

double dabs(double a, double b)

{

if (a < b)

return b - a;

else

return a - b;

}

void calculate\_histogram(long height, long width, unsigned char \*image, unsigned long histogram[])

{

short k;

for (int i = 0; i < height; i++){

for (int j = 0; j < width; j++){

k = image[i \* width + j];

histogram[k] = histogram[k] + 1;

}

}

}

void calculate\_pi(long height, long width, unsigned long histogram[], double pi[])

{

for (int i = 0; i < GRAY\_LEVELS; ++i){

pi[i] = (double)histogram[i] / (double)(height \* width);

}

}

double p1(int k, double pi[])

{

double sum = 0.0;

for (int i = 0; i <= k; i++){

sum += pi[i];

}

return sum;

}

double m(int k, double pi[])

{

double sum = 0.0;

for (int i = 0; i <= k; i++){

sum += i \* pi[i];

}

return sum;

}

double calculate\_sigma(int k, double mg, double pi[])

{

double p1k = p1(k, pi);

double mk = m(k, pi);

if (p1k < 1e-10 || (1 - p1k) < 1e-10)

return 0.0;

else

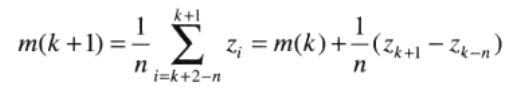
return pow(mg \* p1k - mk, 2) / (p1k \* (1 - p1k));

}

1. **moving average thresholding(spot\_shaded\_text\_image.pgm):**

**Algorithm:**

Let Zk + 1 represent the gray level of the points encountered in the scanning sequence in step K + 1. The moving average (average gray scale) at this new point is given by the following formula:

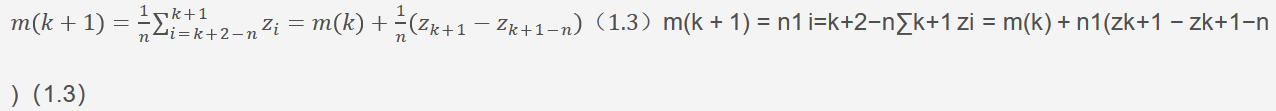
(1.1)

Where m (k) is the pixel value of the k-th point of the input image and n is the number of points used to calculate the average, and m (1) = z1 / n.

According to (1.1), there is the following formula:



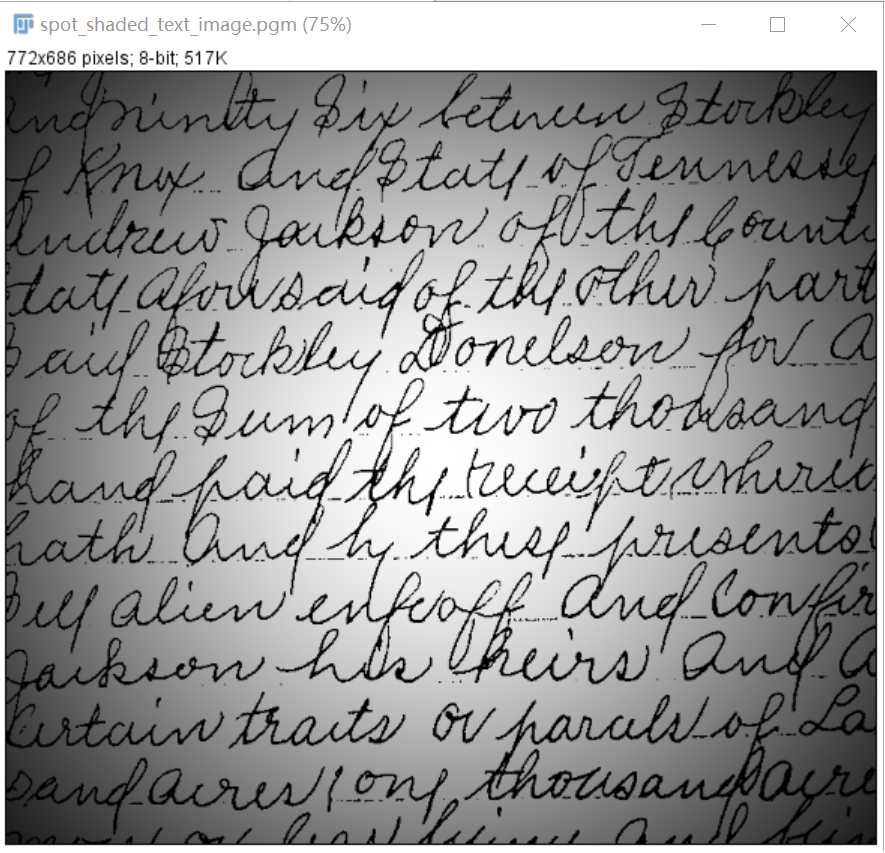
According to (1.1) and (1.2), there is the following formula:



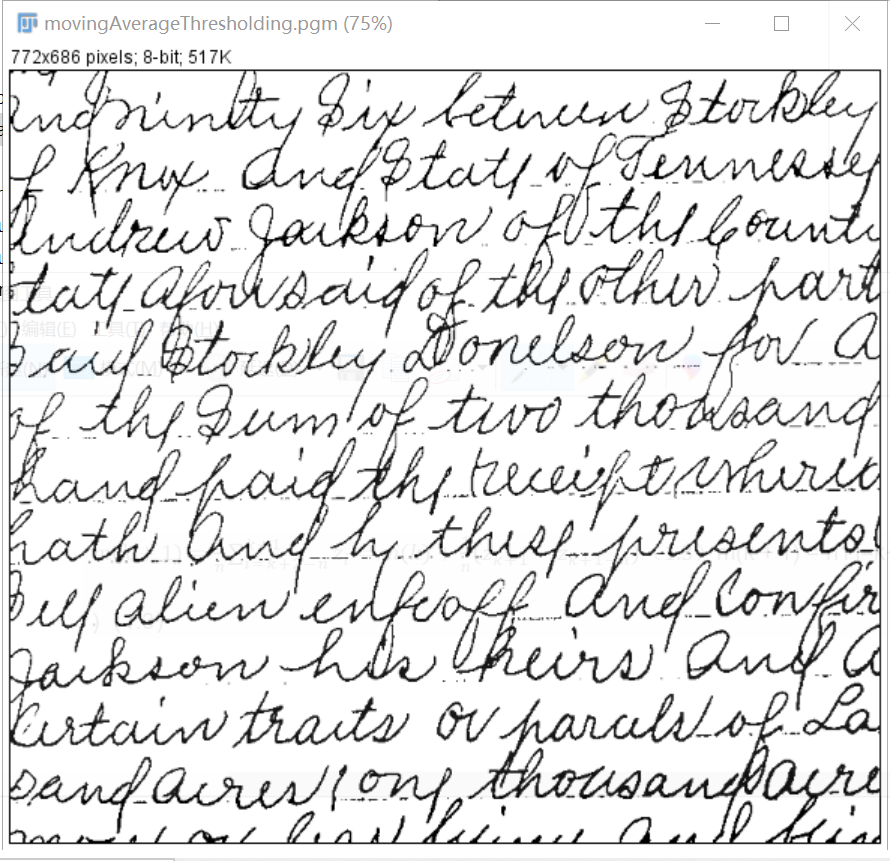
Next, binarization is performed according to the threshold of each point

**Results (including pictures):**

Source:



result:



**Discussion:**

We only apply equation (1.3) when the number of scanned points is greater than n, and in other caseszk+1−n is regarded as 0, which is equivalent to filling n-1 zeros in the boundary of the image.

**Codes:**

Image \*movingAverageThresholding(Image\* image) {

int width, height;

width = image->Width;

height = image->Height;

Image\* outimage;

unsigned char\* tempin, \*tempout;

const char\* comment = "movingAverageThresholding";

outimage = CreateNewImage(image, comment);

tempin = image->data;

tempout = outimage->data;

unsigned char \*temp = (unsigned char\*)malloc(width\*height);

for (int y = 0; y < height; y++){

for (int x = 0; x < width; x++){

if (y % 2 == 0){

temp[y\*width + x] = tempin[y\*width + x];

}

else{

temp[y\*width + x] = tempin[y\*width + width - 1 - x];

}

}

}

float m\_now = 0, m\_pre = 0, dif = 0;

int n = 20;

float b = 0.75;

for (int y = 0; y<height; y++)

for (int x = 0; x<width; x++){

if (y\*width + x + 1<n)

dif = temp[y\*width + x];

else

dif = temp[y\*width + x] - temp[y\*width + x + 1 - n];

dif \*= 1 / n;

m\_now = m\_pre + dif;

m\_pre = m\_now;

if (tempin[y\*width + x]>b\*m\_now)

tempout[y\*width + x] = 255;

else

tempout[y\*width + x] = 0;

}return(outimage);

}

1. **region growing(defective\_weld.pgm and noisy\_region.pgm):**

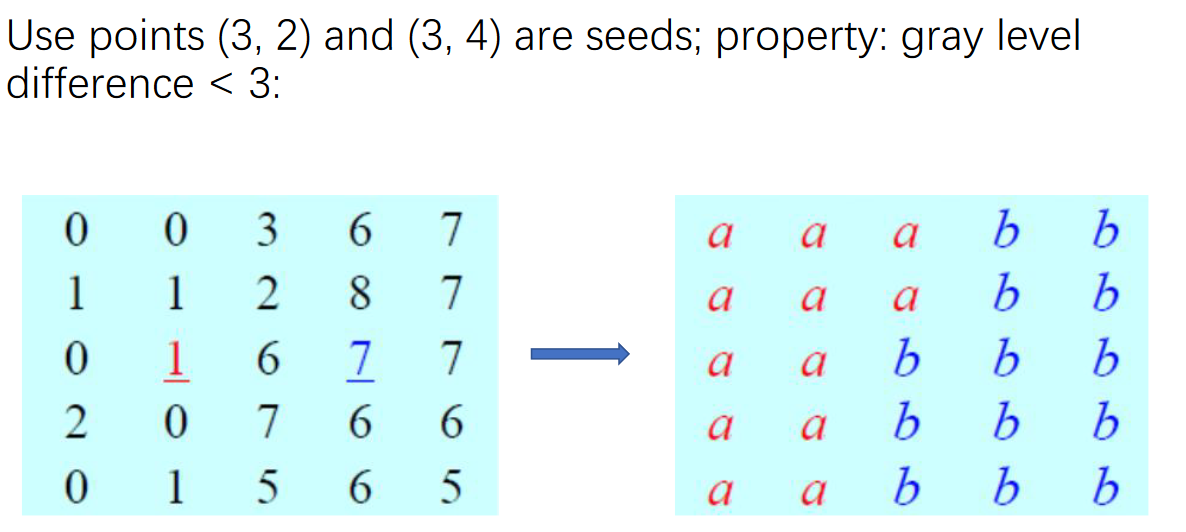
**Algorithm:**

Find all the connected components in the seed array, corrode each connected component into one pixel, mark the successfully corroded pixel as 1, and mark the other pixels as 0

Form an image in the coordinates. If the input image meets the given attribute at the coordinates of the point, it will be ordered; otherwise, it will be ordered.

Add all points with 1 of 8 connected seed points in s to each seed point in s until the growth end condition is met.

Finally, each connected component is marked in different regions to form the final segmented image.

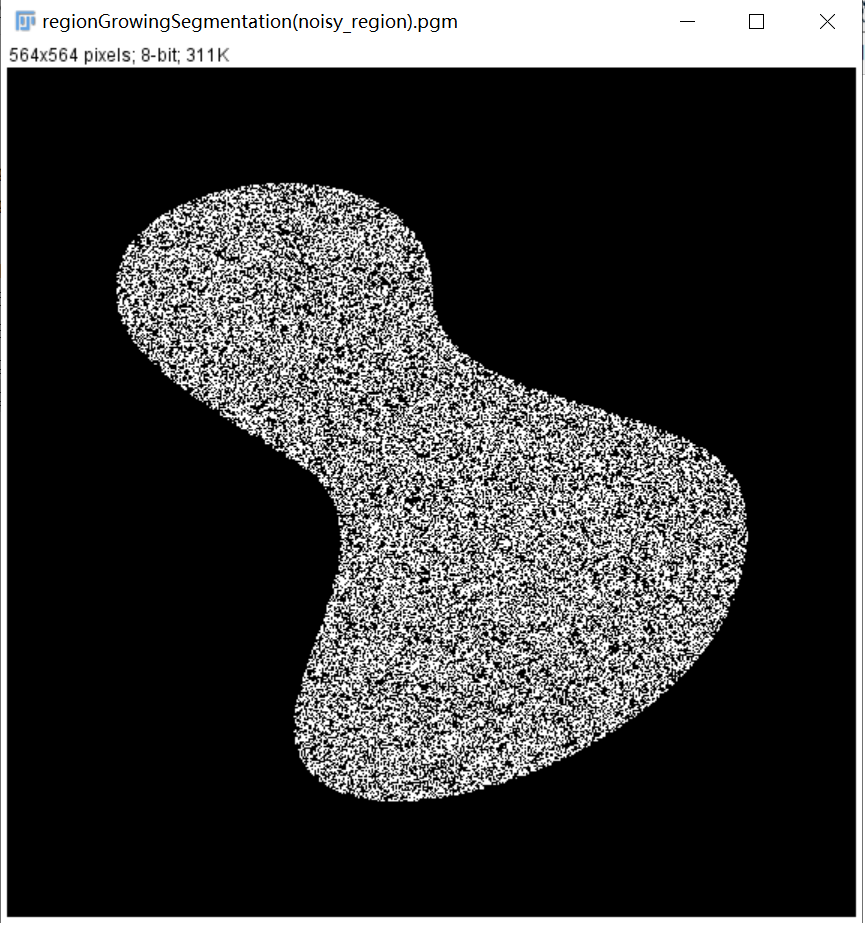


**Results (including pictures):**

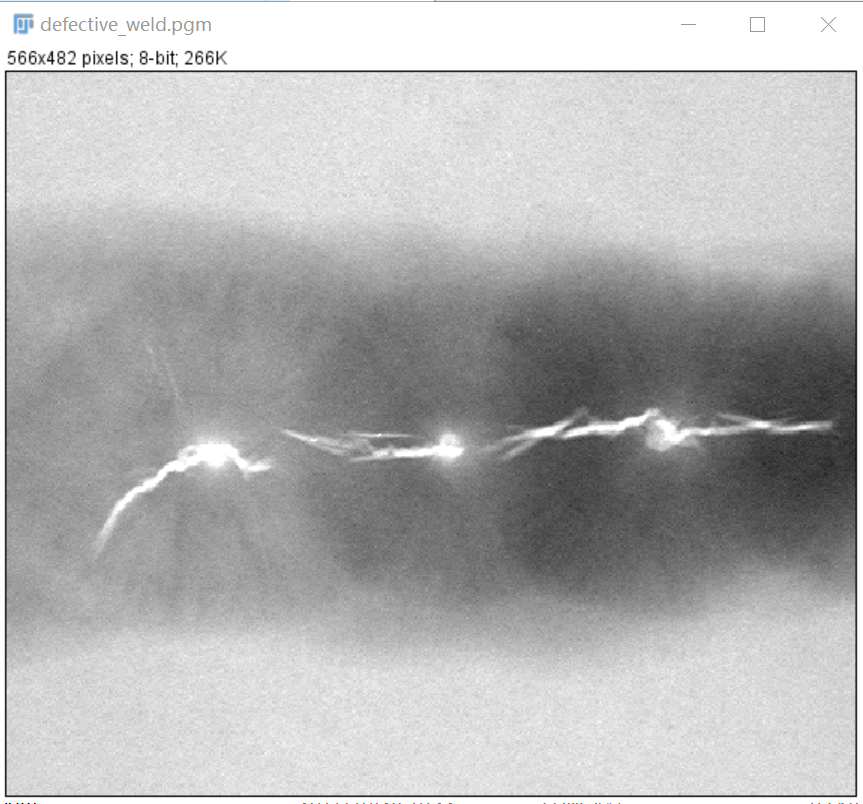
Source:



Result:



Source:



Result:



**Discussion:**

I cannot find the appropriate pixel threshold value for the noisy\_region.pgm, so the effect is not as good as defective\_weld.pgm since the threshold value of the latter has been given in the lecture.

**Codes:**

Image \*regionGrowingSegmentation(Image\* image) {

int width, height;

width = image->Width;

height = image->Height;

Image\* outimage;

unsigned char\* tempin, \*tempout;

const char\* comment = "regionGrowingSegmentation";

outimage = CreateNewImage(image, comment);

tempin = image->data;

tempout = outimage->data;

struct \_complex\* src = (struct \_complex\*)malloc(sizeof(struct \_complex) \* (width \* height));

struct \_complex\* dst = (struct \_complex\*)malloc(sizeof(struct \_complex) \* (width \* height));

int x, y;

for (x = 0; x < height; x++) {

for (y = 0; y < width; y++) {

src[x \* width + y].x = image->data[x \* width + y];

src[x \* width + y].y = 0.0;

}

}

struct \_complex\* S = (struct \_complex\*)malloc(sizeof(struct \_complex) \* (width \* height));

struct \_complex\* Q = (struct \_complex\*)malloc(sizeof(struct \_complex) \* (width \* height));

struct \_complex\* Q1 = (struct \_complex\*)malloc(sizeof(struct \_complex) \* (width \* height));

for (x = 0; x < height; x++) {

for (y = 0; y < width; y++) {

if (src[x \* width + y].x > 254) {

S[x \* width + y].x = 255;

S[x \* width + y].y = 1;

}

else {

S[x \* width + y].x = 0;

S[x \* width + y].y = 0;

}

}

}

for (x = 0; x < height; x++) {

for (y = 0; y < width; y++) {

Q[x \* width + y].x = fabs(src[x \* width + y].x - S[x \* width + y].x);

Q1[x \* width + y].x = 255 - Q[x \* width + y].x;

}

}//转化

for (x = 0; x < height; x++) {

for (y = 0; y < width; y++) {

if (Q1[x \* width + y].x <= 68) {

Q1[x \* width + y].x = 255;

}

else {

Q1[x \* width + y].x = 0;

}

Q1[x \* width + y].y = 0;

}

}

for (x = 0; x < height; x++) {

for (y = 0; y < width; y++) {

if (S[x \* width + y].y == 1) {

Q1[x \* width + y].y = 1;

Q1[x \* width + y].x = 255;

}

}

}

for (x = 0; x < height; x++) {

for (y = 0; y < width; y++) {

if (Q1[x \* width + y].y == 1) {

Grow(Q1, height, width, x, y);

}

}

}

for (x = 0; x < height; x++) {

for (y = 0; y < width; y++) {

if (Q1[x \* width + y].y == 1) {

dst[x \* width + y].x = 255;

}

else {

dst[x \* width + y].x = 0;

}

}

}

for (x = 0; x < height; x++) {

for (y = 0; y < width; y++) {

tempout[x\*width + y] = dst[x \* width + y].x;

}

}return(outimage);

}

void Grow(struct \_complex\* in, int Height, int Width, int oldx, int oldy) {

if (oldx >= 1) {

if (in[(oldx - 1) \* Width + oldy].x == 255) {

if (in[(oldx - 1) \* Width + oldy].y != 1) {

in[(oldx - 1) \* Width + oldy].y = 1;

Grow(in, Height, Width, oldx - 1, oldy);

}

}

}

if (oldx < Height - 1) {

if (in[(oldx + 1) \* Width + oldy].x == 255) {

if (in[(oldx + 1) \* Width + oldy].y != 1) {

in[(oldx + 1) \* Width + oldy].y = 1;

Grow(in, Height, Width, oldx + 1, oldy);

}

}

}

if (oldy >= 1) {

if (in[oldx \* Width + oldy - 1].x == 255) {

if (in[oldx \* Width + oldy - 1].y != 1) {

in[oldx \* Width + oldy - 1].y = 1;

Grow(in, Height, Width, oldx, oldy - 1);

}

}

}

if (oldy < Width - 1) {

if (in[oldx \* Width + oldy + 1].x == 255) {

if (in[oldx \* Width + oldy + 1].y != 1) {

in[oldx \* Width + oldy + 1].y = 1;

Grow(in, Height, Width, oldx, oldy + 1);

}

}

}

}