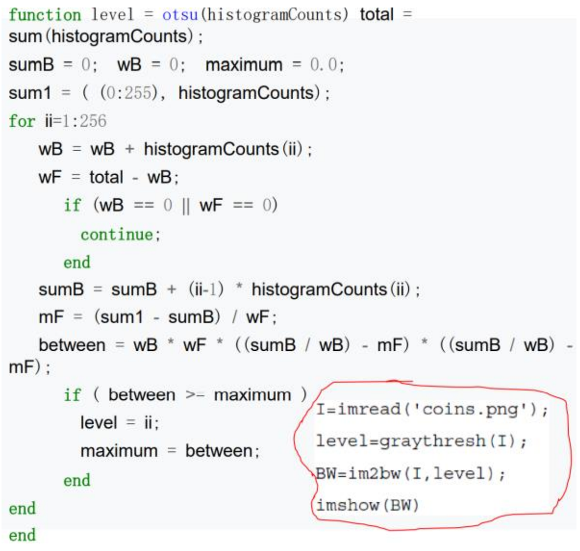
Class Lab 10

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

1. Otus's method (large_septagon_gaussian_noise_mean_0_std_50_added.pgm):

Algorithm:



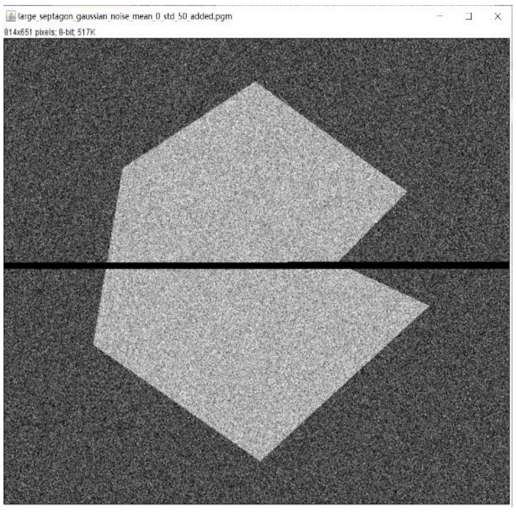
```
function level = otsu(histogramCounts) total =
sum(histogramCounts);
sumB = 0;  wB = 0;  maximum = 0.0;
sum1 = ( (0:255), histogramCounts);
for ii=1:256
    wB = wB + histogramCounts(ii);
    wF = total - wB;
      if (wB == 0 || wF == 0)
        continue;
      end
    sumB = sumB + (ii-1) * histogramCounts(ii);
    mF = (sum1 - sumB) / wF;
    between = wB * wF * ((sumB / wB) - mF) * ((sumB / wB) -
mF);
      if ( between >= maximum )
        level = ii;
        maximum = between;
      end
end
end
```

```
I=imread('coins.png');
level=graythresh(I);
BW=im2bw(I,level);
imshow(BW)
```

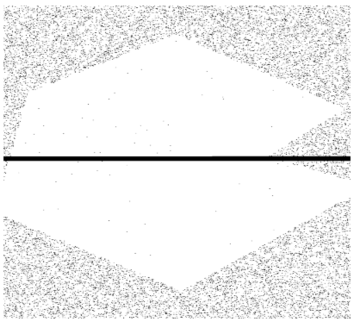
Results (including pictures):

Result of processing "large_septagon_gaussian_noise_mean_0_std_50_added.pgm":

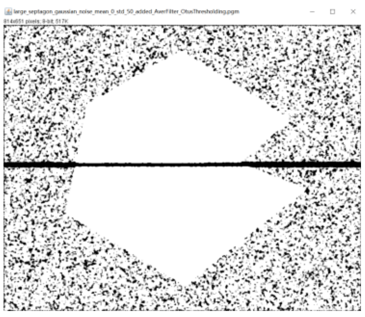
Source Image:


large_septagon_gaussian_noise_mean_0_std_50_added.pgm
814x651 pixels; 8-bit; 517K

Result after Otus's:



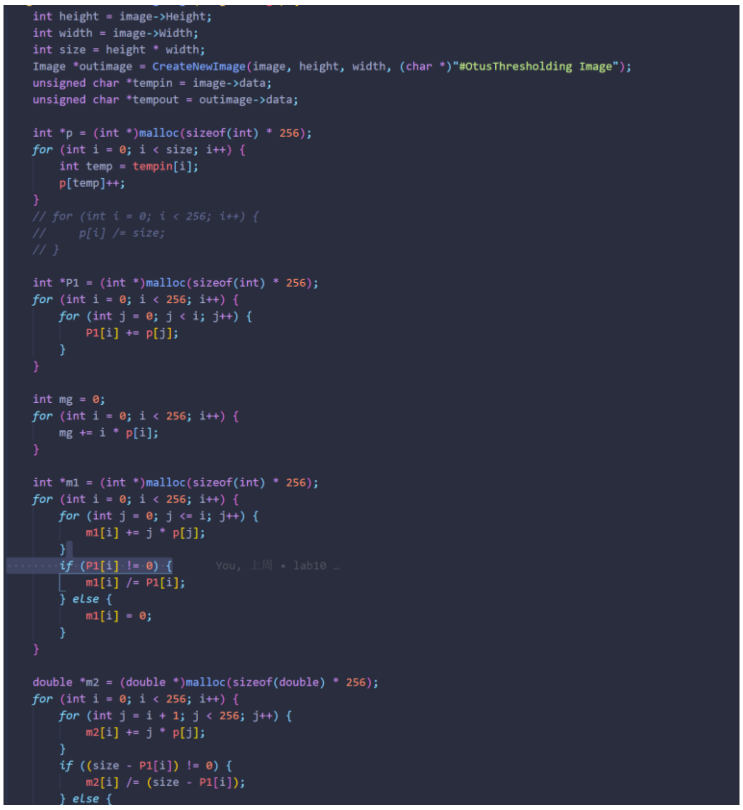
Result after 5X5 and Otus's:


large_septagon_gaussian_noise_mean_0_std_50_added_AverFilter_OtusThresholding.pgm
814x651 pixels; 8-bit; 517K

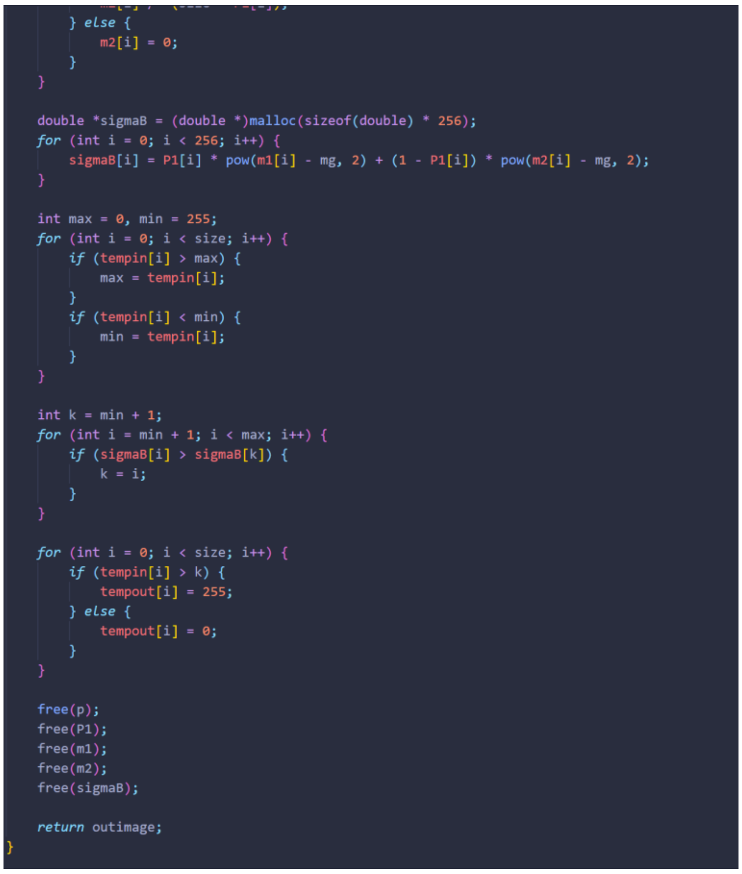
Discussion:

This algorithm is considered to be the best threshold selection algorithm in image segmentation. It is easy to calculate and is not affected by image brightness and contrast, so it has been widely used in digital image processing. It divides the image into background and foreground according to the gray characteristics of the image. Since variance is a measure of the uniformity of gray distribution, the greater the inter-class variance between background and foreground, the greater the difference between the two parts of the image. When part of the foreground is wrongly divided into background or part of the background is wrongly divided into foreground, the difference between the two parts will become smaller. Therefore, the segmentation that maximizes the variance between classes means the smallest misclassification probability.

Codes:



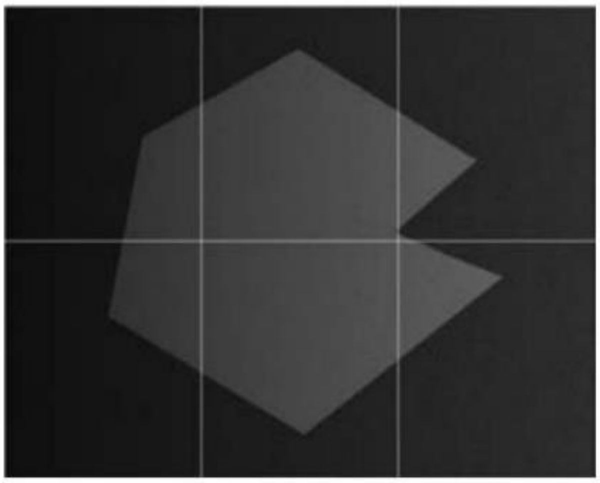
```c
    int height = image->Height;
    int width = image->Width;
    int size = height * width;
    Image *outimage = CreateNewImage(image, height, width, (char *)"#OtusThresholding Image");
    unsigned char *tempin = image->data;
    unsigned char *tempout = outimage->data;

    int *p = (int *)malloc(sizeof(int) * 256);
    for (int i = 0; i < size; i++) {
        int temp = tempin[i];
        p[temp]++;
    }
    // for (int i = 0; i < 256; i++) {
    //     p[i] /= size;
    // }

    int *P1 = (int *)malloc(sizeof(int) * 256);
    for (int i = 0; i < 256; i++) {
        for (int j = 0; j < i; j++) {
            P1[i] += p[j];
        }
    }

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

    int *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) {        You, 上周 • lab10 …
            m1[i] /= P1[i];
        } else {
            m1[i] = 0;
        }
    }

    double *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 ((size - P1[i]) != 0) {
            m2[i] /= (size - P1[i]);
        } else {
```



```c
        } else {
            m2[i] = 0;
        }
    }

    double *sigmaB = (double *)malloc(sizeof(double) * 256);
    for (int i = 0; i < 256; i++) {
        sigmaB[i] = P1[i] * pow(m1[i] - mg, 2) + (1 - P1[i]) * pow(m2[i] - mg, 2);
    }

    int max = 0, min = 255;
    for (int i = 0; i < size; 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; i++) {
        if (sigmaB[i] > sigmaB[k]) {
            k = i;
        }
    }

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

    free(p);
    free(P1);
    free(m1);
    free(m2);
    free(sigmaB);

    return outimage;
}
```

2. Partition(septagon_noisy_shaded):

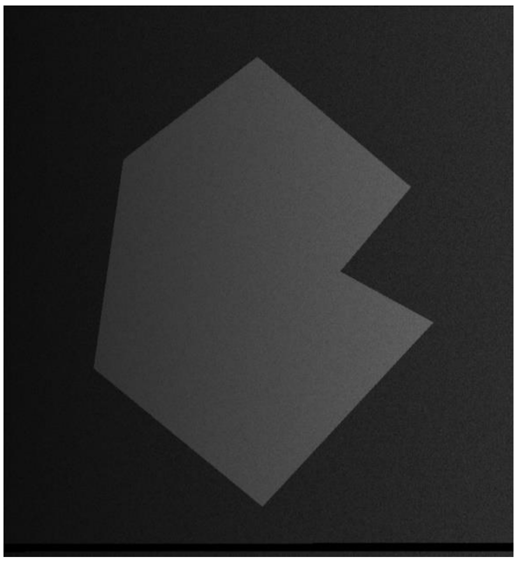
Algorithm:



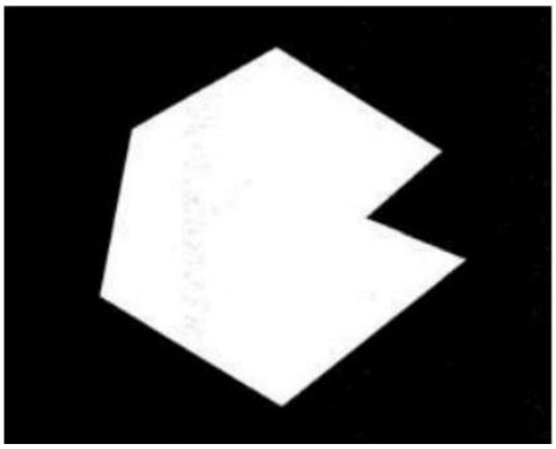
Results (including pictures):

Result of processing “septagon_noisy_shaded.pgm”:

Source Image:



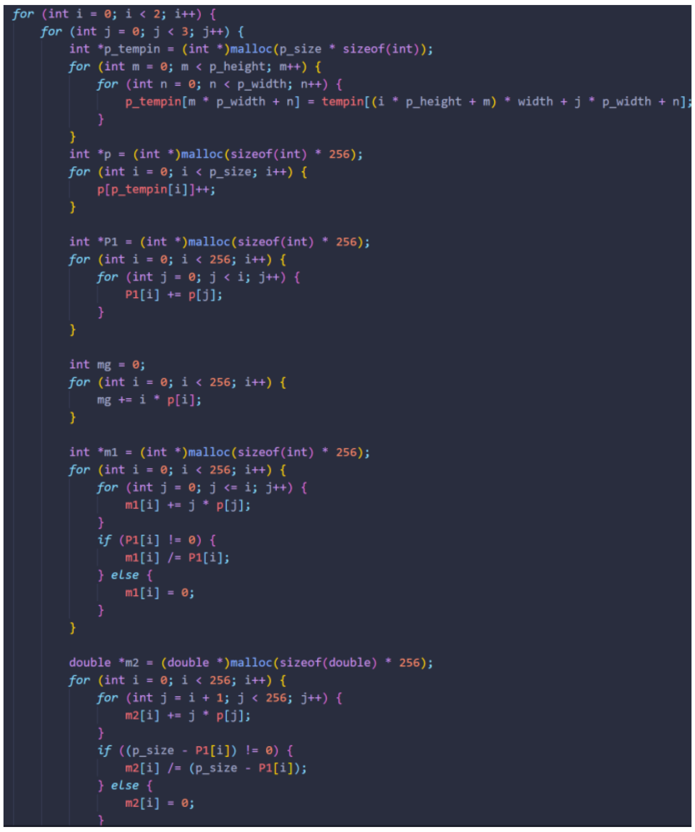
Result after Partition:



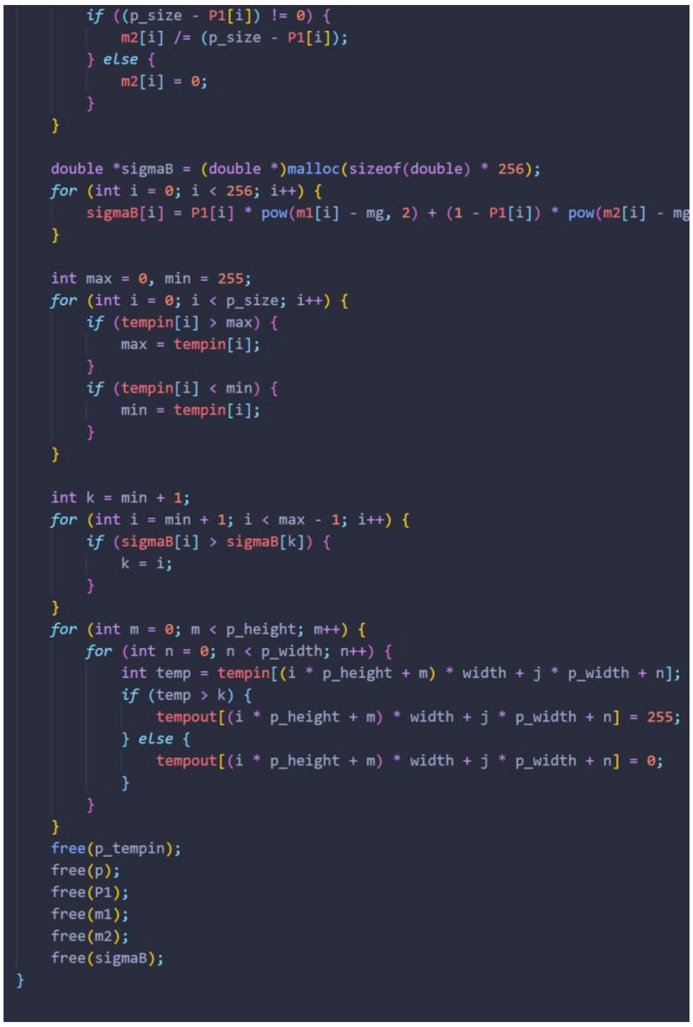
Discussion:

It can be seen that the method of local statistics variable threshold processing is very effective for image segmentation. Three distinct gray levels can be completely distinguished.

Codes:



```
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 3; j++) {
        int *p_tempin = (int *)malloc(p_size * sizeof(int));
        for (int m = 0; m < p_height; m++) {
            for (int n = 0; n < p_width; n++) {
                p_tempin[m * p_width + n] = tempin[(i * p_height + m) * width + j * p_width + n];
            }
        }
        int *p = (int *)malloc(sizeof(int) * 256);
        for (int i = 0; i < p_size; i++) {
            p[p_tempin[i]]++;
        }

        int *P1 = (int *)malloc(sizeof(int) * 256);
        for (int i = 0; i < 256; i++) {
            for (int j = 0; j < i; j++) {
                P1[i] += p[j];
            }
        }

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

        int *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;
            }
        }

        double *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 ((p_size - P1[i]) != 0) {
                m2[i] /= (p_size - P1[i]);
            } else {
                m2[i] = 0;
            }
```



```
        if ((p_size - P1[i]) != 0) {
            m2[i] /= (p_size - P1[i]);
        } else {
            m2[i] = 0;
        }
    }

    double *sigmaB = (double *)malloc(sizeof(double) * 256);
    for (int i = 0; i < 256; i++) {
        sigmaB[i] = P1[i] * pow(m1[i] - mg, 2) + (1 - P1[i]) * pow(m2[i] - mg
    }

    int max = 0, min = 255;
    for (int i = 0; i < p_size; 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 (sigmaB[i] > sigmaB[k]) {
            k = i;
        }
    }
    for (int m = 0; m < p_height; m++) {
        for (int n = 0; n < p_width; n++) {
            int temp = tempin[(i * p_height + m) * width + j * p_width + n];
            if (temp > k) {
                tempout[(i * p_height + m) * width + j * p_width + n] = 255;
            } else {
                tempout[(i * p_height + m) * width + j * p_width + n] = 0;
            }
        }
    }
    free(p_tempin);
    free(p);
    free(P1);
    free(m1);
    free(m2);
    free(sigmaB);
}
```

3. moving average thresholding (spot_shaded_text_image):

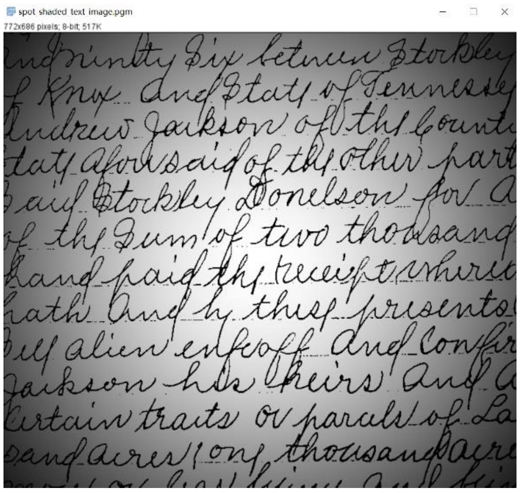
Algorithm:

- During scanning, pixels are visited in a zigzag path, so we compute the average of the intensity of last n visited pixels. The segmentation can be performed using a threshold $T_{xy} = bm_{xy}$, b can be a suitable constant such as $b = 0.5$. Such method is best used for document thresholding

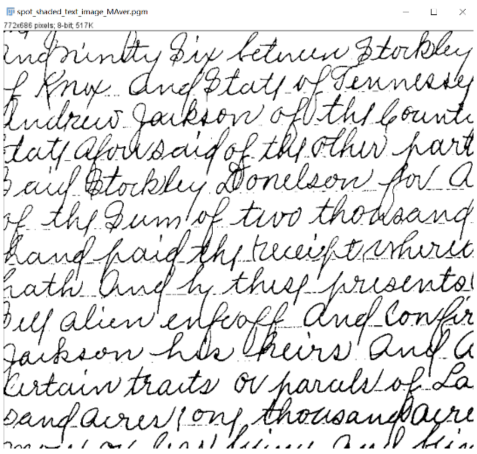
Results (including pictures):

Result of processing “spot_shaded_text_image”:

Source Image:



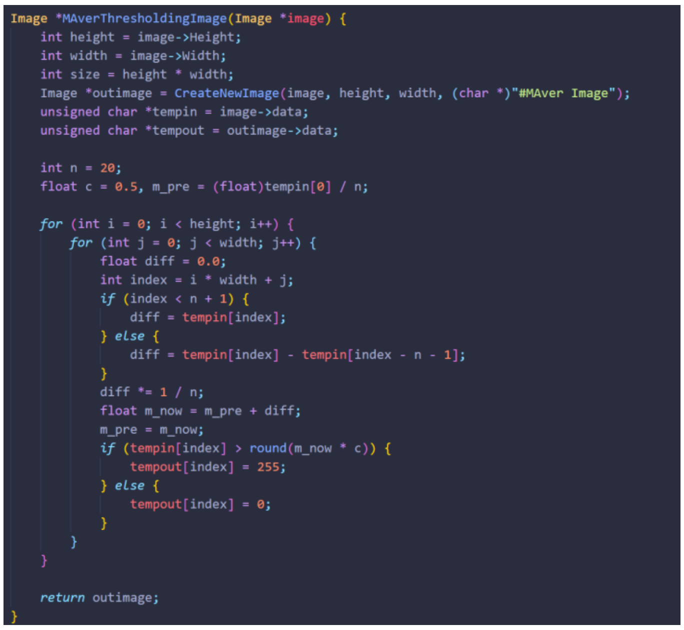
Result after moving average:



Discussion:

Figure shows an image of handwritten text shaded by a spot intensity pattern. This form of intensity shading is typical of images obtained using spot illumination (such as a photographic flash). Figure is the result of segmentation using the Otsu global thresholding method. It is not unexpected that global thresholding could not overcome the intensity variation because the method generally performs poorly when the areas of interest are embedded in a nonuniform illumination field. Figure shows successful segmentation with local thresholding using moving averages. For images of written material, a rule of thumb is to let n equal five times the average stroke width. In this case, the average width was 4 pixels, so we let n = 20 and used c = 0 5.

Codes:



```c
Image *MAverThresholdingImage(Image *image) {
    int height = image->Height;
    int width = image->Width;
    int size = height * width;
    Image *outimage = CreateNewImage(image, height, width, (char *)"#MAver Image");
    unsigned char *tempin = image->data;
    unsigned char *tempout = outimage->data;

    int n = 20;
    float c = 0.5, m_pre = (float)tempin[0] / n;

    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            float diff = 0.0;
            int index = i * width + j;
            if (index < n + 1) {
                diff = tempin[index];
            } else {
                diff = tempin[index] - tempin[index - n - 1];
            }
            diff *= 1 / n;
            float m_now = m_pre + diff;
            m_pre = m_now;
            if (tempin[index] > round(m_now * c)) {
                tempout[index] = 255;
            } else {
                tempout[index] = 0;
            }
        }
    }

    return outimage;
}
```

4. region growing method (defective_weld.pgm):

Algorithm:

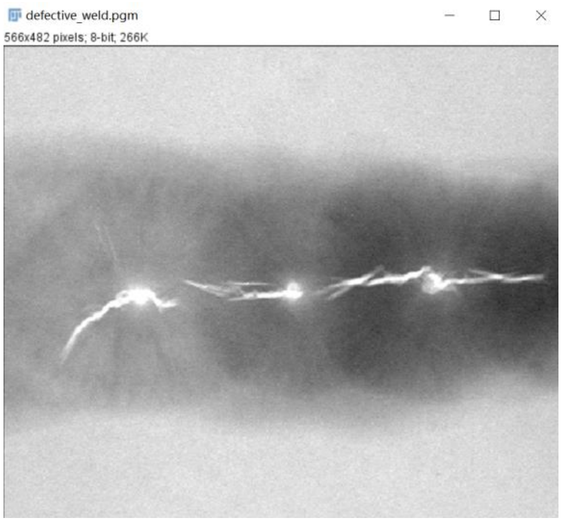
- Start with a set of “seed” points
- Grow regions by appending to each seed those neighboring pixels that have predefined properties similar to the seed (e.g., gray level, texture…)

- The selection of the *seeds* can be operated manually or using automatic procedures based on appropriate criteria:
  - A-priori knowledge can be included
  - It is strictly application-dependent
- The growing is controlled by the connectivity
- The stop rule is another parameter of the algorithm. It can depend on the a priori knowledge on the problem or geometric structures

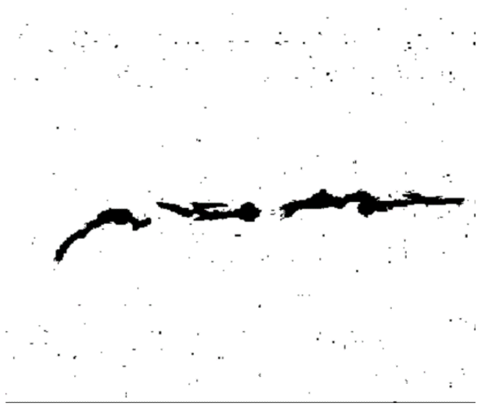
Results (including pictures):

Result of processing “defective_weld.pgm”:

Source Image:



Result after region growing:

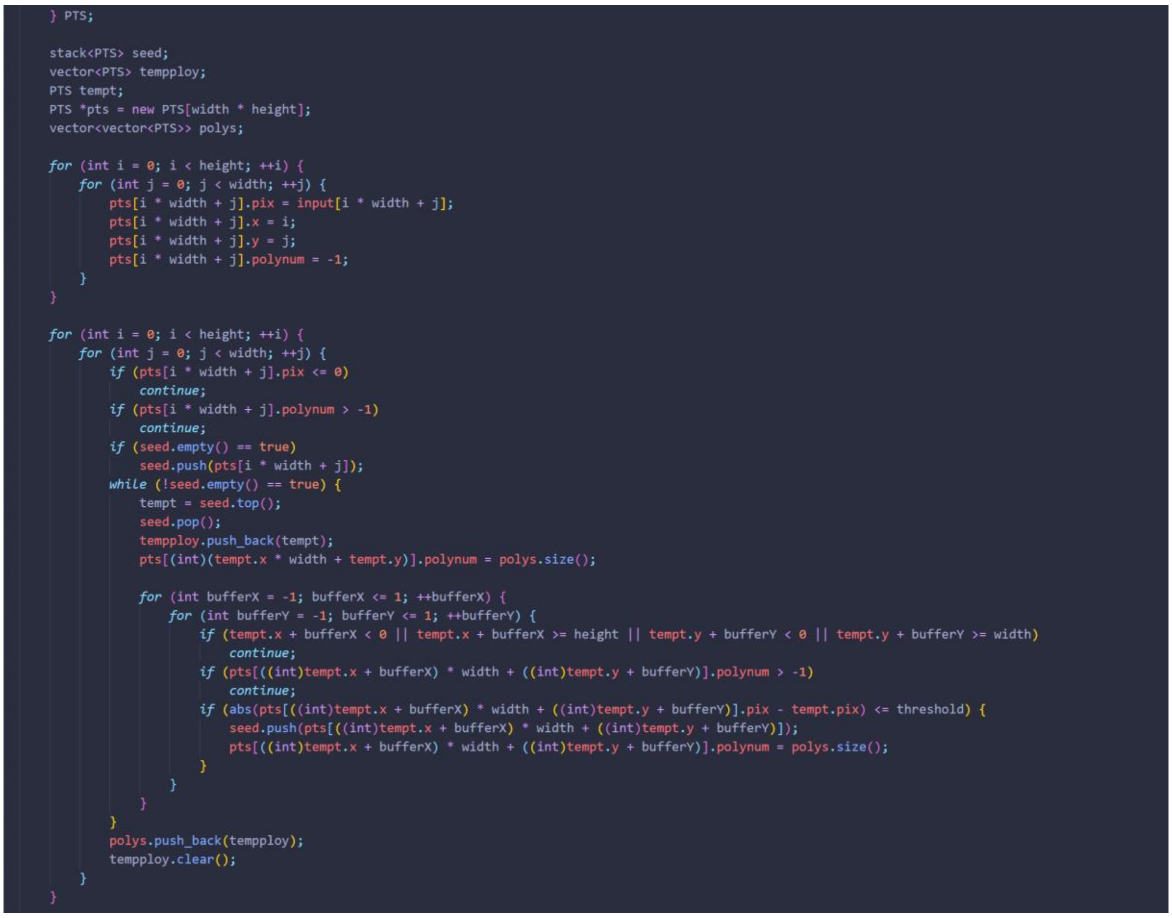


Discussion:

Advantages: The basic idea is relatively simple, and it can usually divide the connected regions with the same characteristics, and provide good boundary information and segmentation results. When there is no prior knowledge to use, it can achieve the best performance, and can be used to segment more complex images, such as natural scenery, coins, medical images, etc.

Disadvantages: Region growth method is an iterative method, with large space and time overhead. Uneven noise and gray scale may lead to void and over-segmentation, and it is often not very good in the processing of shadow effect in the image.

Codes:



```
    } PTS;

    stack<PTS> seed;
    vector<PTS> tempploy;
    PTS tempt;
    PTS *pts = new PTS[width * height];
    vector<vector<PTS>> polys;

    for (int i = 0; i < height; ++i) {
        for (int j = 0; j < width; ++j) {
            pts[i * width + j].pix = input[i * width + j];
            pts[i * width + j].x = i;
            pts[i * width + j].y = j;
            pts[i * width + j].polynum = -1;
        }
    }

    for (int i = 0; i < height; ++i) {
        for (int j = 0; j < width; ++j) {
            if (pts[i * width + j].pix <= 0)
                continue;
            if (pts[i * width + j].polynum > -1)
                continue;
            if (seed.empty() == true)
                seed.push(pts[i * width + j]);
            while (!seed.empty() == true) {
                tempt = seed.top();
                seed.pop();
                tempploy.push_back(tempt);
                pts[(int)(tempt.x * width + tempt.y)].polynum = polys.size();

                for (int bufferX = -1; bufferX <= 1; ++bufferX) {
                    for (int bufferY = -1; bufferY <= 1; ++bufferY) {
                        if (tempt.x + bufferX < 0 || tempt.x + bufferX >= height || tempt.y + bufferY < 0 || tempt.y + bufferY >= width)
                            continue;
                        if (pts[((int)tempt.x + bufferX) * width + ((int)tempt.y + bufferY)].polynum > -1)
                            continue;
                        if (abs(pts[((int)tempt.x + bufferX) * width + ((int)tempt.y + bufferY)].pix - tempt.pix) <= threshold) {
                            seed.push(pts[((int)tempt.x + bufferX) * width + ((int)tempt.y + bufferY)]);
                            pts[((int)tempt.x + bufferX) * width + ((int)tempt.y + bufferY)].polynum = polys.size();
                        }
                    }
                }
            }
            polys.push_back(tempploy);
            tempploy.clear();
        }
    }
```