Class Lab 9

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

1. Roberts (headCT_Vandy, building_original, noisy_fingerprint):

Algorithm:

$$\frac{\partial f(x,y)}{\partial x} = f(x+1,y) - f(x,y)$$
$$\frac{\partial f(x,y)}{\partial y} = f(x,y+1) - f(x,y)$$

−1	0
0	1

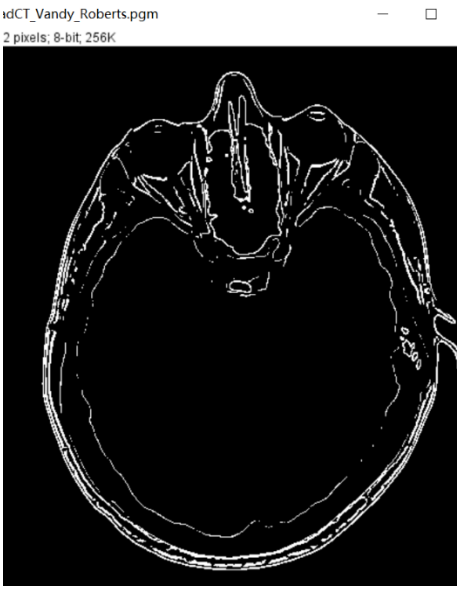
0	−1
1	0

Results (including pictures):

Result of processing “headCT_Vandy.pgm”:

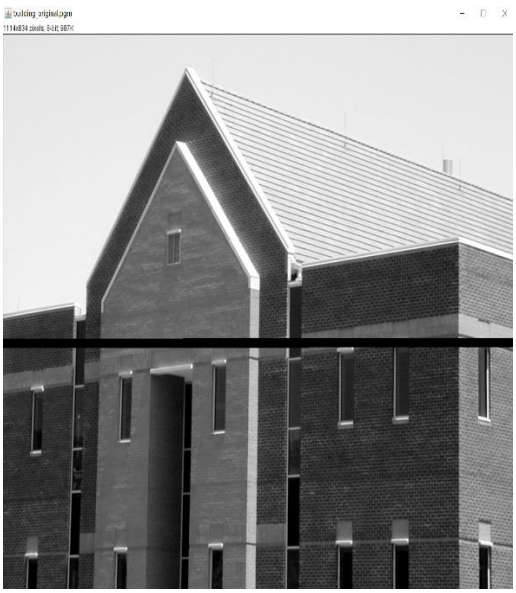
Source Image:

Result after Roberts:

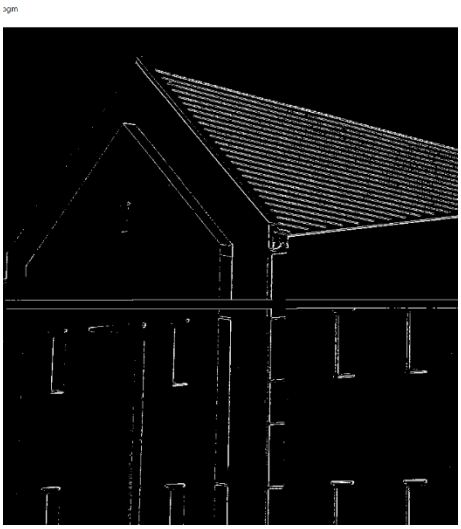


Result of processing “building_original.pgm”:

Source Image:



Result after Roberts:



Result of processing “noisy_fingerprint.pgm”:

Source Image:

Result after Roberts:

Discussion:

As a first-order differential operator, Robert is simple, requires little computation, and is sensitive to details. The role of the operator for edge detection is to provide edge candidate points. Compared with other 3x3 operators, the Robert operator can provide relatively finer edges without post-processing.

Codes:

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

    int filterx[2][2] = {{-1, 0}, {0, 1}};
    int filtery[2][2] = {{0, -1}, {1, 0}};

    for (int i = 1; i < height - 1; i++) {
        for (int j = 1; j < width - 1; j++) {
            int sum1 = 0;
            int sum2 = 0;
            for (int m = 0; m < 2; m++) {
                for (int n = 0; n < 2; n++) {
                    sum1 += tempin[(i + m) * width + j + n] * filterx[m][n];
                    sum2 += tempin[(i + m) * width + j + n] * filtery[m][n];
                }
            }
            int res = abs(sum1) + abs(sum2);
            if (res > 255) {
                res = 255;      You, 昨天 • lab9 …
            }
            tempout[i * width + j] = res;
        }
    }

    Threshold(tempout, 0.33, size);

    return outimage;
}
```

2. Prewitt (headCT_Vandy, building_original, noisy_fingerprint):

Algorithm:

$$\frac{\partial f(x, y)}{\partial x} = f(x + 1, y) - f(x, y)$$
$$\frac{\partial f(x, y)}{\partial y} = f(x, y + 1) - f(x, y)$$

−1	−1	−1	−1	0	1
0	0	0	−1	0	1
1	1	1	−1	0	1

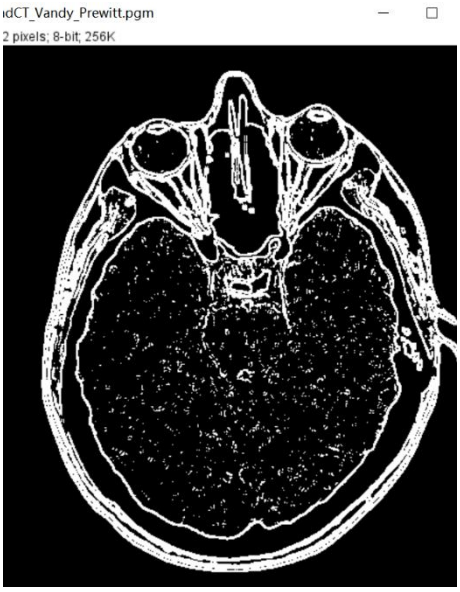
Prewitt

Results (including pictures):

Result of processing “headCT_Vandy.pgm”:

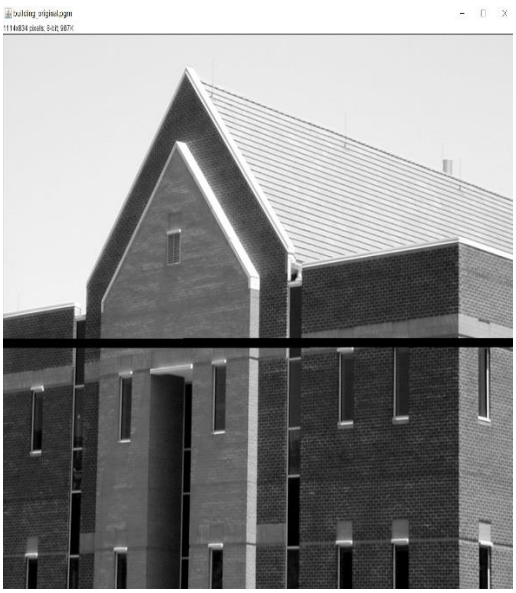
Source Image:

Result after Prewitt:

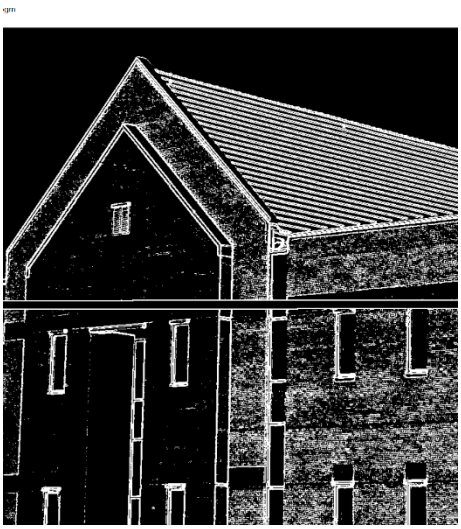


Result of processing “building_original.pgm”:

Source Image:



Result after Prewitt:



Result of processing “noisy_fingerprint.pgm”:

Source Image:

Result after Prewitt:

Discussion:

Since the Prewitt operator uses a 3x3 template to calculate the pixel values in the region, while the Robert operator's template is 2x2, the edge detection results of the Prewitt operator are more obvious than those of the Robert operator in both the horizontal and vertical directions. Prewitt operator is suitable for identifying images with more noise and grayscale gradient

Codes:

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

    int filterx[3][3] = {{-1, -1, -1}, {0, 0, 0}, {1, 1, 1}};
    int filtery[3][3] = {{-1, 0, 1}, {-1, 0, 1}, {-1, 0, 1}};

    for (int i = 1; i < height - 1; i++) {
        for (int j = 1; j < width - 1; j++) {
            int sum1 = 0;
            int sum2 = 0;
            for (int m = -1; m < 2; m++) {
                for (int n = -1; n < 2; n++) {
                    sum1 += tempin[(i + m) * width + j + n] * filterx[m + 1][n + 1];
                    sum2 += tempin[(i + m) * width + j + n] * filtery[m + 1][n + 1];
                }
            }
            int res = abs(sum1) + abs(sum2);
            if (res > 255) {
                res = 255;
            }
            tempout[i * width + j] = res;
        }
    }

    Threshold(outimage->data, 0.33, size);

    return outimage;
}
```

3. Soble (headCT_Vandy, building_original, noisy_fingerprint):

Algorithm:

$$\frac{\partial f(x,y)}{\partial x} = f(x+1,y) - f(x,y)$$
$$\frac{\partial f(x,y)}{\partial y} = f(x,y+1) - f(x,y)$$

−1	−2	−1
0	0	0
1	2	1

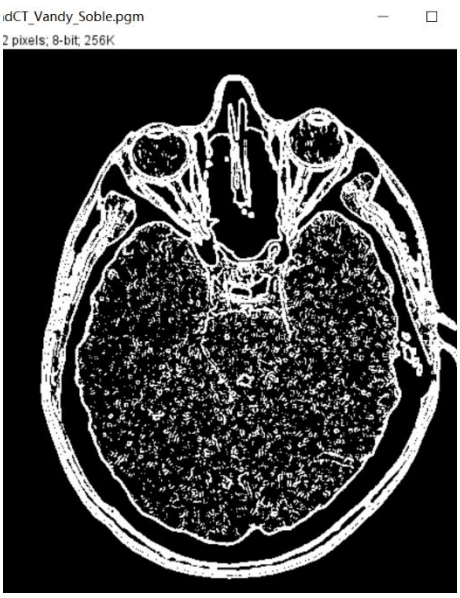
−1	0	1
−2	0	2
−1	0	1

Results (including pictures):

Result of processing “building_original.pgm”:

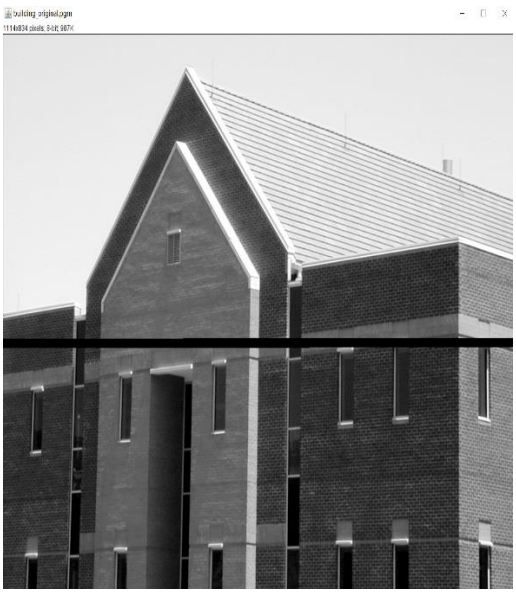
Source Image:

Result after Soble:

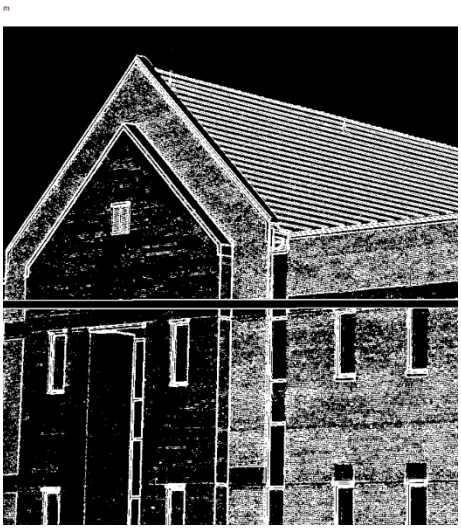


Result of processing “headCT_Vandy.pgm”:

Source Image:



Result after Soble:



Result of processing “noisy_fingerprint.pgm”:

Source Image:

Result after Soble:

Discussion:

Sobel operator not only produces better detection effect, but also has a smooth suppression effect on noise, but the obtained edge is thicker and may appear false edge. Compared with the Prewitt operator, the Sobel operator has weighted the influence of the position of the pixel, which can reduce the blurring degree of the edge, so the effect is better.

Codes:

```
Image *SobleGradientImage(Image *image) {       You, 昨天 • lab9 …
    int height = image->Height;
    int width = image->Width;
    int size = height * width;
    Image *outimage = CreateNewImage(image, height, width, (char *)"#SobleGradient Image");
    unsigned char *tempin = image->data;
    unsigned char *tempout = outimage->data;

    int filterx[3][3] = {{-1, -2, -1}, {0, 0, 0}, {1, 2, 1}};
    int filtery[3][3] = {{-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1}};

    for (int i = 1; i < height - 1; i++) {
        for (int j = 1; j < width - 1; j++) {
            int sum1 = 0;
            int sum2 = 0;
            for (int m = -1; m < 2; m++) {
                for (int n = -1; n < 2; n++) {
                    sum1 += tempin[(i + m) * width + j + n] * filterx[m + 1][n + 1];
                    sum2 += tempin[(i + m) * width + j + n] * filtery[m + 1][n + 1];
                }
            }
            int res = abs(sum1) + abs(sum2);
            if (res > 255) {
                res = 255;
            }
            tempout[i * width + j] = res;
        }
    }

    Threshold(outimage->data, 0.33, size);

    return outimage;
}
```

4. Canny Edge (headCT_Vandy, noisy_fingerprint):

Algorithm:

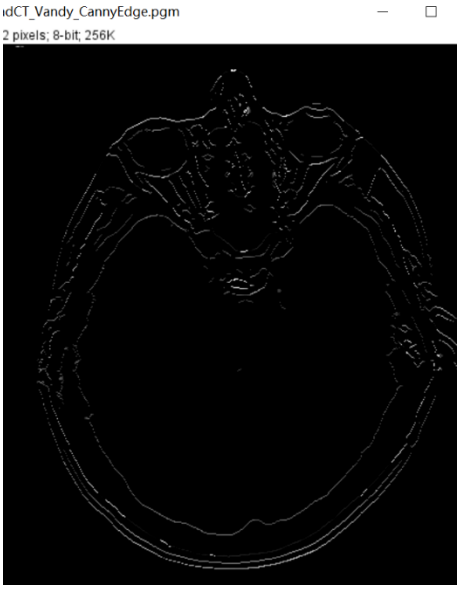
1. Smooth input image with a Gaussian filter (remove noise)
2. Compute the gradient magnitude and angle images
3. Apply nonmaxima suppression to the gradient magnitude image
4. Use double thresholding and connectivity analysis to detect and link edges

Results (including pictures):

Result of processing "headCT_Vandy.pgm":

Source Image:

Result after Canny:



Result of processing "noisy_fingerprint.pgm":

Source Image:

Result after Canny:

Discussion:

The details of the main edges are clearly improved in the Canny results, while more irrelevant features are suppressed. Continuity, fineness and straightness of the lines are also better in the Canny image.

Codes:

```
Image *outimage = CreateNewImage(image, height, width, (char *)"#CannyEdge Image");
unsigned char *tempin = image->data;
unsigned char *tempout = outimage->data;

int gsize = 0;
while (1) {
    if (gsize > 6 * sigma) {
        break;
    }
    gsize++;
}

double gaussian[gsize * gsize];

int k = (gsize - 1) / 2;
for (int i = 0; i < gsize; i++) {
    for (int j = 0; j < gsize; j++) {
        double e = -(double)(pow((i - k - 1), 2) + pow((j - k - 1), 2)) / (2 * sigma * sigma);
        gaussian[i * gsize + j] = (double)1 / (2 * PI * sigma * sigma) * exp(e);
    }
}

for (int i = k; i < height - k; i++) {
    for (int j = k; j < width - k; j++) {
        double sum = 0;
        for (int m = 0; m < gsize; m++) {
            for (int n = 0; n < gsize; n++) {
                int x = i + m - k;
                int y = j + n - k;
                sum += tempin[x * width + y] * gaussian[m * gsize + n];
            }
        }
        tempout[i * width + j] = sum;
    }
}

double *Ms = (double *)malloc(sizeof(double) * size);
memset(Ms, 0, sizeof(double) * size);
double *A = (double *)malloc(sizeof(double) * size);
memset(A, 0, sizeof(double) * size);

int filterx[3][3] = {{-1, -2, -1}, {0, 0, 0}, {1, 2, 1}};
int filtery[3][3] = {{-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1}};

for (int i = 1; i < height - 1; i++) {
    for (int j = 1; j < width - 1; j++) {
        int sum1 = 0;
        int sum2 = 0;
        for (int m = -1; m < 2; m++) {
            for (int n = -1; n < 2; n++) {
                sum1 += tempout[(i + m) * width + j + n] * filterx[m + 1][n + 1];
                sum2 += tempout[(i + m) * width + j + n] * filtery[m + 1][n + 1];
            }
        }
        sum1 = sum1 > 255 ? 255 : sum1;
        sum2 = sum2 > 255 ? 255 : sum2;
        Ms[i * width + j] = sqrt(pow(sum1, 2) + pow(sum2, 2));
        A[i * width + j] = atan2(sum2, sum1) * 180 / PI;
    }
```

```
    }
}

double *g = (double *)malloc(sizeof(double) * size);
memset(g, 0, sizeof(double) * size);
double *gh = (double *)malloc(sizeof(double) * size);
memset(gh, 0, sizeof(double) * size);
double *gl = (double *)malloc(sizeof(double) * size);
memset(gl, 0, sizeof(double) * size);
double *temp = (double *)malloc(sizeof(double) * size);
memset(gl, 0, sizeof(double) * size);
int Th = 60, Tl = 30;
for (int i = 1; i < height - 1; i++) {
    for (int j = 1; j < width - 1; j++) {
        double angle = A[i * width + j];
        double ms = Ms[i * width + j];
        if (angle > -22.5 && angle < 22.5 || angle < 157.5 && angle > -157.5) {
            if (ms < Ms[(i - 1) * width + j] || ms < Ms[(i + 1) * width + j]) {
                g[i * width + j] = 0;
            } else {
                g[i * width + j] = ms;
            }
        } else if (angle > 67.5 && angle < 112.5 || angle < -67.5 && angle > -112.5) {
            if (ms < Ms[i * width + j - 1] || ms < Ms[i * width + j + 1]) {
                g[i * width + j] = 0;
            } else {
                g[i * width + j] = ms;
            }
        } else if (angle < 157.5 && angle > 112.5 || angle > -22.5 && angle < -67.5) {
            if (ms < Ms[(i - 1) * width + j + 1] || ms < Ms[(i + 1) * width + j - 1]) {
                g[i * width + j] = 0;
            } else {
                g[i * width + j] = ms;
            }
        } else {
            if (ms < Ms[(i - 1) * width + j - 1] || ms < Ms[(i + 1) * width + j + 1]) {
                g[i * width + j] = 0;
            } else {
                g[i * width + j] = ms;
            }
        }
        if (g[i * width + j] > Tl) {
            gl[i * width + j] = 1;
            temp[i * width + j] = g[i * width + j];
        } else {
            gl[i * width + j] = 0;
        }
        if (g[i * width + j] > Th) {
            gh[i * width + j] = 1;
            temp[i * width + j] = g[i * width + j];
        } else {
            gh[i * width + j] = 0;
        }
        gl[i * width + j] -= gh[i * width + j];
    }
}
```

```
        for (int j = 0; j < width; j++) {
            int p5 = i * width + j;
            if (gh[p5] == 1) {
                int p1 = (i - 1) * width + (j - 1);
                int p2 = (i - 1) * width + j;
                int p3 = (i - 1) * width + (j + 1);
                int p4 = i * width + (j - 1);
                int p6 = i * width + (j + 1);
                int p7 = (i + 1) * width + (j - 1);
                int p8 = (i + 1) * width + j;
                int p9 = (i + 1) * width + (j + 1);
                if (gl[p1] == 1) {
                    gl[p1] = 1.5;
                }
                if (gl[p2] == 1) {
                    gl[p2] = 1.5;
                }
                if (gl[p3] == 1) {
                    gl[p3] = 1.5;
                }
                if (gl[p4] == 1) {
                    gl[p4] = 1.5;
                }
                if (gl[p5] == 1) {
                    gl[p4] = 1.5;
                }
                if (gl[p6] == 1) {
                    gl[p6] = 1.5;
                }
                if (gl[p7] == 1) {
                    gl[p7] = 1.5;
                }
                if (gl[p8] == 1) {
                    gl[p8] = 1.5;
                }
                if (gl[p9] == 1) {
                    gl[p9] = 1.5;
                }
            }
        }
    }

    for (int i = 0; i < size; i++) {
        if (gl[i] != 1.5 && gh[i] != 1) {
            temp[i] = 0;
        }
        outimage->data[i] = temp[i];
    }

    free(g);
    free(gh);
    free(gl);
    free(temp);
    free(Ms);
    free(A);

    return outimage;
}
```

5. Global Thresholding (polymersomes, noisy_fingerprint):

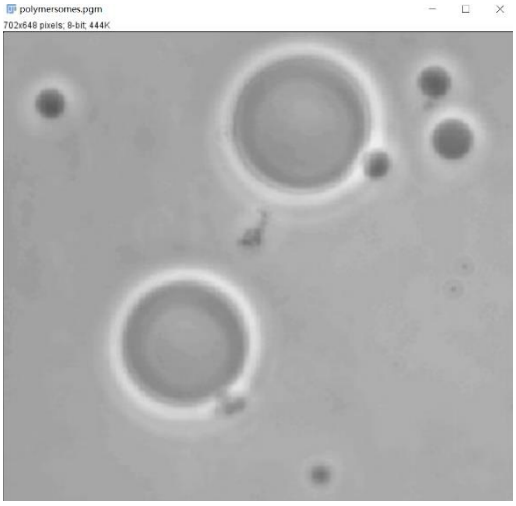
Algorithm:

1. Select an initial threshold T: usually, $T = (\text{gray}_{\max} + \text{gray}_{\min})/2$
2. Segment the image using T: two groups of pixels → G_1 ($< T$) and G_2 ($\geq T$)
3. Compute average intensities of G_1 and G_2: μ_1 and μ_2
4. Compute a new threshold T: $T = (\mu_1 + \mu_2)/2$
5. Repeat Steps 2 to 4 until the difference in T in successive iterations is smaller than a predefined parameter T_0

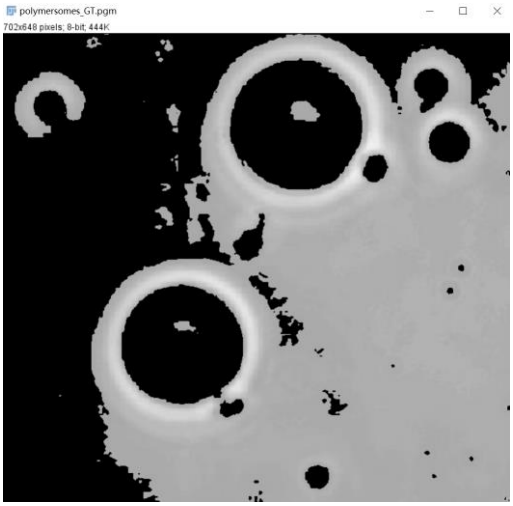
Results (including pictures):

Result of processing "polymersomes.pgm":

Source Image:



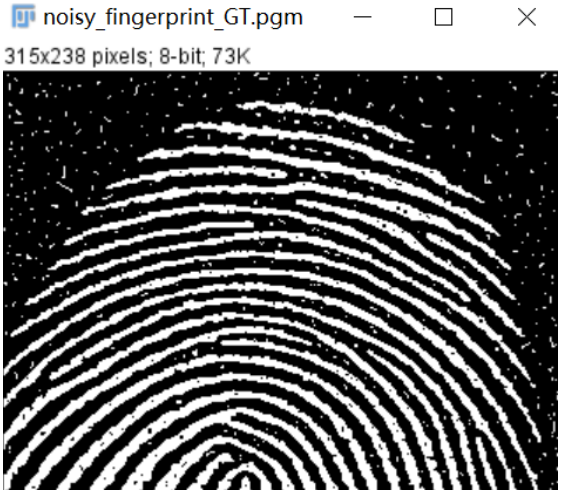
Result after Global:



Result of processing "noisy_fingerprint.pgm":

Source Image:

Result after Global:



Discussion:

Global thresholding is very effective when the grayscale distributions of target and background pixels are very different. But for similar binary images, this thresholding cannot remove noise very well.

Codes:

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

float threshold = (max + min) / 2.0;

int *G1 = (int *)malloc(sizeof(int) * size);
memset(G1, 0, sizeof(int) * size);
int *G2 = (int *)malloc(sizeof(int) * size);
memset(G2, 0, sizeof(int) * size);
int *pos = (int *)malloc(sizeof(int) * size);
memset(pos, 0, sizeof(int) * size);

while (1) {
    int m = 0, n = 0;
    for (int i = 0; i < size; i++) {
        if (tempin[i] < threshold) {
            G1[m] = tempin[i];
            m++;
        } else {
            G2[n] = tempin[i];
            pos[i] = tempin[i];
            n++;
        }
    }

    int m1 = 0, m2 = 0;
    for (int i = 0; i < m; i++) {
        m1 += G1[i];
    }
    m1 /= m;

    for (int i = 0; i < n; i++) {
        m2 += G2[i];
    }
    m2 /= n;

    float temp = (m1 + m2) / 2.0;
    if (abs(temp - threshold) < 1) {
        break;
    } else {
        threshold = temp;
        memset(G1, 0, sizeof(int) * size);
        memset(G2, 0, sizeof(int) * size);
        memset(pos, 0, sizeof(int) * size);
    }
}
printf("threshold: %f\n", threshold);
```