

## Lab 1

1. Use Roberts, Prewitt, Sobel gradient operators to obtain gradient images, then threshold the images to compare the results among different operators. The images are headCT\_Vandy.pgm, building\_original.pgm, noisy\_fingerprint.pgm
2. Implement Canny edge detection and LoG detection algorithms on headCT\_Vandy.pgm and noisy\_fingerprint.pgm
3. Use global thresholding to perform segmentation separately on polymersomes.pgm and noisy\_fingerprint.pgm

### Algorithm:

1. Gradient images

- a) Roberts

$$g(x, y) = |-f(i, j) + f(i + 1, j + 1)| + |f(i + 1, j) - f(i, j + 1)|$$

-1	0
0	1

0	-1
1	0

Roberts

Where f is the original image.

- b) Prewitt

$$g_x(x, y) = |f(i + 1, j - 1) + f(i + 1, j) + f(i + 1, j + 1) - f(i - 1, j - 1) - f(i - 1, j) - f(i - 1, j + 1)|$$

$$g_y(x, y) = |f(i - 1, j + 1) + f(i, j + 1) + f(i + 1, j + 1) - f(i - 1, j - 1) - f(i, j - 1) - f(i + 1, j - 1)|$$

$$g(x, y) = g_x(x, y) + g_y(x, y)$$

-1	-1	-1
0	0	0
1	1	1

-1	0	1
-1	0	1
-1	0	1

Prewitt

Where f is the original image.

- c) Sobel

$$g_x(x, y) = |f(i + 1, j - 1) + 2f(i + 1, j) + f(i + 1, j + 1) - f(i - 1, j - 1) - 2f(i - 1, j) - f(i - 1, j + 1)|$$

$$g_y(x, y) = |f(i - 1, j + 1) + 2f(i, j + 1) + f(i + 1, j + 1) - f(i - 1, j - 1) - 2f(i, j - 1) - f(i + 1, j - 1)|$$

-1	-2	-1
0	0	0
1	2	1

-1	0	1
-2	0	2
-1	0	1

Sobel

$$g(x, y) = g_x(x, y) + g_y(x, y)$$

## 2. Edge Detection

### a) Canny

Step1: Smooth the image with a Gaussian filter

$$f_s(x, y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{m^2+n^2}{2\sigma^2}} f(x, y)$$

Step2: Compute the gradient magnitude and angle images

$$g_x = \frac{\partial f_s}{\partial x}, g_y = \frac{\partial f_s}{\partial y}$$

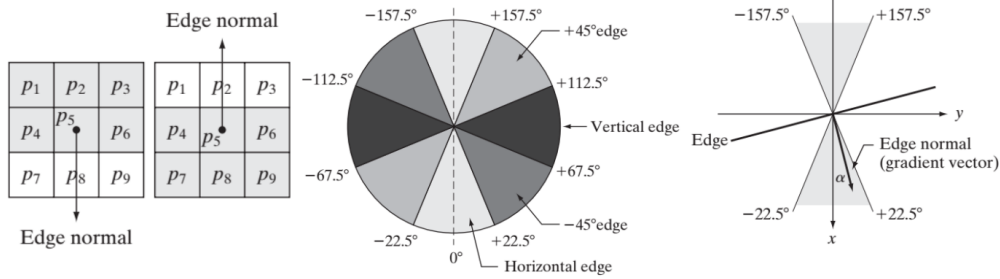
$$M(x, y) = \sqrt{g_x^2 + g_y^2}, \alpha(x, y) = \tan^{-1}\left(\frac{g_x}{g_y}\right)$$

Step3: Apply nonmaximal suppression to the gradient magnitude image

1) Find the direction  $d_k$  that is closest to  $\alpha(x, y)$

$$g_N(x, y) = \begin{cases} 0, & \text{if } M(x, y) < \text{at least one of its two neighbors along } d_k \\ M(x, y), & \text{otherwise} \end{cases}$$

Here  $d_1, d_2, d_3, d_4$  denote horizontal,  $-45^\circ$ , vertical, and  $45^\circ$  edge directions



Where  $g_N(x, y)$  is the nonmaximal suppressed image

Step4: Use double thresholding and connectivity analysis to detect and link edges

Use two thresholds  $T_H$  and  $T_L$  with ratio 2:1 or 3:1 to generate two images with initial pixel values set to 0:

$$\text{Strong edge image : } g_{NH}(x, y) = g_N(x, y) \geq T_H$$

$$g_{NL}(x, y) = g_N(x, y) \geq T_L$$

*Weak edge image :  $g_{NL}(x, y) = g_N(x, y) - g_{NH}(x, y)$ , to remove the strong pixels*

- 1) Locate the next unvisited edge pixel  $p$  in  $g_{NH}(x, y)$
- 2) Mark as valid edge pixels all the weak pixels in  $g_{NL}(x, y)$  that are connected to  $p$  using 8-connectivity
- 3) If all nonzero pixels in  $g_{NH}(x, y)$  have been visited, then go to the step 4. Else return to step 1
- 4) Set to zero all pixels in  $g_{NL}(x, y)$  that were not marked as valid edge pixels
- 5) Append nonzero pixels from  $g_{NL}(x, y)$  to  $g_{NH}(x, y)$

### b) LoG

$$\nabla^2 G(x, y) = \frac{\partial^2 G(x, y)}{\partial x^2} + \frac{\partial^2 G(x, y)}{\partial y^2} = \frac{\partial}{\partial x} \left[ \frac{-x}{\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \right] + \frac{\partial}{\partial y} \left[ \frac{-y}{\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \right]$$

$$g(x, y) = [\nabla^2 G(x, y)] * f(x, y) = \nabla^2 [G(x, y) * f(x, y)]$$

Where  $g(x, y)$  is the output image,  $f(x, y)$  is the input image

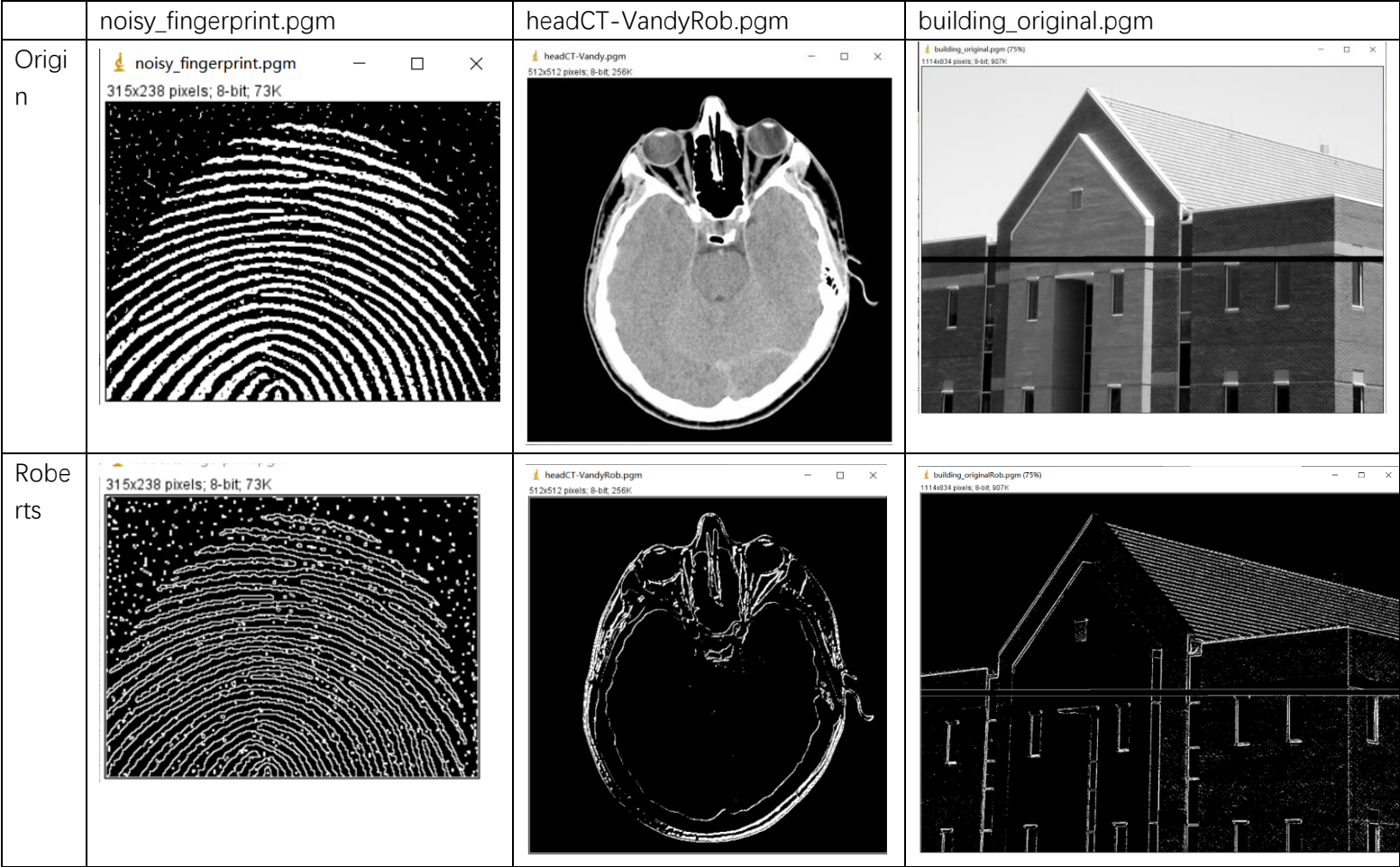
The mask is


0	0	-1	0	0
0	-1	-2	-1	0
-1	-2	16	-2	-1
0	-1	-2	-1	0
0	0	-1	0	0

3. Global Thresholding
- Step 1. Select an initial estimate threshold  $T$ . Here we set  $T = (\text{gray}_{\min} + \text{gray}_{\max})/2$
- Step 2. Segment the image with  $T$  to produce two sets of pixels:  $G_1 > T$ ,  $G_2 \leq T$
- Step 3. Calculate the average values  $\mu_1$  and  $\mu_2$  for the pixels in  $G_1$  and  $G_2$ , respectively
- Step 4. Compute a new threshold  $T = (\mu_1 + \mu_2)/2$
- Step 5. Repeat steps 2-4 until the difference of  $T$  values in successive iterations is smaller than a predefined parameter  $T_0$



Results (compare the results with the original image):

1. Gradient images



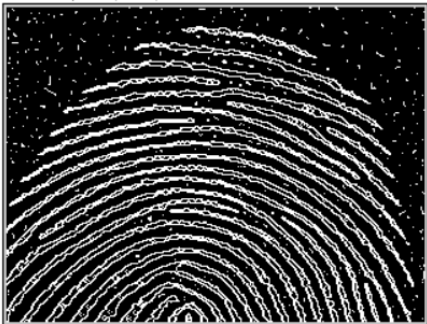
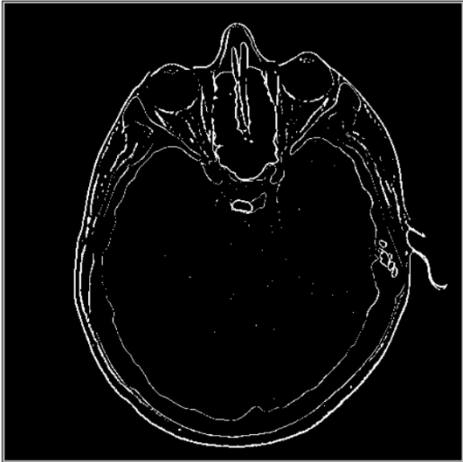


Prewitt			
Sobel			


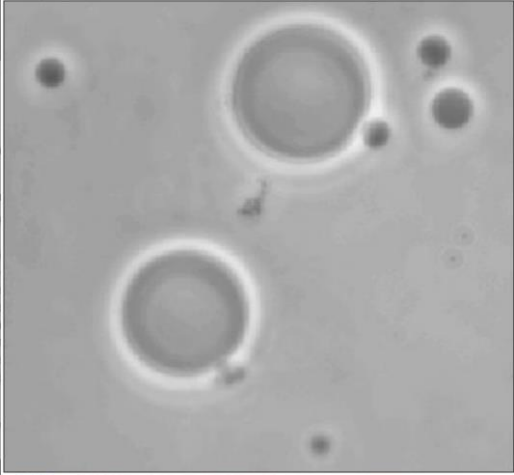
2. Edge detection

	noisy_fingerprint.pgm	headCT-VandyRob.pgm
Origin		



Canny		
LoG		

3. Global Thresholding

	noisy_fingerprint.pgm	polymersomes.pgm
Original		

**Discussion:**

## 1. Gradient images

- 1) Roberts operator is commonly used to process low noise images with steepness. When the edge of the image is close to positive or negative 45 degrees, the algorithm is more ideal. Its disadvantage is that the edge positioning is not very accurate, the extracted edge lines are thicker.
- 2) Prewitt operator uses 3\*3 mask to calculate the pixel values in the region, while the Robert operator's mask is 2\*2, so the edge detection results of the Prewitt operator are clearer in both horizontal and vertical directions than that of the Robert operator. Prewitt operator is suitable to recognize the image with more noise and gray gradient.
- 3) Sobel operator adds the consideration of weight based on Prewitt operator and believes that the distance of adjacent points has different influences on the current pixel point. The closer the distance is, the greater the influence of the current pixel point, to achieve image sharpening and highlight the edge contour. Sobel operator is more accurate in edge positioning, which is often used in images with more noise and gray gradient.

## 2. Edge detection

- 1) The goal of Canny operator is to detect edges as close to the actual edges as possible, and as much as possible, at the same time, to reduce the noise interference to edge detection.
- 2) LoG is born for improving the performance of Laplace operator. Because Laplace operator is sensitive to discrete points and noise when it implements edge detection through image manipulation. Therefore, firstly, Gaussian filter is used for image denoising processing, and then Laplace operator is used for edge detection, which can improve the anti-noise ability of the operator

## 3. Global Thresholding

The global threshold method determines a threshold value according to the histogram or gray spatial distribution of the image, to realize the transformation from gray image to binary image.

**Codes:**

## 1. Roberts

```

Image* Roberts(Image* image) {
    Image* outimage;
    int i, j, p, q, sumX, sumY, res;
    unsigned char* tempin, * tempout;
    int size = image->Height * image->Width;

    outimage = CreateNewImage(image, "Roberts", image->Width, image->Height);
    tempout = outimage->data;
    tempin = image->data;

    int X[2][2] = { {-1, 0}, {0, 1} };
    int Y[2][2] = { {0, -1}, {1, 0} };

    for (i = 1; i < image->Height - 1; i++) {
        for (j = 1; j < image->Width; j++) {
            sumX = 0;
            sumY = 0;
            for (p = 0; p < 2; p++) {
                for (q = 0; q < 2; q++) {
                    sumX += tempin[(i + p) * image->Width + j + q] * X[p][q];
                    sumY += tempin[(i + p) * image->Width + j + q] * Y[p][q];
                }
            }
            res = abs(sumX) + abs(sumY);
            if (res > 255) res = 255;
            tempout[i * image->Width + j] = res;
        }
    }

    Threshold(tempout, 0.25, size);

    return outimage;
}

```

## 2. Prewitt

```

Image* Prewitt(Image* image) {
    Image* outimage;
    int i, j, p, q, sumX, sumY, res;
    unsigned char* tempin, * tempout;
    int size = image->Height * image->Width;

    outimage = CreateNewImage(image, "Prewitt", image->Width, image->Height);
    tempout = outimage->data;
    tempin = image->data;

    int X[3][3] = { {-1, -1, -1}, {0, 0, 0}, {1, 1, 1} };
    int Y[3][3] = { {-1, 0, 1}, {-1, 0, 1}, {-1, 0, 1} };

    for (i = 1; i < image->Height - 1; i++) {
        for (j = 1; j < image->Width; j++) {
            sumX = 0;
            sumY = 0;
            for (p = -1; p < 2; p++) {
                for (q = -1; q < 2; q++) {
                    sumX += tempin[(i + p) * image->Width + j + q] * X[p + 1][q + 1];
                    sumY += tempin[(i + p) * image->Width + j + q] * Y[p + 1][q + 1];
                }
            }
            res = abs(sumX) + abs(sumY);
            if (res > 255) res = 255;
            tempout[i * image->Width + j] = res;
        }
    }

    Threshold(tempout, 0.35, size);

    return outimage;
}

```

## 3. Sobel

```

Image* Sobel(Image* image) {
    Image* outimage;
    int i, j, p, q, sumX, sumY, res;
    unsigned char* tempin, * tempout;
    int size = image->Height * image->Width;

    outimage = CreateNewImage(image, "Sobel", image->Width, image->Height);
    tempout = outimage->data;
    tempin = image->data;

    int X[3][3] = { {-1, -2, -1}, {0, 0, 0}, {1, 2, 1} };
    int Y[3][3] = { {-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1} };

    for (i = 1; i < image->Height - 1; i++) {
        for (j = 1; j < image->Width; j++) {
            sumX = 0;
            sumY = 0;
            for (p = -1; p < 2; p++) {
                for (q = -1; q < 2; q++) {
                    sumX += tempin[(i + p) * image->Width + j + q] * X[p + 1][q + 1];
                    sumY += tempin[(i + p) * image->Width + j + q] * Y[p + 1][q + 1];
                }
            }
            res = abs(sumX) + abs(sumY);
            if (res > 255) res = 255;
            tempout[i * image->Width + j] = res;
        }
    }

    Threshold(tempout, 0.35, size);

    return outimage;
}

```

## 4. Canny

```

//stepl gaussian filter
for (i = 0; i < kernel_size; i++) {
    for (j = 0; j < kernel_size; j++) {
        double e = -(double)(pow((i - kcenter - 1), 2) + pow((j - kcenter - 1), 2)) / (2 * sigma * sigma);
        gausFilter[i * kernel_size + j] = (double)1 / (2 * PI * sigma * sigma) * exp(e);
    }
}

for (i = kcenter; i < image->Height - kcenter; i++) {
    for (j = kcenter; j < image->Width - kcenter; j++) {
        sum = 0;
        for (p = 0; p < kernel_size; p++) {
            for (q = 0; q < kernel_size; q++) {
                x = i + p - kcenter;
                y = j + q - kcenter;
                sum += tempin[x * image->Width + y] * gausFilter[p * kernel_size + q];
            }
        }
        tempout[i * image->Width + j] = sum;
    }
}

```



```

//step2 Compute the gradient magnitude and angle images
double* M = (double*)malloc(sizeof(double) * size);
double* alpha = (double*)malloc(sizeof(double) * size);
memset(M, 0, sizeof(double) * size);
memset(alpha, 0, sizeof(double) * size);

for (i = 1; i < image->Height - 1; i++) {
    for (j = 1; j < image->Width; j++) {
        sumX = 0;
        sumY = 0;
        for (p = -1; p < 2; p++) {
            for (q = -1; q < 2; q++) {
                sumX += tempin[(i + p) * image->Width + j + q] * X[p + 1][q + 1];
                sumY += tempin[(i + p) * image->Width + j + q] * Y[p + 1][q + 1];
            }
        }
        if (sumX > 255) sumX = 255;
        if (sumY > 255) sumY = 255;
        M[i * image->Width + j] = sqrt(pow(sumX, 2) + pow(sumY, 2));
        alpha[i * image->Width + j] = atan2(sumY, sumX) * 180 / PI;
    }
}

```

```

for (int i = 1; i < image->Height - 1; i++) {
    for (int j = 1; j < image->Width - 1; j++) {
        double angle = alpha[i * image->Width + j];
        double ms = M[i * image->Width + j];
        if (angle > -22.5 && angle < 22.5 || angle < 157.5 && angle > -157.5) {
            if (ms < M[(i - 1) * image->Width + j] || ms < M[(i + 1) * image->Width + j]) {
                Gn[i * image->Width + j] = 0;
            }
            else {
                Gn[i * image->Width + j] = ms;
            }
        }
        else if (angle > 67.5 && angle < 112.5 || angle < -67.5 && angle > -112.5) {
            if (ms < M[i * image->Width + j - 1] || ms < M[i * image->Width + j + 1]) {
                Gn[i * image->Width + j] = 0;
            }
            else {
                Gn[i * image->Width + j] = ms;
            }
        }
        else if (angle < 157.5 && angle > 112.5 || angle > -22.5 && angle < -67.5) {
            if (ms < M[(i - 1) * image->Width + j + 1] || ms < M[(i + 1) * image->Width + j - 1]) {
                Gn[i * image->Width + j] = 0;
            }
            else {
                Gn[i * image->Width + j] = ms;
            }
        }
    }
}

```

```
else if (angle < 157.5 && angle > 112.5 || angle > -22.5 && angle < -67.5) {
    if (ms < M[(i - 1) * image->Width + j + 1] || ms < M[(i + 1) * image->Width + j - 1]) {
        Gn[i * image->Width + j] = 0;
    }
    else {
        Gn[i * image->Width + j] = ms;
    }
}
else {
    if (ms < M[(i - 1) * image->Width + j - 1] || ms < M[(i + 1) * image->Width + j + 1]) {
        Gn[i * image->Width + j] = 0;
    }
    else {
        Gn[i * image->Width + j] = ms;
    }
}
if (Gn[i * image->Width + j] > Tl) {
    Gn1[i * image->Width + j] = 1;
    temp[i * image->Width + j] = Gn[i * image->Width + j];
}
else {
    Gn1[i * image->Width + j] = 0;
}
if (Gn[i * image->Width + j] > Th) {
    GnH[i * image->Width + j] = 1;
    temp[i * image->Width + j] = Gn[i * image->Width + j];
}
}
```

```
else {
    Gn1[i * image->Width + j] = 0;
}
if (Gn[i * image->Width + j] > Th) {
    GnH[i * image->Width + j] = 1;
    temp[i * image->Width + j] = Gn[i * image->Width + j];
}
else {
    GnH[i * image->Width + j] = 0;
}
Gn1[i * image->Width + j] -= GnH[i * image->Width + j];
}
}
```

```

for (int i = 0; i < image->Height; i++) {
    for (int j = 0; j < image->Width; j++) {
        if (Gnh[i * image->Width + j] == 1) {
            if (Gnl[(i - 1) * image->Width + (j - 1)] == 1) {
                Gnl[(i - 1) * image->Width + (j - 1)] = 1.5;
            }
            if (Gnl[(i - 1) * image->Width + j] == 1) {
                Gnl[(i - 1) * image->Width + j] = 1.5;
            }
            if (Gnl[(i - 1) * image->Width + (j + 1)] == 1) {
                Gnl[(i - 1) * image->Width + (j + 1)] = 1.5;
            }
            if (Gnl[i * image->Width + (j - 1)] == 1) {
                Gnl[i * image->Width + (j - 1)] = 1.5;
            }
            if (Gnl[i * image->Width + (j + 1)] == 1) {
                Gnl[i * image->Width + (j + 1)] = 1.5;
            }
            if (Gnl[(i + 1) * image->Width + (j - 1)] == 1) {
                Gnl[(i + 1) * image->Width + (j - 1)] = 1.5;
            }
            if (Gnl[(i + 1) * image->Width + j] == 1) {
                Gnl[(i + 1) * image->Width + j] = 1.5;
            }
            if (Gnl[(i + 1) * image->Width + (j + 1)] == 1) {
                Gnl[(i + 1) * image->Width + (j + 1)] = 1.5;
            }
        }

        if (Gnh[i * image->Width + j] == 1) {
            for (p = -1; p < 2; p++) {
                for (q = -1; q < 2; q++) {
                    if (Gnl[(i + p) * image->Width + j + q] == 1) {
                        Gnl[(i + p) * image->Width + j + q] = 3;
                    }
                }
            }
        }
    }
}

```

## 5. LoG

```
Image* LoG(Image* image) {
    Image* outimage;
    int i, j, p, q, res, sumX, sumY;
    float sigma = 0.5;
    double sum;
    int kernel_size = 1 + 2 * ceil(3 * sigma);
    unsigned char* tempin, * tempout;
    int size = image->Height * image->Width;

    outimage = CreateNewImage(image, "LoG", image->Width, image->Height);
    tempout = outimage->data;
    tempin = image->data;

    for (int i = 2; i < image->Height - 2; ++i) {
        for (int j = 2; j < image->Width - 2; ++j) {
            int d = 16 * tempin[i * image->Width + j]
                - tempin[(i - 2) * image->Width + j]
                - tempin[(i - 1) * image->Width + j - 1]
                - 2 * tempin[(i - 1) * image->Width + j]
                - tempin[(i - 1) * image->Width + j + 1]
                - tempin[i * image->Width + j - 2]
                - 2 * tempin[i * image->Width + j - 2]
                - 2 * tempin[i * image->Width + j + 1]
                - tempin[i * image->Width + j + 2]
                - tempin[(i + 1) * image->Width + j - 1]
                - 2 * tempin[(i + 1) * image->Width + j]
                - tempin[(i + 1) * image->Width + j + 1]
                - tempin[(i + 2) * image->Width + j];

            if (d > 255)
                tempout[i * image->Width + j] = 255;
            else
                tempout[i * image->Width + j] = 0;
        }
    }
    return(outimage);
}
```

## 6. Global Thresholding

```
for (int i = 0; i < image->Height; i++) {
    for (int j = 0; j < image->Width; j++) {
        if (tempin[i * image->Width + j] <= T) {
            mul += tempin[i * image->Width + j];
            m++;
        }
        else {
            mu2 += tempin[i * image->Width + j];
            n++;
        }
    }
}
if (m != 0)
    mul /= m;
if (n != 0)
    mu2 /= n;
newT = (mul + mu2) / 2.0;
if (abs(T - newT) < T0) break;
else T = newT;
}

for (i = 0; i < image->Height; i++) {
    for (j = 0; j < image->Width; j++) {
        if (tempin[i * image->Width + j] > T) {
            tempout[i * image->Width + j] = 0;
        }
        else {
            tempout[i * image->Width + j] = 255;
        }
    }
}

return outimage;
```