9. Roberts, Prewitt, Sobel, Threshold, and Edge Detection

1. Obtain gradient images and threshold them

- **1.1 Roberts:**

Algorithm:

The Roberts edge operator is a 2x2 mask that uses the difference between two diagonally adjacent pixels, and then takes the sum of the squares of the differences between the two diagonal pixels. The template is showing as:

−1	0	0	−1
0	1	1	0

And the algorithms are:

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

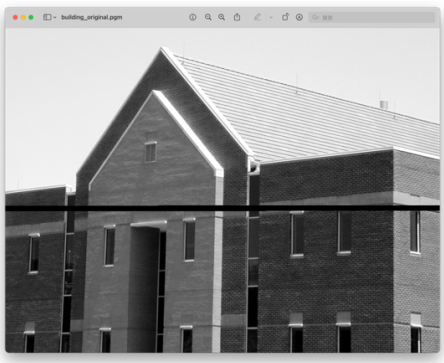
$$g_y = \frac{\partial f(x,y)}{\partial y} = f(x,y+1) - f(x,y)$$

$$p_{out} = \sqrt[2]{g_x + g_y}.$$

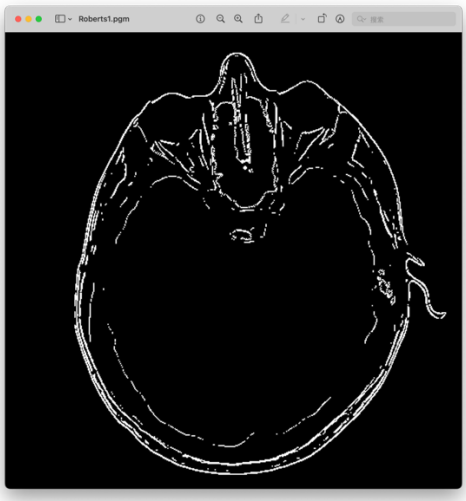
Results (including pictures):

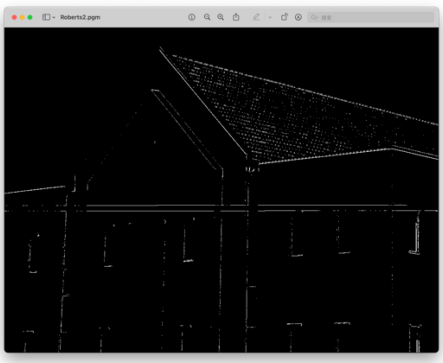
Process result of “headCT_Vandy.pgm”, “building_original.pgm”, and “noisy_fingerprint”:

Source Image:



Result of Roberts:





Discussion:

The Roberts algorithm uses the local difference operator to find the edges. When the edge in the image is close to +45 degrees or -45 degrees, the algorithm works better. The Roberts operator has high positioning accuracy for the edge. From the processing results, we found it is sensitive to noise, so it is suitable for image segmentation with obvious edges and less noise.

Codes:

```
// Algorithms Code:
void Roberts(Image *image) {
    unsigned char *tempin, *tempout;
    float temp1, temp2;
    Image *outimage;
    outimage = CreateNewImage(image, (char*)"#testing function");
    tempin = image->data;
    tempout = outimage->data;

    for(int i = 0; i < image->Height-1; i++) {
        for(int j = 0; j < image->Width-1; j++) {
            temp1 = pow((float)tempin[image->Width * i + j] - (float)tempin[image->Width * (i+1) + j + 1], 2);
            temp2 = pow((float)tempin[image->Width * i + j + 1] - (float)tempin[image->Width * (i+1) + j], 2);
            tempout[image->Width * i + j] = (int)sqrt(temp1 + temp2);
        }
    }
    outimage = Threshold(outimage);
    SavePNMImage(outimage, (char*)"Roberts.pgm");
}
```

- **1.2 Prewitt:**

Algorithm:

The Prewitt operator uses a 3×3 template to calculate the values in the region, its formula is:

$$d_y = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \qquad d_x = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

And the corresponding algorithms are:

$$G(i) = |[f(i-1,j-1) + f(i-1,j) + f(i-1,\ j+1)] - [f(i+1,j-1) + f(i+1,\ j) + f(i+1,\ j+1)]|$$

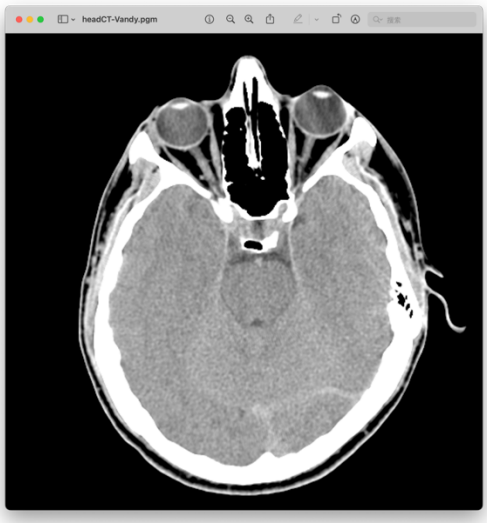
$$G(j) = |[f(i-1,j+1) + f(i,j+1) + f(i+1,\ j+1)] - [f(i-1,j-1) + f(i,j-1) + f(i+1,\ j-1)]|$$

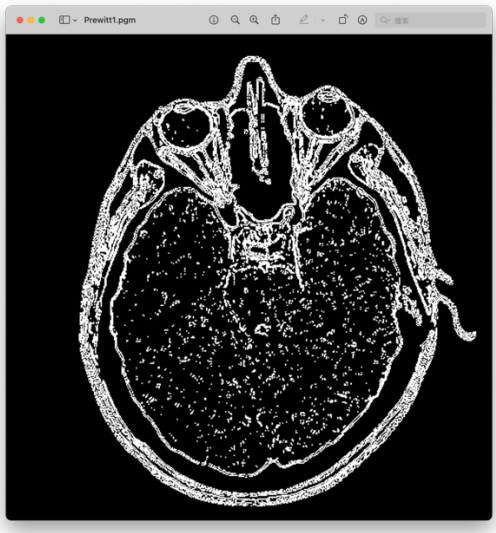
$$And\ P(i,j) = G(i) + G(j)$$

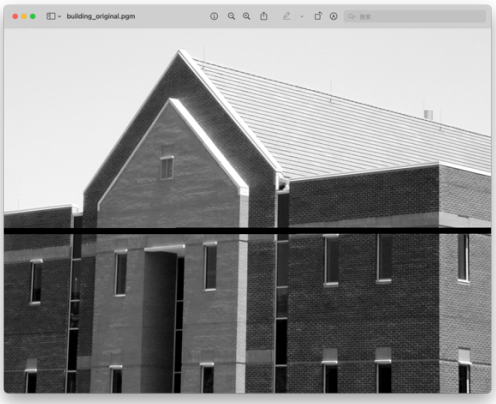
Results (including pictures):

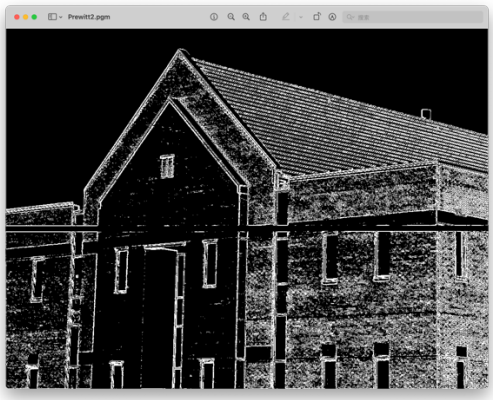
Process result of “headCT_Vandy.pgm”, “building_original.pgm”, and “noisy_fingerprint”:

Source Image:









Discussion:

Since the Prewitt operator uses a 3 × 3 template to calculate the pixel values in the region, so its edge detection results are more obvious than those of the Robert operator in both the horizontal and vertical directions than Roberts. The Prewitt operator has a suppressing effect on noise. The principle of suppressing noise is through pixel averaging, but the Prewitt operator is not as good as the Roberts operator in positioning the edge, and it may generate edges with multiple pixels width. Therefore, the Prewitt operator is suitable for identifying images with more noise and grayscale gradients.

Codes:

```
111 void Prewitt(Image *image) {
112     unsigned char *tempin, *tempout;
113     int index, square[9], temp1, temp2;
114     Image *outimage;
115     outimage = CreateNewImage(image, (char*)"#testing function");
116     tempin = image->data;
117     tempout = outimage->data;
118 
119     for(int i = 0; i < image->Height; i++) {
120         for(int j = 0; j < image->Width; j++) {
121             index = 0;
122             // record the values in the 3x3 square:
123             for(int m = -1; m <= 1; m++) {
124                 for(int n = -1; n <= 1; n++) {
125                     // use boundary check:
126                     square[index++] = boundaryCheck(j + n, i + m, image->Width, image->Height) ?
                            tempin[image->Width * (i + m) + (j + n)] : 0;
127                 }
128             }
129             temp1 = square[2] + square[5] + square[8] - square[0] - square[3] - square[6];
130             temp2 = square[0] + square[1] + square[2] - square[6] - square[7] - square[8];
131             tempout[image->Width * i + j] = abs(temp1) + abs(temp2);
132         }
133     }
134     outimage = Threshold(outimage);
135     SavePNMImage(outimage, (char*)"Prewitt.pgm");
136 }
```

- **1.3 Sobel:**

Algorithm:

Sobel operator is a first-order differential operator, and $\nabla f = grad(f) = \begin{bmatrix} g_x \\ g_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$ indicates the direction of maximum rate of change at (x, y). We use the horizontal and vertical masks are:

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \quad and \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}.$$

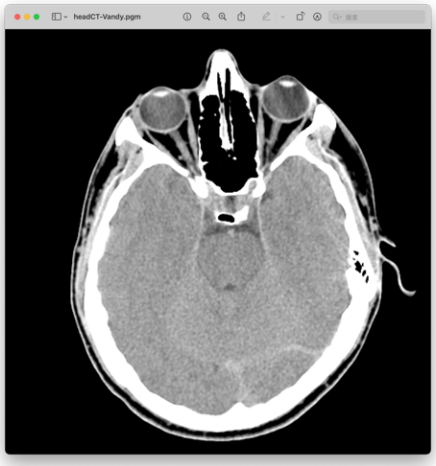
Then combine the horizontal and vertical gray values of each pixel to calculate the new grey value:

$$G = \sqrt[2]{G_x^2 + G_y^2}.$$

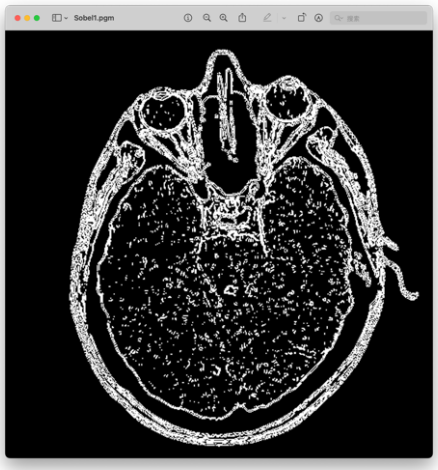
Results (including pictures):

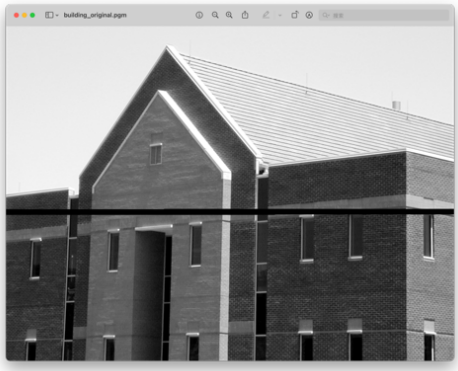
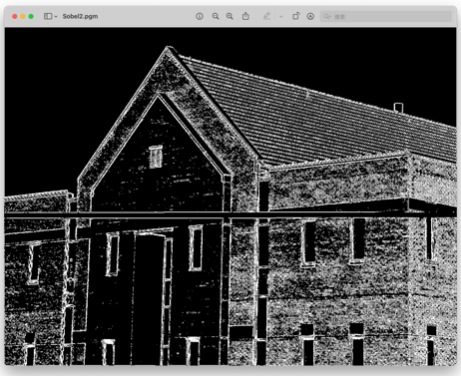
Process result of “headCT_Vandy.pgm”, “building_original.pgm”, and “noisy_fingerprint”:

Source Image:



Result of Sobel:



Discussion:

The Sobel operator detects the edge according to that the weighted difference of the intensity of the upper and lower, left and right neighbors of the pixel reaches the extreme value at the edge. Because the Sobel operator combines Gaussian smoothing and differential derivation, its result is more noise resistant. And since the it weights the influence of the position of the pixel, Sobel has better edge direction information than Prewitt and Roberts.

Codes:

```
83  void Sobel(Image *image) {
84      unsigned char *tempin, *tempout;
85      int index, square[9];
86      float temp1, temp2;
87      Image *outimage;
88      outimage = CreateNewImage(image, (char*)"#testing function");
89      tempin = image->data;
90      tempout = outimage->data;
91
92      for(int i = 0; i < image->Height; i++) {
93          for(int j = 0; j < image->Width; j++) {
94              index = 0;
95              // record the values in the 3x3 square:
96              for(int m = -1; m <= 1; m++) {
97                  for(int n = -1; n <= 1; n++) {
98                      // use boundary check:
99                      square[index++] = boundaryCheck(j + n, i + m, image->Width, image->Height) ?
                            tempin[image->Width * (i + m) + (j + n)] : 0;
100                 }
101             }
102             temp1 = abs((float)square[2] + (float)square[5]*2 + (float)square[8] - (float)square[0] -
                    (float)square[3]*2 - (float)square[6]);
103             temp2 = abs((float)square[6] + (float)square[7]*2 + (float)square[8] - (float)square[0] -
                    (float)square[1]*2 - (float)square[2]);
104             tempout[image->Width * i + j] = (int)sqrt(pow(temp1, 2) + pow(temp2, 2));
105         }
106     }
107     outimage = Threshold(outimage);
108     SavePNMImage(outimage, (char*)"Sobel.pgm");
109 }
```

- **1.4 Thresholding:**

Algorithm:

It goes through all the pixels in the image and obtains the maximum intensity, and multiply by a factor$(33\%\ in\ my\ case)$ to get the threshold we need. And if the original pixels' intensity is larger than the threshold, we set it to white(255), otherwise it will be discarded(0).

***The results of thresholding have been included in the above 3 algorithms.**

Code:

```
Image *Threshold(Image *image) {
    unsigned char *tempin, *tempout;
    Image *outimage;
    int size = image->Width * image->Height, max = 0;
    outimage = CreateNewImage(image, (char*)"#testing function");
    tempin = image->data;
    tempout = outimage->data;

    for(int i = 0; i < size; i++) {
        if(tempin[i] > max) max = tempin[i];
    }
    int threshold = round((float)max * 0.33);

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

2. Edge detection algorithms (headCT_Vandy, noisy_fingerprint):

- **2.1 Canny:**

Algorithm:

The algorithm of Canny has four main steps:

(1) **Gaussian filter**: it is to reduce the noise. The computation of derivatives is sensitive to noise, so filters must be used to improve the performance of noise-dependent edge detectors. The algorithm and the template I use here are:

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

$$\frac{1}{273} * \begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

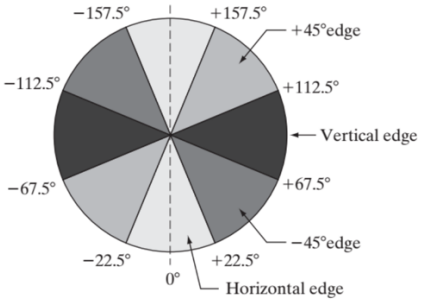
(2) **Calculate gradient magnitude and direction**: using **Sobel Operator**. And the formulas of gradient magnitude and direction are:

$$G(m,n) = \sqrt{g_x(m,n)^2 + g_y(m,n)^2}$$

$$\theta = arctan\frac{g_y(m,n)}{g_x(m,n)}$$

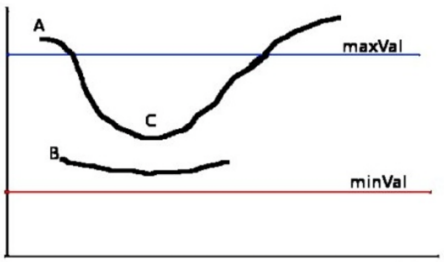
where g_x, g_y are gradient values in different directions.

(3) **Filter Non-maximum**: if a pixel belongs to an edge, then the gradient value of this pixel in the gradient direction is the largest. Otherwise, it is not an edge, and its value should be set to 0. There are 4 directions in total, and the center one should be the maximum value:



where the direction is determined by the angel obtained last step.

(4) **Detection with double threshold algorithm:** Any pixels greater than $maxVal$ is detected as an edge, and anyone below $minval$ is detected as a non-edge. For a pixel in the middle, if it is adjacent to a pixel determined to be an edge, it is determined to be an edge, otherwise it is a non-edge.

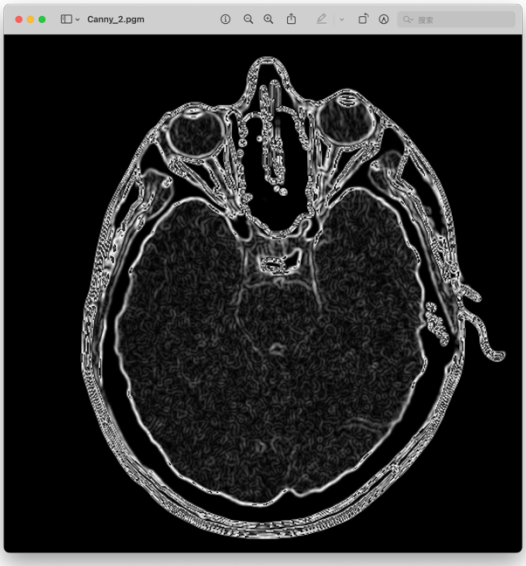


Results (including pictures):

Process result of "headCT_Vandy.pgm" and "noisy_fingerprint.pgm":

Source Image:

Result after Gaussian and Sobel **(step 2)**:



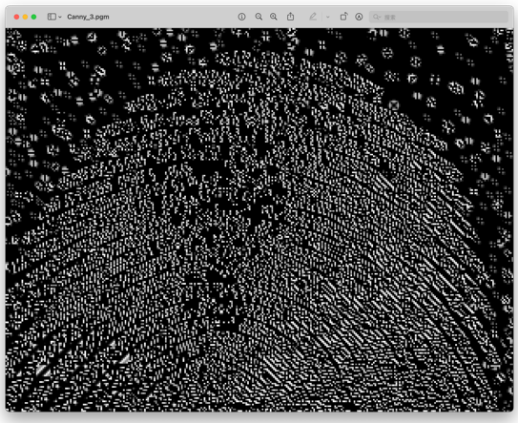
Result after non-maximum filter **(step 3)**:

Final result after Canny **(step 4)**:

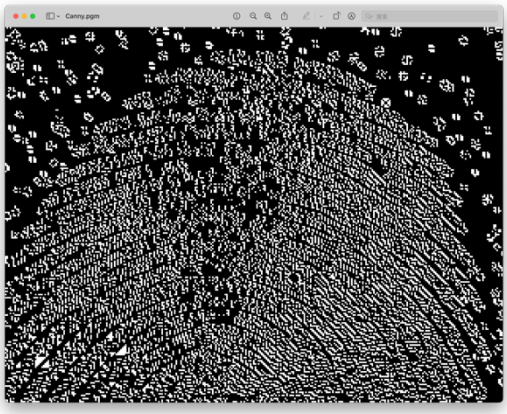
Source Image:

Result after Gaussian and Sobel **(step 2)**:

Result after non-maximum filter **(step 3)**:



Final result after Canny **(step 4)**:



Discussion:

The Canny is a very excellent edge detection algorithm overall. It finds almost all edges and no false edges, and the detected edge points are all single. The positional deviations of edge points

at different scales are almost the same. It has some noise immunity, but it can't deal with the level of "noise_fingerprint". Moreover, the Canny operator can better detect the weak edges.

Codes:

(1) Step 1: Gaussian Filter

```
199      // Step 1: Gaussian Blur:
200      image = Gaussian(image);
201      tempin = image->data;
202      SavePNMImage(image, (char*)"Gaussian.pgm");


345  Image *Gaussian(Image *image) {
346      unsigned char *tempin, *tempout;
347      Image *outimage;
348      int index, mask[25];
349      outimage = CreateNewImage(image, (char*)"#testing function");
350      tempin = image->data;
351      tempout = outimage->data;
352
353      for(int i = 0; i < image->Height; i++) {
354          for(int j = 0; j < image->Width; j++) {
355              index = 0;
356              // record the values in the 5x5 square:
357              for(int m = -2; m <= 2; m++) {
358                  for(int n = -2; n <= 2; n++) {
359                      // use boundary check:
360                      mask[index++] = boundaryCheck(j + n, i + m, image->Width, image->Height) ?
                             tempin[image->Width * (i + m) + (j + n)] : 0;
361                  }
362              }
363              float temp = 41*mask[12] + 16*(mask[6]+mask[8]+mask[16]+mask[18]) +
                     26*(mask[7]+mask[11]+mask[13]+mask[17]) + 7*(mask[2]+mask[10]+mask[14]+mask[22]) +
                     4*(mask[1]+mask[3]+mask[5]+mask[9]+mask[15]+mask[19]+mask[21]+mask[23]) +
                     (mask[0]+mask[4]+mask[20]+mask[24]);
364              tempout[image->Width * i + j] = (int)(temp / 273.0);
365          }
366      }
367      return(outimage);
368  }
```

(2) Step 2: Sobel Operator and obtain the directions

```
204      // Step 2: Sobel
205      for(int i = 0; i < image->Height; i++) {
206          for(int j = 0; j < image->Width; j++) {
207              index = 0;
208              // record the values in the 3x3 square:
209              for(int m = -1; m <= 1; m++) {
210                  for(int n = -1; n <= 1; n++) {
211                      // use boundary check:
212                      square[index++] = boundaryCheck(j + n, i + m, image->Width, image->Height) ?
                             tempin[image->Width * (i + m) + (j + n)] : 0;
213                  }
214              }
215              temp1 = abs((float)square[2] + (float)square[5]*2 + (float)square[8] - (float)square[0] -
                     (float)square[3]*2 - (float)square[6]);
216              temp2 = abs((float)square[6] + (float)square[7]*2 + (float)square[8] - (float)square[0] -
                     (float)square[1]*2 - (float)square[2]);
217              tempout[image->Width * i + j] = (int)sqrt(pow(temp1, 2) + pow(temp2, 2));
218
219              if(temp1 == 0.0 || temp2 == 0.0) angel[image->Width * i + j] = 0;
220              else {
221                  float theta = atan2(temp1, temp2) * (180.0 / pi);
222                  if((theta <= 22.5 && theta >= -22.5) || (theta <= -157.5) || (theta >= 157.5))
                         angel[image->Width * i + j] = 1; // "-"
223                  else if((theta > 22.5 && theta <= 67.5) || (theta > -157.5 && theta <= -112.5))
                         angel[image->Width * i + j] = 2; // "/"
224                  else if((theta > 67.5 && theta <= 112.5) || (theta >= -112.5 && theta < -67.5))
                         angel[image->Width * i + j] = 3; // "|"
225                  else if((theta >= -67.5 && theta < -22.5) || (theta > 112.5 && theta < 157.5))
                         angel[image->Width * i + j] = 4; // "\"
226              }
227          }
228      }
229      SavePNMImage(outimage, (char*)"Canny_2.pgm");
```

(3) Step 3: Non-maximum suppression

```
    // Step 3: Non-Maximum Supression:
    for(int i = 0; i < image->Height; i++) {
        for(int j = 0; j < image->Width; j++) {
            int currPosition = image->Width * i + j;
            if(angel[currPosition] == 0) tempout1[currPosition] = 0;
            else if(angel[currPosition] == 1) {
                if(tempout[currPosition] >= tempout[currPosition-1] && tempout[currPosition] >=
                    tempout[currPosition+1]) tempout1[currPosition] = tempout[currPosition];
                else tempout1[currPosition] = 0;
            }
            else if(angel[currPosition] == 2) {
                if(tempout[currPosition] >= tempout[image->Width * (i-1) + j + 1] && tempout[currPosition] >=
                    tempout[image->Width * (i+1) + j - 1]) tempout1[currPosition] = tempout[currPosition];
                else tempout1[currPosition] = 0;
            }
            else if(angel[currPosition] == 3) {
                if(tempout[currPosition] >= tempout[image->Width * (i-1) + j] && tempout[currPosition] >=
                    tempout[image->Width * (i+1) + j]) tempout1[currPosition] = tempout[currPosition];
                else tempout1[currPosition] = 0;
            }
            else if(angel[currPosition] == 4) {
                if(tempout[currPosition] >= tempout[image->Width * (i-1) + j - 1] && tempout[currPosition] >=
                    tempout[image->Width * (i+1) + j + 1]) tempout1[currPosition] = tempout[currPosition];
                else tempout1[currPosition] = 0;
            }
        }
    }
    SavePNMImage(outimage1, (char*)"Canny_3.pgm");
```

(4) Step 4: Detection with double threshold algorithm

```
    // Step 4: Detection with double threshold algorithm:
    int label_table[image->Width * image->Height], label_update[image->Width * image->Height];
    for(int i = 0; i < image->Height; i++) {
        for(int j = 0; j < image->Width; j++) {
            int currPosition = image->Width * i + j;
            if(tempout1[currPosition] > 120) {
                label_table[currPosition] = 1;
                label_update[currPosition] = 1;
            }
            else if(tempout1[currPosition] < 50) {
                label_table[currPosition] = 0;
                label_update[currPosition] = 0;
            }
            else {
                label_table[currPosition] = 2;
                label_update[currPosition] = 2;
            }
        }
    }

    for(int i = 0; i < image->Height; i++) {
        for(int j = 0; j < image->Width; j++) {
            int currPosition = image->Width * i + j;

            if(label_table[currPosition] == 2) {
                int sum = 0;
                for(int m = -1; m <= 1; m++) {
                    for(int n = -1; n <= 1; n++) {
                        sum += label_table[image->Width * (i + m) + (j + n)];
                        if(label_table[image->Width * (i + m) + (j + n)] == 1) {
                            label_update[currPosition] = 1;
                        }
                    }
                }
                if(sum == 2) label_update[currPosition] = 0;
            }
        }
    }
    // output the result image:
    for(int i = 0; i < image->Height; i++) {
        for(int j = 0; j < image->Width; j++) {
            int currPosition = image->Width * i + j;
            if(label_update[currPosition] == 1) tempout2[currPosition] = 255;
            else tempout2[currPosition] = 0;
        }
    }
    SavePNMImage(outimage2, (char*)"Canny.pgm");
}
```

- **2.2 LoG:**

Algorithm:

The algorithm of LoG has three main steps:

(1) Gaussian filter: Same as in **Canny**:

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

$$\frac{1}{273} * \begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

(2) Calculate the Laplace:

$$\mathrm{Laplace(f)} = \frac{\partial^2 \mathrm{f}}{\partial \mathrm{x}^2} + \frac{\partial^2 \mathrm{f}}{\partial \mathrm{y}^2}.$$

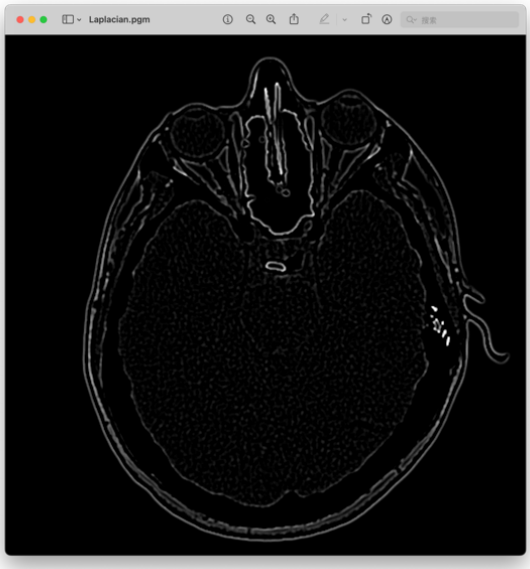
The 3x3 mask template we use is that $\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$

(3) Find the Zero Crossings of the resulting image from step 2. A 3*3 mask centered on P, with four cases to test: left/right, up/down, and two opposite corners. Then the absolute value of the difference between at least a pair of values exceeds the set threshold, while P itself has a high enough intensity, then P would be a zero crossing point.

Results (including pictures):

Process result of "licoln.pgm" and "U.pgm":

Image after Gaussian and Laplace **(step 2)**:



Result after LoG **(step 3)**:

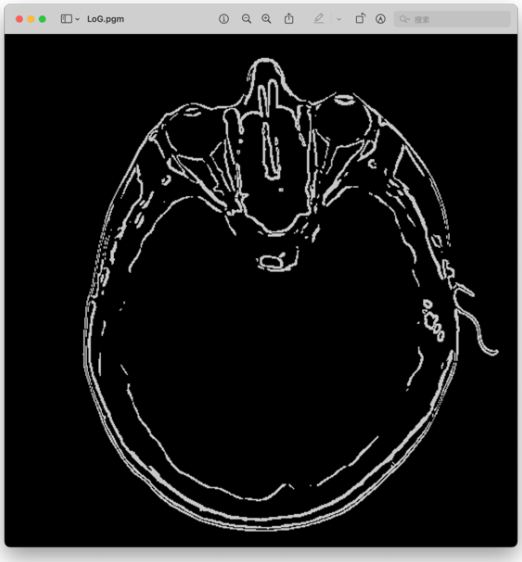
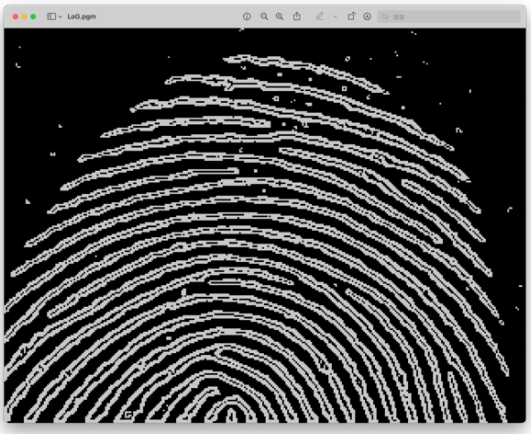


Image after Gaussian and Laplace **(step 2)**:

Result after LoG **(step 3)**:



Discussion:

Edges at different scales should be detected using LoG operators of different scales. The LoG operator is more accurate than the edge point of the Canny operator. Its disadvantage is that it may produce false edges due to the noise, and the positioning error of some curved edges may be large. From the perspective of sensitivity to noise, the LoG edge detector uses the second derivative zero-crossing detection method, so it is more sensitive to noise, but smoothing it through a Gaussian filter will greatly improve the effect.

Codes:

```
305 void LoG(Image *image) {
306     unsigned char *tempin, *tempout, *tempout1;
307     Image *outimage, *outimage1;
308     int index, square[9];
309 
310     outimage = CreateNewImage(image, (char*)"#testing function");
311     outimage1 = CreateNewImage(image, (char*)"#testing function");
312     tempout1 = outimage1->data;
313 
314     // Step 1: Gaussian Blur:
315     image = Gaussian(image);
316     tempin = image->data;
317 
318     // Step 2: Laplacian:
319     outimage = Laplacian(image);
320     tempout = outimage->data;
321     SavePNMImage(outimage, (char*)"Laplacian.pgm");
322 
323     // Step 3: Find zero crossings:
324     for(int i = 0; i < image->Height; i++) {
325         for(int j = 0; j < image->Width; j++) {
326             index = 0;
327             // record the values in the 3x3 square:
328             for(int m = -1; m <= 1; m++) {
329                 for(int n = -1; n <= 1; n++) {
330                     // use boundary check:
331                     square[index++] = boundaryCheck(j + n, i + m, image->Width, image->Height) ?
                            tempin[image->Width * (i + m) + (j + n)] : 0;
332                 }
333             }
334             if(square[4] > 80 && (abs(square[8]-square[0]) > 60 || abs(square[7]-square[1]) > 60 ||
                    abs(square[6]-square[2]) > 60 || abs(square[5]-square[3]) > 60))  {
335                 tempout1[image->Width * i + j] = 200;
336             }
337             else tempout1[image->Width * i + j] = 0;
```

```
339             }
340         }
341         SavePNMImage(outimage1, (char*)"LoG.pgm");
342     }
```

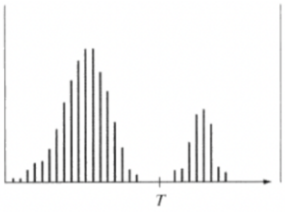
Laplacian Operator:

```
158 Image *Laplacian(Image *image) {
159     unsigned char *tempin, *tempout;
160     int sum = 0;
161     Image *outimage;
162     outimage = CreateNewImage(image, (char*)"#testing function");
163     tempin = image->data;
164     tempout = outimage->data;
165 
166     for(int i = 0; i < image->Height; i++) {
167         for(int j = 0; j < image->Width; j++) {
168             sum = 0;
169 
170             for(int m = -1; m <= 1; m++) {
171                 for(int n = -1; n <= 1; n++) {
172                     // use boundary check:
173                     sum += boundaryCheck(j + n, i + m, image->Width, image->Height) ? tempin[image->Width * (i
                            + m) + (j + n)] : 0;
174                 }
175             }
176             int temp = sum - 9 * tempin[image->Width * i + j];
177             // handle excess values:
178             if(temp > 255) temp = 255;
179             if(temp < 0) temp = 0;
180             tempout[image->Width * i + j] = temp;
181         }
182     }
183     return (outimage);
184 }
```

3. Global thresholding (polymersomes, noisy_fingerprint):

Algorithm:

The idea of global thresholding is that we get a value T to separate the pixels in the image into two parts.



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

The values those larger than T will be assigned to 255, otherwise they will be discarded.

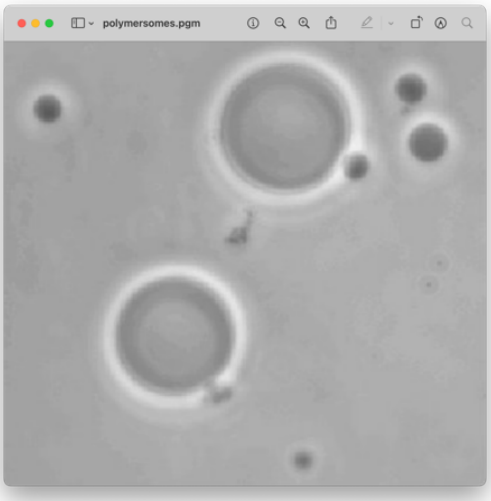
The steps to find a proper T is that:

1. *Choose an initial estimate for the global threshold* T
2. *Divide the image by* T*, resulting in two sets of pixels:* $G1$ *consists of pixels larger than* T*,* $G2$ *consists of pixels smaller than* T
3. *Calculate the average intensity* $m1$ *and* $m2$ *for the pixels of* $G1$ *and* $G2$ *respectively*
4. *Calculate the new threshold* $T = 1/2 * (m1 + m2)$
5. *Repeat steps 2-4 a certain number of times*

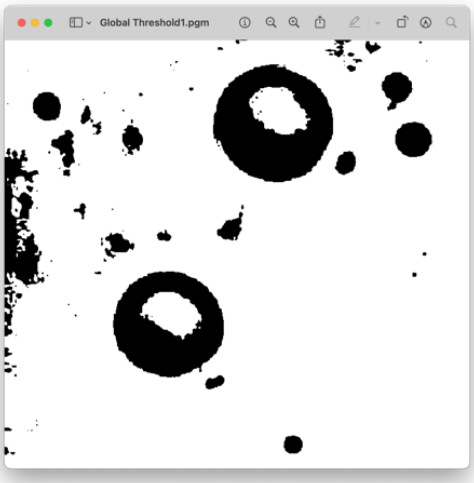
Results (including pictures):

Process result of “polymersomes”, and “noisy_fingerprint”:

Source Image:



Result after global thresholding:



Source image:

Result after global thresholding:

Discussion:

The global thresholding uses the threshold T as the segmentation point to extract the brighter objects under the dark background in the image. This algorithm works well when there is a fairly sharp valley between the histograms of the foreground and background of the image. However, it totally does not work on the binary image such as “noisy_fingerprint”, since all the white components value are larger than the threshold so there will be no difference after processing.

Codes:

```
void GlobalThreshold(Image *image) {
    unsigned char *tempin, *tempout;
    Image *outimage;
    float T = 128; // provide an initial T
    int size = image->Width * image->Height, times = 0;
    outimage = CreateNewImage(image, (char*)"#testing function");
    tempin = image->data;
    tempout = outimage->data;

    while(times < 30) {
        float count1 = 0, count2 = 0, sum1 = 0, sum2 = 0;
        times++;

        for(int i = 0; i < size; i++) {
            if(tempin[i] >= T) {
                sum2 += tempin[i];
                count2++;
            }
            else {
                sum1 += tempin[i];
                count1++;
            }
        }
        float average1 = sum1 / count1;
        float average2 = sum2 / count2;
        T = (average1 + average2) / 2.0;// update the T1
    }
    // output the image:
    for(int i = 0; i < size; i++) {
        if(tempin[i] >= T) tempout[i] = 255;
        else tempout[i] = 0;
    }
    SavePNMImage(outimage, (char *)"Global Threshold.pgm");
}
```