# 4. Image Sharpen, Gamma Correction, and Histogram Enhancement

## 1. Image sharpening (lena, goldhill):

- Laplacian operator:

**Algorithm:**

Laplace operator is a second-order differential operator, and use the following formula:

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

In a two-dimensional function $f(x, y)$, the second-order differences in the two directions of $x$ and $y$ are respectively:

$$\frac{\partial^2 f}{\partial x^2} = 2f(x, y) - f(x + 1, y) - f(x - 1, y),$$
$$\frac{\partial^2 f}{\partial y^2} = 2f(x, y) - f(x, y + 1) - f(x, y - 1).$$

We get the difference form of the Laplace operator:

$$\nabla^2 f(x, y) = 4f(x, y) - f(x + 1, y) - f(x - 1, y) - f(x, y + 1) - f(x, y - 1).$$

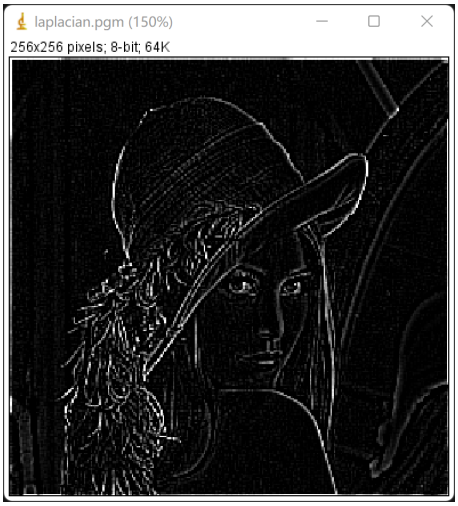
So we can use the 3x3 coefficient mask as: $g = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$ to implement.

**Results (including pictures):**

Process result of "lena.pgm":

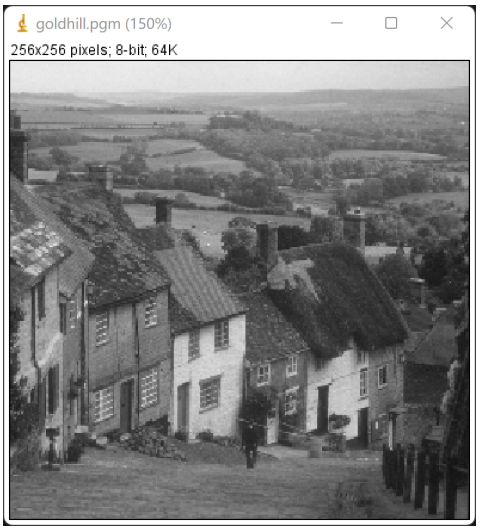
Source Image:


Result after sharpened with Laplacian operator:

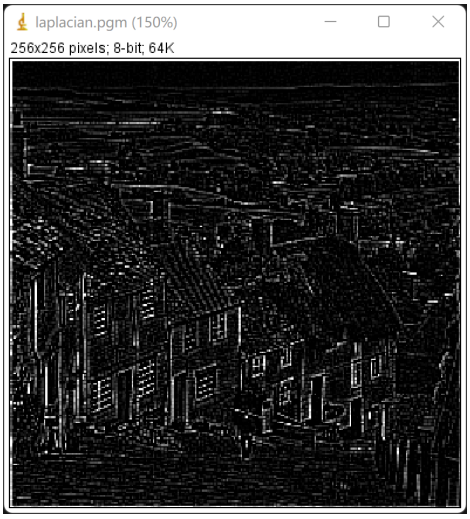


Process result of “goldhill.pgm”

Source Image:




Result after sharpened with Laplacian operator:




**Discussion:**

The Laplacian operator achieves the effect of sharpening the image by enhancing the grayscale contrast. Since it is a second-order differential operator that enhances the areas of sudden grayscale changes in the image and weakens the slowly changing areas of grayscale. But after processing the image loses the direction information of the edge and enhances the effect of noise.

**Codes:**

```
57  // Algorithms Code:
58  Image *Laplacian(Image *image) {
59      unsigned char *tempin, *tempout;
60      int sum = 0;
61      Image *outimage;
62      outimage = CreateNewImage(image, (char*)"#testing function");
63      tempin = image->data;
64      tempout = outimage->data;
65
66      for(int i = 0; i < image->Height; i++) {
67          for(int j = 0; j < image->Width; j++) {
68              sum = 0;
69              for(int m = -1; m <= 1; m += 2) {
70                  for(int n = -1; n <= 1; n += 2) {
71                      // use boundary check:
72                      sum += boundaryCheck(j + n, i + m, image->Width, image->Height) ?
                            tempin[image->Width * (i + m) + (j + n)] : 0;
73                  }
74              }
75              int temp = tempin[image->Width * i + j] * 4 - sum;
76              // handle excess values:
77              if(temp > 255) temp = 255;
78              if(temp < 0) temp = 0;
79              tempout[image->Width * i + j] = temp;
80          }
81      }
82      return (outimage);
83  }

214 int boundaryCheck(int index_x, int index_y, int width, int height) {
215     if(index_x >= 0 && index_y >= 0 && index_x < width && index_y < height) return 1;
216     else return 0;
217 }
```

- Sobel operator:

**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):**

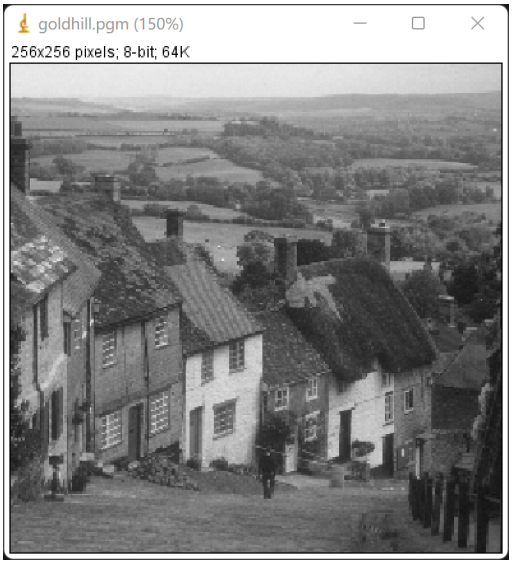
Process result of "lena.pgm":

Source Image:

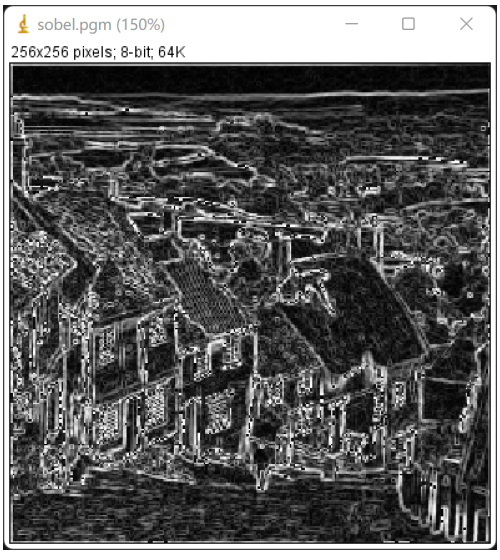

Result after sharpened with Sobel operator:


Process result of "goldhill.pgm":

Source Image:




Result after sharpened with Sobel operator:



**Discussion:**

The Sobel operator is a first-order differential edge detection operator, which introduces an operation similar to the local average, so it has a smoothing effect on noise while strengthening the edges of the image objects.

After differentiation, the value of the flat place is almost 0, while the absolute value of the edge place is very large. But the Sobel operator does not strictly distinguish the main body of the image from the background, so the extracted image contours are sometimes unsatisfactory.

**Codes:**

```
Image *Sobel(Image *image) {
    unsigned char *tempin, *tempout;
    int index, square[9], temp1, temp2;
    Image *outimage;
    outimage = CreateNewImage(image, (char*)"#testing function");
    tempin = image->data;
    tempout = outimage->data;

    for(int i = 0; i < image->Height; i++) {
        for(int j = 0; j < image->Width; j++) {
            index = 0;
            // record the values in the 3x3 square:
            for(int m = -1; m <= 1; m++) {
                for(int n = -1; n <= 1; n++) {
                    // use boundary check:
                    square[index++] = boundaryCheck(j + n, i + m, image->Width, image->Height) ?
                        tempin[image->Width * (i + m) + (j + n)] : 0;
                }
            }
            temp1 = abs(square[2] + 2*square[5] + square[8] - square[0] - 2*square[3] - square[6]);
            temp2 = abs(square[6] + 2*square[7] + square[8] - square[0] - 2*square[1] - square[2]);
            tempout[image->Width * i + j] = sqrt(pow(temp1, 2) + pow(temp2, 2));
        }
    }
    return (outimage);
}
```

## 2. Gamma correction(lena, goldhill):

**Algorithm:**

First process the pixels using **normalization**: convert the pixel value to a real number between $0\ and\ 1$. The algorithm is: $(f + 0.5)/256$, where $f$ is the original pixel value.

Then use the formula of gamma correction: $\boldsymbol{s = cr^{\gamma}}$, where $c$ and $\gamma$ are constants.

Finally, **denormalize** the corrected values: inverse transform the compensated real values to integer values between $0$ and $255$. The algorithm is $s * 256 - 0.5$.

**Results (including pictures):**

Process result of “lena.pgm”:

Source Image:


Gamma correction using value 0.1:


Gamma correction using value 0.4:

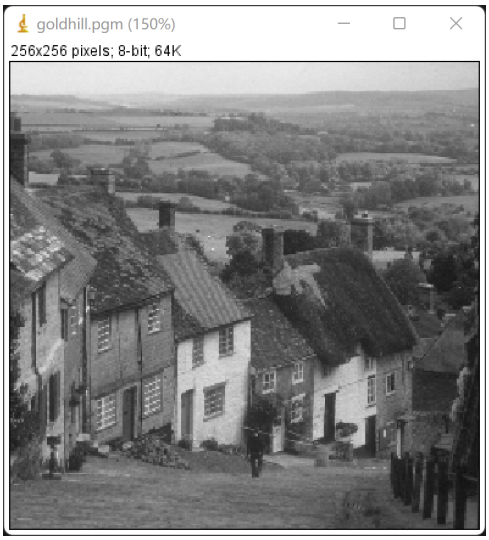

Gamma correction using value 0.7:


Gamma correction using value 1.0:

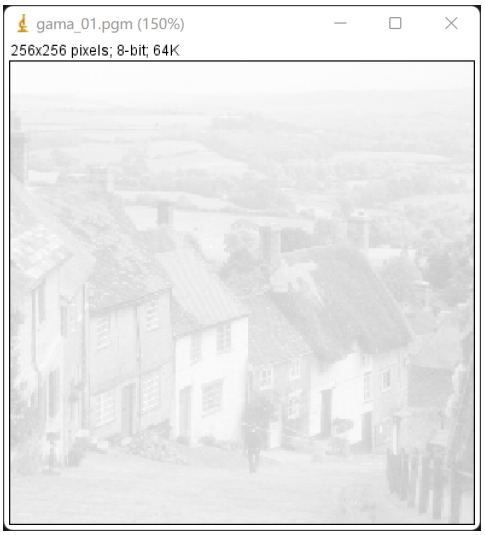

Gamma correction using value 1.3:

Process result of “goldhill.pgm”:

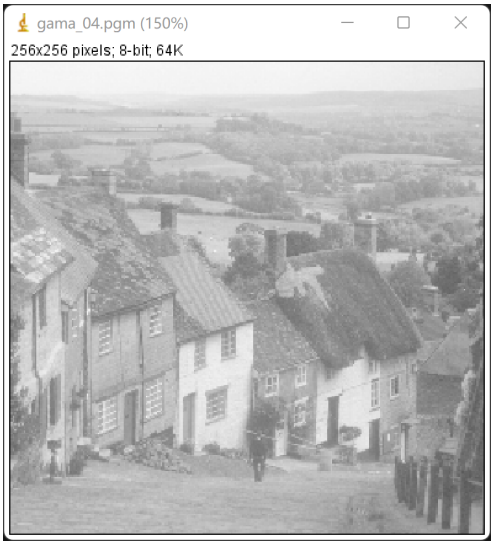
Source Image:



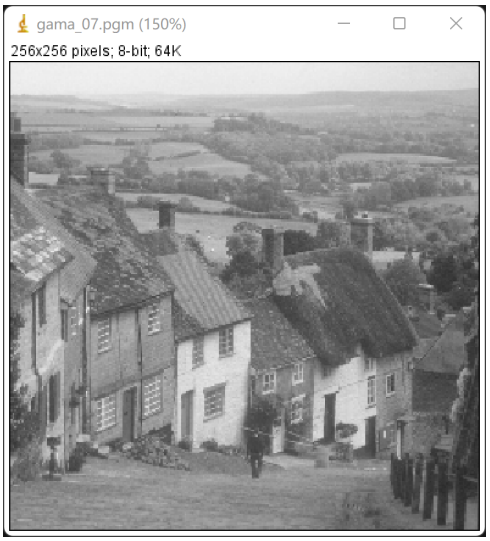

Gamma correction using value 0.1:



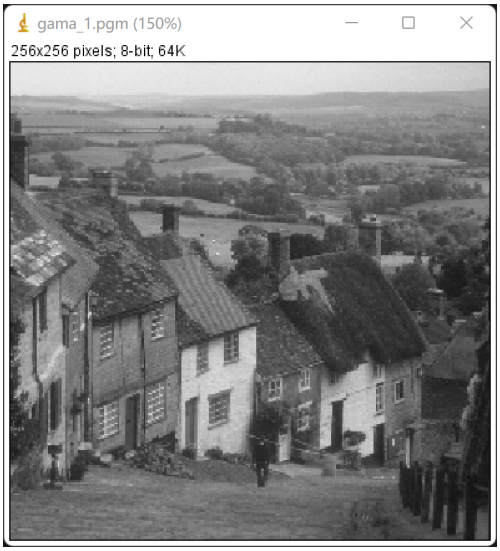

Gamma correction using value 0.4:



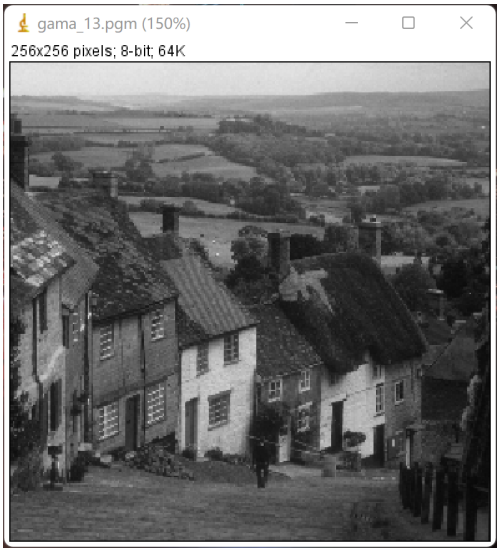

Gamma correction using value 0.7:




Gamma correction using value 1.0:




Gamma correction using value 1.3:



The variance results:

```
Loading goldhill.pgm .....Image Type PGM
The variance of gamma value 0.1 is: 115.19
The variance of gamma value 0.4 is: 1040.28
The variance of gamma value 0.7 is: 1896.06
The variance of gamma value 1.0 is: 2408.00
Saving Image laplacian.pgm
```

**Discussion:**

Gamma correction is a method of non-linear color editing that changes the ratio of dark and light parts of an image, thereby altering the effect of image contrast. When $\gamma < 1$, the contrast is decreased, and when $\gamma > 1$, the contrast is increased. As can be seen from the results, the lower the $\gamma$, the lower the contrast of the image and the smaller the variance of the intensity values in the image.

**Codes:**

```
111 Image *Gamma(Image *image, float ratio) {
112     unsigned char *tempin, *tempout;
113     float temp;
114     float variance, average, sum = 0, N = image->Width * image->Height; // calculate variance
115 
116     Image *outimage;
117     outimage = CreateNewImage(image, (char*)"#testing function");
118     tempin = image->data;
119     tempout = outimage->data;
120 
121     for(int i = 0; i < image->Height; i++) {
122         for(int j = 0; j < image->Width; j++) {
123             temp = ((float)tempin[image->Width * i + j] + 0.5) / 256; // normalized
124             temp = pow(temp, ratio); // power the parameter
125             temp = (int)(temp * 256 - 0.5); // denormalization
126             tempout[outimage->Width * i + j] = (unsigned char)temp;
127             sum += temp;
128         }
129     }
130 
131     // calculate & output the variance:
132     average = sum / N;
133     sum = 0;
134     for(int i = 0; i < image->Height; i++) {
135         for(int j = 0; j < image->Width; j++) {
136             sum += pow(tempout[outimage->Width * i + j] - average, 2);
137         }
138     }
139     variance = sum / N;
140     printf("The variance of gamma value %.1f is: %.2f\n", ratio, variance);
141     return (outimage);
142 }
```

## 3. Histogram enhancement(lena, goldhill):

- Global histogram enhancement

**Algorithm:**

We define $r_k$ is the $k^{th}$ intensity value of the image, $n_k$ is the number of pixels with value $r_k$. And $s_k$ is the value of the output pixels, and $p_k$ will be the pixel value after normalized.

So the algorithm of global histogram enhancement is:

$$s_k = \left(\sum_{i=0}^{k} \frac{n_i}{MN}\right)(L-1) = \left(\sum_{i=0}^{k} p(r_k)\right)(L-1),$$

$(M * N \ is \ the \ image \ size \ \ and \ \ L = 256 \ in \ the \ grey \ image).$

**Results (including pictures):**

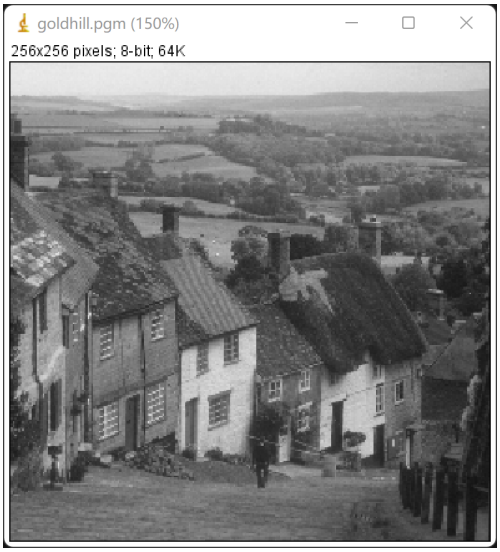
Process result of "lena.pgm":

Source Image:

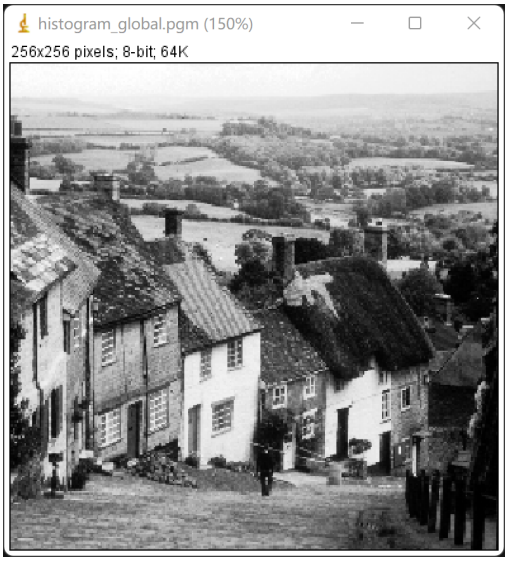

Result after global histogram enhancement:


Process result of "goldhill.pgm":

Source Image:




Result after global histogram enhancement:




**Discussion:**

The effect of global histogram enhancement is that the pixels of the image tend to occupy the entire possible gray level and are evenly distributed, so that the gray details of the image are rich and the dynamic range is large. It can be seen from the results that the image contrast is significantly enhanced after processing.

**Codes:**

```
144 Image *Global_histogram(Image *image) {
145     unsigned char *tempin, *tempout, temp;
146     int histogram_sum[256], histogram[256], currSum = 0; // used for statistics
147     float constant = (float)255 / (float)(image->Width * image->Height);// (L-1)/(M*N)
148 
149     Image *outimage;
150     outimage = CreateNewImage(image, (char*)"#testing function");
151     tempin = image->data;
152     tempout = outimage->data;
153 
154     // initialize the array:
155     for(int i = 0; i < 256; i++) histogram[i] = 0;
156 
157     for(int i = 0; i < image->Height; i++) {
158         for(int j = 0; j < image->Width; j++) {
159             temp = tempin[image->Width * i + j];
160             histogram[temp] += 1;
161         }
162     }
163     for(int i = 0; i < 256; i++) {
164         currSum += histogram[i];
165         histogram_sum[i] = currSum;
166     }
167 
168     // output the image:
169     for(int i = 0; i < image->Height; i++) {
170         for(int j = 0; j < image->Width; j++) {
171             temp = tempin[image->Width * i + j];
172             tempout[outimage->Width * i + j] = (int)(histogram_sum[temp] * constant);
173         }
174     }
175     return (outimage);
176 }
```

- Local histogram enhancement

**Algorithm:**

The principle of local histogram enhancement is similar to the global one, but it will recalculate $n_k$(the number of pixels with value $r_k$) in the 3x3 regions for every pixel in the image.

So the algorithm of local histogram enhancement is:

$$s_k = \left(\sum_{i=0}^{k} \frac{n_i}{3*3}\right)(L-1) = \left(\sum_{x=-1}^{1}\sum_{y=-1}^{1} n(i_x + x, i_y + y)\right)\left(\frac{L-1}{9}\right).$$

**Results (including pictures):**

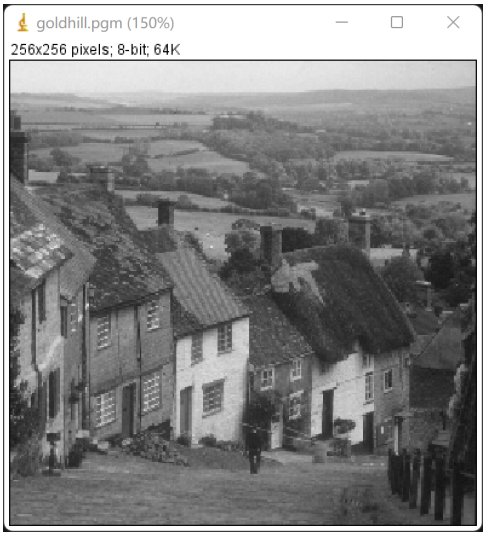
Process result of “lena.pgm”:

Source Image:


Result after local histogram enhancement:

Process result of “goldhill.pgm”:

Source Image:




Result after local histogram enhancement:


**Discussion:**

Local histogram enhancement is used for local contrast enhancement of the image. It needs a mask and moves through all the pixels to change the values at the center of the mask.

I use overlapping mask to implement the algorithm, because this not only allows sufficient contrast enhancement of local details in the image, but also eliminates blocking artifacts. However, since the total number of sub-block equalization times is equal to the total number of pixels in the image, the algorithm requires more computation.

**Codes:**

```
178 Image *Local_histogram(Image *image) {
179     unsigned char *tempin, *tempout, temp;
180     int histogram_sum[256], histogram[256]; // used for statistics
181     float constant = (float)255 / (float)9; // M*N changed to 9
182 
183     Image *outimage;
184     outimage = CreateNewImage(image, (char*)"#testing function");
185     tempin = image->data;
186     tempout = outimage->data;
187 
188     // process all the pixels:
189     for(int i = 0; i < image->Height; i++) {
190         for(int j = 0; j < image->Width; j++) {
191             // initialize the array:
192             for(int k = 0; k < 256; k++) histogram[k] = 0;
193 
194             for(int x = -1; x <= 1; x++) {
195                 for(int y = -1; y <= 1; y++) {
196                     // use boundary check:
197                     temp = boundaryCheck(j + y, i + x, image->Width, image->Height) ?
                            tempin[image->Width * (i + x) + (j + y)] : 0;
198                     histogram[temp] += 1;
199                 }
200             }
201             for(int k = 0, currSum = 0; k < 256; k++) {
202                 currSum += histogram[k];
203                 histogram_sum[k] = currSum;
204             }
205 
206             // output the image:
207             temp = tempin[image->Width * i + j];
208             tempout[outimage->Width * i + j] = (int)(histogram_sum[temp] * constant);
209         }
210     }
211     return (outimage);
212 }
```