



COMPUTER VISION

Course ID: ME4201

Semester 232 – Class P01

ASSIGNMENT 1

Group 4

No.	Student name	Student ID
1	Nguyễn Đức Đạt	2111009
2	Nguyễn Phước Đạt	2111014
3	Nguyễn Trọng Phúc	2152883
4	Đinh Ngọc Quỳnh Anh	2151002

List of Exercises

Exercise 1.	2
Exercise 2.	3
Exercise 3.	7
Exercise 4.	9

Exercise 1.

(Note: The websites in this exercise may take a while to load all the necessary information.)

The field of view (FOV) of a camera is a rectangle of shape $m \times n$. Therefore, the minimum FOV must be a rectangle inscribed by the arcs of the working space. So, the value of FOV is

$$\text{FOV} = (\text{Arc diameter} + \text{Rectangle length}) \times (\text{Arc diameter}) = 3000 \times 1000 \text{ mm}$$

Enter parameters such as FOV, smallest feature (1 mm), moving object... at the website Basler <http://tinyurl.com/2p8pv8m6>. From there, the website outputs 8 cameras according to the given requirements (visit <http://tinyurl.com/4x7xadjk> to see the results). The results are divided into 3 groups as shown below:

Initial FOV (mm)	3000 × 1000		
Model	Basler boost boA8100-16cc/16cm	Basler boost boA6500-36cc/36cm	Basler boost boA9344- 30cc/30cm/70cc/70cm
Sensor size (mm)	26.2 × 17.4	21 × 15.8	29.9 × 22.4
Sensor diagonal (mm)	31.45	26.28	37.4
Camera resolution (px)	8192 × 5460	6580 × 4935	9344 × 7000
Select FOV width (mm)	3000		
New FOV height (mm)	$3000 \cdot \frac{5460}{8192} = 2000$	$3000 \cdot \frac{4935}{6580} = 2250$	$3000 \cdot \frac{7000}{9344} = 2248$
Camera FOV (mm)	3000 × 2000	3000 × 2250	3000 × 2248
Select fixed focal length (mm)	35		
Working distance (mm)	$\frac{35\sqrt{3000^2 + 2000^2}}{31.45} = 4013$	$\frac{35\sqrt{3000^2 + 2250^2}}{26.28} = 4995$	$\frac{35\sqrt{3000^2 + 2248^2}}{37.4} = 3509$

Table 1. Properties and calculation values of Basler cameras

The formulas used in the table above are:

- **Camera FOV** FOV height = FOV width $\cdot \frac{\text{Camera resolution height}}{\text{Camera resolution width}}$.
- **Working distance** Working distance = $\frac{\text{FOV} \times \text{Focal length}}{\text{Sensor size}}$.

Then, we access the Basler website <http://tinyurl.com/28prxsize>, choose appropriate camera and input the parameters Object width (3000 mm), Focal length (35 mm) to check the calculated value:

Model	Formula's result	Website's result	Results link
boA8100-16cc/16cm	4013 mm	4043 mm	Click here
boA6500-36cc/36cm	4995 mm	5035 mm	Click here
boA9344-30cc/30cm/70cc/70cm	3509 mm	3547 mm	Click here

Table 2. Working distance results

Exercise 2.

The matrix of the picture is
$$\begin{bmatrix} 30 & 20 & 30 & 30 \\ 80 & 30 & 100 & 110 \\ 120 & 160 & 30 & 150 \\ 220 & 230 & 240 & 250 \end{bmatrix}.$$

a) The histogram equalization table of the picture matrix:

Pixel	Number of pixels	Probability	Cumulative density function (CDF)	New pixel
20	1	0.0625	1	0
30	5	0.3125	6	85
80	1	0.0625	7	102
100	1	0.0625	8	119
110	1	0.0625	9	136
120	1	0.0625	10	153
150	1	0.0625	11	170
160	1	0.0625	12	187
220	1	0.0625	13	204
230	1	0.0625	14	221
240	1	0.0625	15	238
250	1	0.0625	16	255

Table 3. Histogram equalization table

The new pixel values are calculated according to the following formula:

$$r(\text{Pixel}) = \frac{(\text{CDF} - \text{CDF}_{\min})(\text{Number of gray levels used} - 1)}{\text{Number of elements in the matrix} - \text{CDF}_{\min}}$$

From [Table 3](#), we have $\text{CDF}_{\min} = 1$. There are 16 elements in the matrix, and OpenCV libraries use 256 gray levels for histogram equalization. So, the formula becomes:

$$r(\text{Pixel}) = 17(\text{CDF} - 1)$$

Here are some values calculated according to the formula above:

$$\begin{aligned} r(30) &= 17(6 - 1) = 85 \\ r(100) &= 17(8 - 1) = 119 \\ r(220) &= 17(13 - 1) = 204 \\ r(240) &= 17(15 - 1) = 238 \end{aligned}$$

The image matrix after histogram equalization:

$$\begin{bmatrix} 85 & 0 & 85 & 85 \\ 102 & 85 & 119 & 136 \\ 153 & 187 & 85 & 170 \\ 204 & 221 & 238 & 255 \end{bmatrix}$$

b) The following code listing uses C++ and OpenCV 4.9.0 to perform histogram equalization in section a):

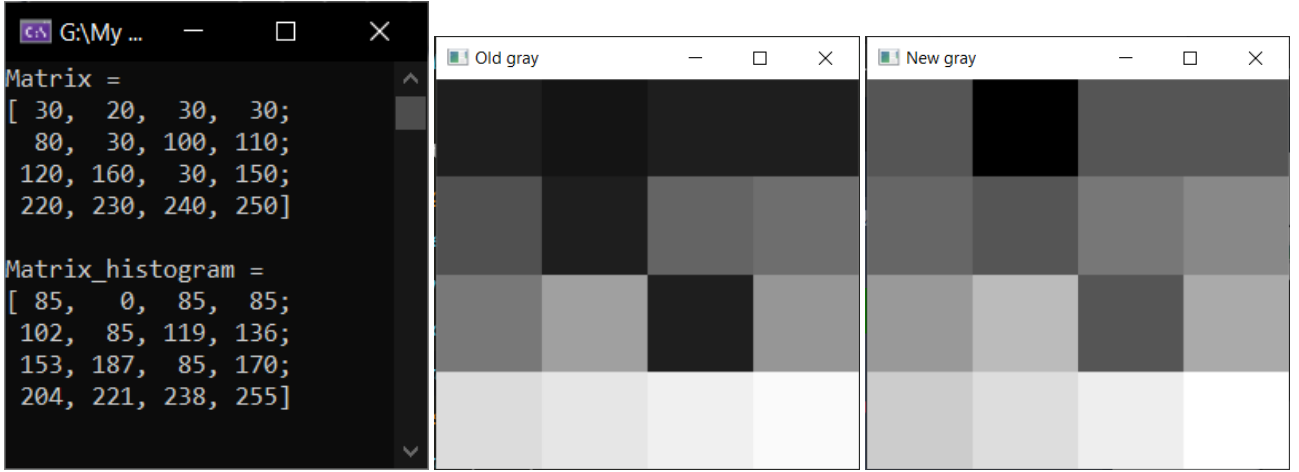
Code listing 1. Exercise 2.b)

```

1 #include "opencv2/opencv.hpp"
2
3 using namespace cv;
4 using namespace std;
5
6 void histogram(string const& name, Mat const& Image)
7 {
8     int bin = 255;
9     int histsize[] = { bin };
10    float range[] = { 0,255 };
11    const float* ranges[] = { range };
12    Mat hist;
13    int chanel[] = { 0 };
14    int hist_height = 256;
15    Mat hist_image = Mat::zeros(hist_height, bin, CV_8SC3);
16    calcHist(&Image, 1, chanel, Mat(), hist, 1, histsize, ranges,
17            true, false);
18    double max_val = 0;
19    minMaxLoc(hist, 0, &max_val);
20    for (int i = 0; i < bin; i++)
21    {
22        float binV = hist.at<float>(i);
23        int height = cvRound(binV * hist_height / max_val);
24        line(hist_image, Point(i, hist_height - height), Point(i,
25            hist_height), Scalar::all(255));
26    }
27    imshow(name, hist_image);
28 }
29
30 int main(int argv, char** argc)
31 {
32    float img[16] = { 30,20,30,30,
33                    80,30,100,110,
34                    120,160,30,150,
35                    220,230,240,250 };
36    Mat gray_img = Mat(4, 4, CV_32F, img);
37    Mat gray_img_his;
38    namedWindow("Old gray", WINDOW_FREERATIO);
39    namedWindow("New gray", WINDOW_FREERATIO);
40    gray_img.convertTo(gray_img, CV_8UC1);
41    equalizeHist(gray_img, gray_img_his);
42    cout << "Matrix = " << endl << " " << gray_img << endl << endl;
43    cout << "Matrix_histogram = " << endl << " " << gray_img_his <<
44        endl << endl;
45    imshow("Old gray", gray_img);
46    imshow("New gray", gray_img_his);
47    histogram("Old histogram", gray_img);
48    histogram("New histogram ", gray_img_his);
49    waitKey();
50 }

```

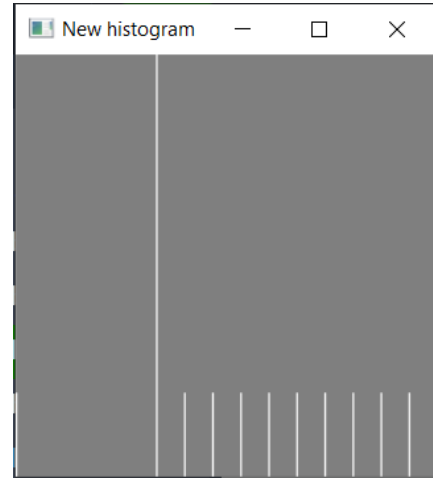
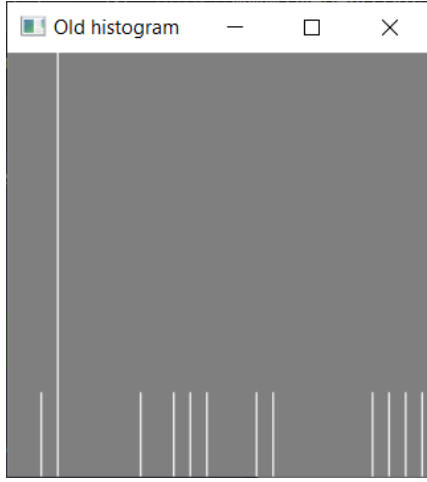
The results of the code above:



(a) Matrix equalization

(b) Picture before equalization

(c) Picture after equalization



(d) Histogram before equalization

(e) Histogram after equalization

Figure 1. The picture before and after histogram equalization

c) We choose the value $T_0 = \frac{\max + \min}{2} = \frac{250 + 20}{2} = 135$ then we divide the elements of the matrix into 2 groups:

- Group of elements smaller than T_0 : 30, 20, 30, 30, 80, 30, 100, 110, 120, 30. The average value of this group is

$$m_1 = \frac{30 + 20 + 30 + 30 + 80 + 30 + 100 + 110 + 120 + 30}{10} = 58$$

- Group of elements greater than T_0 : 160, 150, 220, 230, 240, 250. The average value of this group is

$$m_2 = \frac{160 + 150 + 220 + 230 + 240 + 250}{6} = 208.33$$

The value $T_1 = \frac{m_1 + m_2}{2} = \frac{58 + 208.33}{2} = 133.17$. So $\Delta T = |T_1 - T_0| = |133.17 - 135| = 1.83$

We continue dividing the elements into 2 groups:

- Group of elements smaller than T_1 : 30, 20, 30, 30, 80, 30, 100, 110, 120, 30. The average value of this group is

$$m_1 = \frac{30 + 20 + 30 + 30 + 80 + 30 + 100 + 110 + 120 + 30}{10} = 58$$

- Group of elements greater than T_1 : 160, 150, 220, 230, 240, 250. The average value of this group is

$$m_2 = \frac{160 + 150 + 220 + 230 + 240 + 250}{6} = 208.33$$

The value $T_2 = \frac{m_1 + m_2}{2} = \frac{58 + 208.33}{2} = 133.17$. So $\Delta T = |T_2 - T_1| = 0$.

Therefore, the Otsu thresholding value is $T = 133.17$ and the result matrix is shown below:

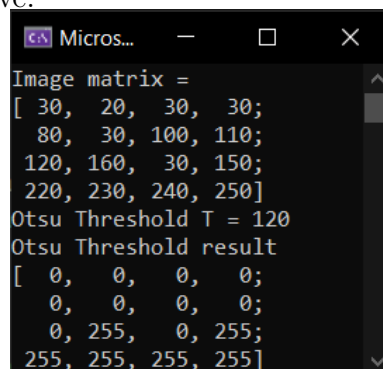
$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 255 & 0 & 255 \\ 255 & 255 & 255 & 255 \end{bmatrix}$$

The following code listing uses C++ and OpenCV 4.9.0 to check the result above:

Code listing 2. Exercise 2.c)

```
1 #include <opencv2/opencv.hpp>
2
3 using namespace cv;
4 using namespace std;
5
6 int main() {
7     Mat source = (Mat_<uint8_t>(4, 4) << 30, 20, 30, 30,
8         80, 30, 100, 110,
9         120, 160, 30, 150,
10        220, 230, 240, 250);
11     Mat dst;
12     double thresh = 0, maxValue = 255;
13     long double thres = threshold(source, dst, thresh, maxValue,
14     THRESH_OTSU);
15
16     cout << "Image matrix = " << endl << " " << source << endl;
17     cout << "Otsu Threshold T = " << thres << endl;
18     cout << "Otsu Threshold result" << endl << " " << dst << endl;
19
20     waitKey(0);
21     return 0;
22 }
```

The result from the code above:



```

Microsoft Windows [Version 10.0.17134.0] (c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\user> g++ exercise2.c -I C:\opencv\include -L C:\opencv\bin -o exercise2.exe
C:\Users\user> .\exercise2.exe

Image matrix =
[ 30, 20, 30, 30;
 80, 30, 100, 110;
120, 160, 30, 150;
220, 230, 240, 250]
Otsu Threshold T = 120
Otsu Threshold result
[ 0, 0, 0, 0;
 0, 0, 0, 0;
 0, 255, 0, 255;
255, 255, 255, 255]
  
```

Figure 2. Otsu Thresholding result from OpenCV

Exercise 3.

The matrix of the picture is $\begin{bmatrix} 1 & 2 & 3 & 3 & 10 & 11 \\ 3 & 10 & 11 & 16 & 11 & 18 \\ 16 & 11 & 18 & 22 & 24 & 25 \\ 23 & 24 & 25 & 25 & 4 & 7 \end{bmatrix}$ and the template is $\begin{bmatrix} 3 & 10 & 11 \\ 16 & 11 & 18 \\ 23 & 24 & 25 \end{bmatrix}$.

Denote the element in the x -th row and y -th column of the image matrix is $I(x, y)$ and the element in the x' -th row and y' -th column of the template matrix is $T(x', y')$. The Normalized Cross-Correlation (NCC) coefficients are determined according to the following formula:

$$R(x, y) = \frac{\sum_{x', y'} [T(x', y') \cdot I(x + x' - 1, y + y' - 1)]}{\sqrt{\sum_{x', y'} T^2(x', y') \cdot \sum_{x', y'} I^2(x + x' - 1, y + y' - 1)}}$$

Substitute the $\sum_{x', y'} T^2(x', y') = 3^2 + 10^2 + 11^2 + 16^2 + 11^2 + 18^2 + 23^2 + 24^2 + 25^2 = 2661$ into the $R(x, y)$ formula, we obtain:

$$R(x, y) = \frac{\sum_{x', y'} [T(x', y') \cdot I(x + x' - 1, y + y' - 1)]}{\sqrt{2661 \sum_{x', y'} I^2(x + x' - 1, y + y' - 1)}}$$

Here are some values obtained from the formula above:

$$\begin{aligned} R(1, 1) &= \frac{1 \cdot 3 + 2 \cdot 10 + 3 \cdot 11 + 3 \cdot 16 + 10 \cdot 11 + 11 \cdot 18 + 16 \cdot 23 + 11 \cdot 24 + 18 \cdot 25}{\sqrt{2661(1^2 + 2^2 + 3^2 + 3^2 + 10^2 + 11^2 + 16^2 + 11^2 + 18^2)}} = 0.942133 \\ R(1, 2) &= \frac{2 \cdot 3 + 3 \cdot 10 + 3 \cdot 11 + 10 \cdot 16 + 11 \cdot 11 + 16 \cdot 18 + 11 \cdot 23 + 18 \cdot 24 + 22 \cdot 25}{\sqrt{2661(2^2 + 3^2 + 3^2 + 10^2 + 11^2 + 16^2 + 11^2 + 18^2 + 22^2)}} = 0.960840 \\ R(1, 3) &= \frac{3 \cdot 3 + 3 \cdot 10 + 10 \cdot 11 + 11 \cdot 16 + 16 \cdot 11 + 11 \cdot 18 + 18 \cdot 23 + 22 \cdot 24 + 24 \cdot 25}{\sqrt{2661(3^2 + 3^2 + 10^2 + 11^2 + 16^2 + 11^2 + 18^2 + 22^2 + 24^2)}} = 0.971414 \\ R(2, 1) &= \frac{3 \cdot 3 + 10 \cdot 10 + 11 \cdot 11 + 16 \cdot 16 + 11 \cdot 11 + 18 \cdot 18 + 23 \cdot 23 + 24 \cdot 24 + 25 \cdot 25}{\sqrt{2661(3^2 + 10^2 + 11^2 + 16^2 + 11^2 + 18^2 + 23^2 + 24^2 + 25^2)}} = 1 \end{aligned}$$

The NCC coefficients matrix is shown below:

$$\begin{bmatrix} R(1, 1) & R(1, 2) & R(1, 3) & R(1, 4) \\ R(2, 1) & R(2, 2) & R(2, 3) & R(2, 4) \end{bmatrix} = \begin{bmatrix} 0.942133 & 0.960840 & 0.971414 & 0.999847 \\ 1 & 0.976255 & 0.881395 & 0.792734 \end{bmatrix}$$

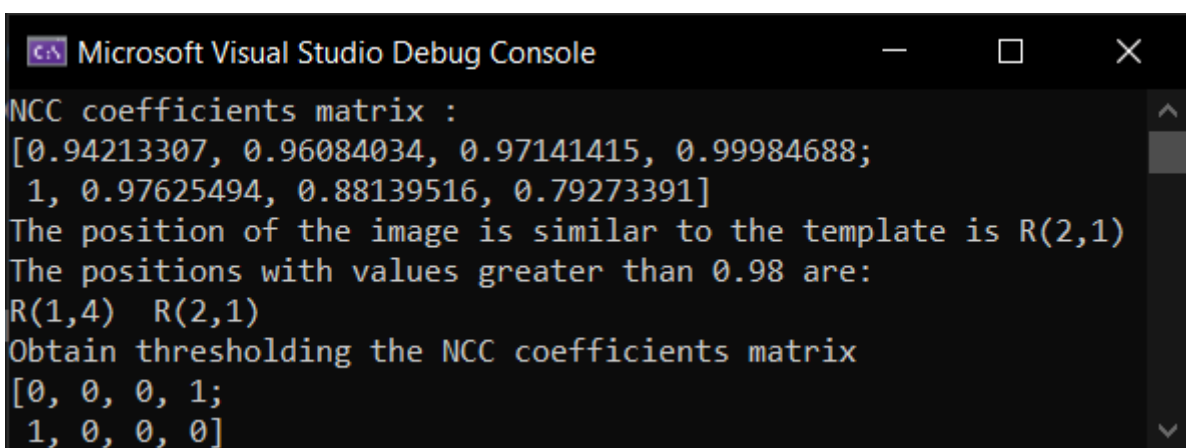
The position of the image is similar to the template is $R(2, 1)$. Applying binary thresholding to the NCC coefficients matrix with the threshold value $T = 0.98$ and the maximum value 1, we obtain the thresholded matrix below:

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

The following code listing is a C++ code using OpenCV 4.9.0 to check the calculation above:

Code listing 3. Exercise 3

```
1 #include <opencv2/opencv.hpp>
2
3 using namespace cv;
4 using namespace std;
5
6 int main()
7 {
8     Mat img = (Mat_<float>(4, 6) << 1, 2, 3, 3, 10, 11,
9         3, 10, 11, 16, 11, 18,
10        16, 11, 18, 22, 24, 25,
11        23, 24, 25, 25, 4, 7);
12     Mat temp = (Mat_<float>(3, 3) << 3, 10, 11,
13        16, 11, 18,
14        23, 24, 25);
15
16     Mat ncc;
17     matchTemplate(img, temp, ncc, TM_CCORR_NORMED);
18     cout << "NCC coefficients matrix :" << endl << ncc << endl;
19     double minVal, maxVal;
20     Point minLoc, maxLoc;
21     minMaxLoc(ncc, &minVal, &maxVal, &minLoc, &maxLoc);
22     cout << "The position of the image is similar to the template is R("
23     << maxLoc.y + 1 << "," << maxLoc.x + 1 << ")" << endl;
24     Mat bin;
25     threshold(ncc, bin, 0.98, 1, THRESH_BINARY);
26     vector<Point> locs;
27     findNonZero(bin, locs);
28     cout << "The positions with values greater than 0.98 are:" << endl
29     ;
30     for (auto loc : locs) {
31         cout << "R(" << loc.y + 1 << "," << loc.x + 1 << ")\t";
32     }
33     cout << "\nObtain thresholding the NCC coefficients matrix" <<
34     endl << " " << bin << endl;
35     waitKey();
36 }
```



```
Microsoft Visual Studio Debug Console
NCC coefficients matrix :
[0.94213307, 0.96084034, 0.97141415, 0.99984688;
 1, 0.97625494, 0.88139516, 0.79273391]
The position of the image is similar to the template is R(2,1)
The positions with values greater than 0.98 are:
R(1,4) R(2,1)
Obtain thresholding the NCC coefficients matrix
[0, 0, 0, 1;
 1, 0, 0, 0]
```

Figure 3. Template matching with NCC coefficients

Exercise 4.

The matrix of the picture is $\begin{bmatrix} 9 & 10 & 11 & 10 & 9 & 10 & 11 \\ 10 & 9 & 95 & 95 & 95 & 10 & 11 \\ 10 & 95 & 10 & 11 & 10 & 95 & 11 \\ 10 & 9 & 95 & 95 & 95 & 13 & 11 \\ 10 & 10 & 10 & 10 & 10 & 13 & 1 \end{bmatrix}$. Adding duplicate border to

the image matrix $\begin{bmatrix} 9 & 9 & 10 & 11 & 10 & 9 & 10 & 11 & 11 \\ 9 & 9 & 10 & 11 & 10 & 9 & 10 & 11 & 11 \\ 10 & 10 & 9 & 95 & 95 & 95 & 10 & 11 & 11 \\ 10 & 10 & 95 & 10 & 11 & 10 & 95 & 11 & 11 \\ 10 & 10 & 9 & 95 & 95 & 95 & 13 & 11 & 11 \\ 10 & 10 & 10 & 10 & 10 & 10 & 13 & 1 & 1 \\ 10 & 10 & 10 & 10 & 10 & 10 & 13 & 1 & 1 \end{bmatrix}$.

a) The Gaussian Filter function is $G_\sigma(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right)$. Substituting $\sigma = 0.3$ into $G_\sigma(x, y)$, then we calculate the kernel matrix:

$$\begin{bmatrix} G_{0.3}(-1, -1) & G_{0.3}(0, -1) & G_{0.3}(1, -1) \\ G_{0.3}(-1, 0) & G_{0.3}(0, 0) & G_{0.3}(1, 0) \\ G_{0.3}(-1, 1) & G_{0.3}(0, 1) & G_{0.3}(1, 1) \end{bmatrix} = \begin{bmatrix} 0.000026 & 0.006836 & 0.000026 \\ 0.006836 & 1.768388 & 0.006836 \\ 0.000026 & 0.006836 & 0.000026 \end{bmatrix}$$

The sum of all the elements of the matrix is 1.79584. Therefore, we obtain the filter kernel matrix:

$$\frac{1}{1.79584} \begin{bmatrix} 0.000026 & 0.006836 & 0.000026 \\ 0.006836 & 1.768388 & 0.006836 \\ 0.000026 & 0.006836 & 0.000026 \end{bmatrix}$$

The matrix after filtering with Gaussian kernel is

$$\begin{bmatrix} 9.007643 & 9.997444 & 11.313337 & 10.326082 & 9.336310 & 10.001266 & 10.996164 \\ 9.993623 & 9.662446 & 94.025522 & 94.351643 & 94.021715 & 10.650982 & 10.997415 \\ 10.323551 & 93.695564 & 10.977020 & 11.636878 & 10.977093 & 93.718434 & 11.319788 \\ 9.997444 & 9.662446 & 94.021715 & 94.351643 & 94.036986 & 13.616411 & 10.970664 \\ 9.999985 & 9.997444 & 10.324816 & 10.326082 & 10.336340 & 12.943854 & 1.084251 \end{bmatrix}$$

The elements of the matrix after filtering are calculated using the convolution formula. Here are some of the calculated element values:

$$a(1, 1) = \frac{9 \cdot 0.000026 + 9 \cdot 0.006836 + 10 \cdot 0.000026 + 9 \cdot 0.006836 + 9 \cdot 1.768388}{1.79584}$$

$$+ \frac{10 \cdot 0.006836 + 10 \cdot 0.000026 + 10 \cdot 0.006836 + 9 \cdot 0.000026}{1.79584} = 9.007643$$

$$a(1, 2) = \frac{9 \cdot 0.000026 + 10 \cdot 0.006836 + 11 \cdot 0.000026 + 9 \cdot 0.006836 + 10 \cdot 1.768388}{1.79584}$$

$$+ \frac{11 \cdot 0.006836 + 10 \cdot 0.000026 + 9 \cdot 0.006836 + 95 \cdot 0.000026}{1.79584} = 9.997444$$

$$a(2, 2) = \frac{9 \cdot 0.000026 + 10 \cdot 0.006836 + 11 \cdot 0.000026 + 10 \cdot 0.006836 + 9 \cdot 1.768388}{1.79584}$$

$$+ \frac{95 \cdot 0.006836 + 10 \cdot 0.000026 + 95 \cdot 0.006836 + 10 \cdot 0.000026}{1.79584} = 9.662446$$

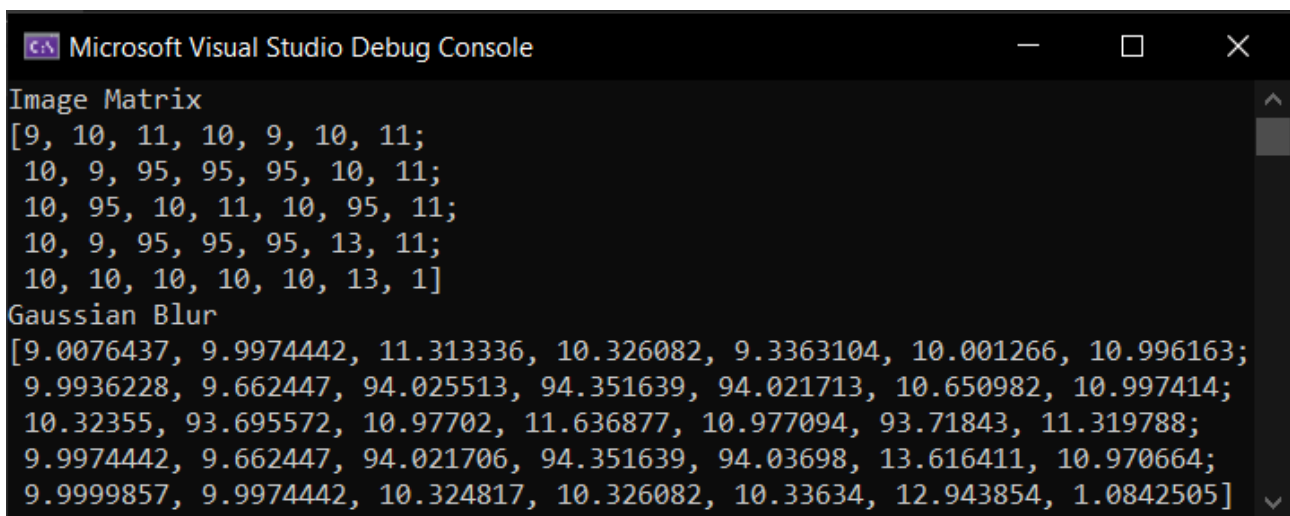
$$a(3, 3) = \frac{9 \cdot 0.000026 + 95 \cdot 0.006836 + 95 \cdot 0.000026 + 95 \cdot 0.006836 + 10 \cdot 1.768388}{1.79584}$$

$$+ \frac{11 \cdot 0.006836 + 9 \cdot 0.000026 + 95 \cdot 0.006836 + 95 \cdot 0.000026}{1.79584} = 10.977020$$

The following code listing is a C++ code using OpenCV 4.9.0 to check the calculation above:

Code listing 4. Exercise 4.a)

```
1 #include <opencv2/opencv.hpp>
2
3 using namespace cv;
4 using namespace std;
5
6 int main()
7 {
8     float sigma = 0.3;
9     Mat src = (Mat_<float>(5, 7) << 9, 10, 11, 10, 9, 10, 11,
10         10, 9, 95, 95, 95, 10, 11,
11         10, 95, 10, 11, 10, 95, 11,
12         10, 9, 95, 95, 95, 13, 11,
13         10, 10, 10, 10, 10, 13, 1);
14     Mat dst;
15     GaussianBlur(src, dst, Size(3, 3), sigma, sigma,
16     BORDER_REPLICATE);
17     cout << "Image Matrix" << endl << "" << src << endl;
18     cout << "Gaussian Blur" << endl << "" << dst << endl;
19     waitKey();
20 }
```



```
Microsoft Visual Studio Debug Console

Image Matrix
[9, 10, 11, 10, 9, 10, 11;
 10, 9, 95, 95, 95, 10, 11;
 10, 95, 10, 11, 10, 95, 11;
 10, 9, 95, 95, 95, 13, 11;
 10, 10, 10, 10, 10, 13, 1]
Gaussian Blur
[9.0076437, 9.9974442, 11.313336, 10.326082, 9.3363104, 10.001266, 10.996163;
 9.9936228, 9.662447, 94.025513, 94.351639, 94.021713, 10.650982, 10.997414;
 10.32355, 93.695572, 10.97702, 11.636877, 10.977094, 93.71843, 11.319788;
 9.9974442, 9.662447, 94.021706, 94.351639, 94.03698, 13.616411, 10.970664;
 9.9999857, 9.9974442, 10.324817, 10.326082, 10.33634, 12.943854, 1.0842505]
```

Figure 4. Gaussian Blur filtering result from OpenCV

b) The Sobel operator uses two different kernel matrices:

- Taking the derivative with respect to x : $M_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$
- Taking the derivative with respect to y : $M_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$

Using the Sobel M_x operator, we obtain the image after filtering:

$$\begin{bmatrix} 2 & 91 & 86 & -6 & -85 & -78 & 4 \\ 84 & 172 & 88 & -2 & -86 & -165 & -81 \\ 168 & 170 & 4 & 0 & 1 & -166 & -169 \\ 83 & 170 & 88 & 0 & -77 & -176 & -100 \\ -1 & 85 & 86 & 0 & -73 & -111 & -38 \end{bmatrix}$$

Here are some values that were calculated through the Sobel M_x operator:

$$a(1,1) = 9 \cdot (-1) + 9 \cdot 0 + 10 \cdot 1 + 9 \cdot (-2) + 9 \cdot 0 + 10 \cdot 2 + 10 \cdot (-1) + 10 \cdot 0 + 9 \cdot 1 = 2$$

$$a(2,2) = 9 \cdot (-1) + 10 \cdot 0 + 11 \cdot 1 + 10 \cdot (-2) + 9 \cdot 0 + 95 \cdot 2 + 10 \cdot (-1) + 95 \cdot 0 + 10 \cdot 1 = 172$$

Using the Sobel M_y operator, we obtain the image after filtering:

$$\begin{bmatrix} 2 & 83 & 252 & 340 & 257 & 86 & 0 \\ 88 & 170 & 84 & 2 & 88 & 171 & 85 \\ 0 & 0 & 0 & 0 & 3 & 6 & 3 \\ -85 & -170 & -86 & -2 & -83 & -174 & -112 \\ 1 & -83 & -254 & -340 & -255 & -95 & -30 \end{bmatrix}$$

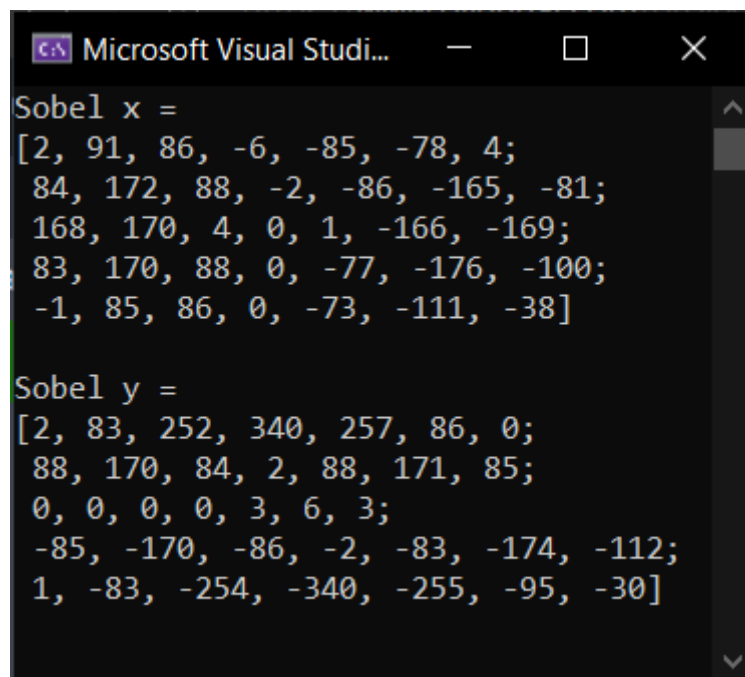
Here are some values that were calculated through the Sobel M_y operator:

$$a(1,1) = 9 \cdot (-1) + 9 \cdot (-2) + 10 \cdot (-1) + 9 \cdot 0 + 9 \cdot 0 + 10 \cdot 0 + 10 \cdot 1 + 10 \cdot 2 + 9 \cdot 1 = 2$$

$$a(2,2) = 9 \cdot (-1) + 10 \cdot (-2) + 11 \cdot (-1) + 10 \cdot 0 + 9 \cdot 0 + 95 \cdot 0 + 10 \cdot 1 + 95 \cdot 2 + 10 \cdot 1 = 170$$

Code listing 5. Exercise 4.b)

```
1 #include <opencv2/opencv.hpp>
2
3 using namespace cv;
4 using namespace std;
5
6 int main()
7 {
8     Mat src = (Mat_<double>(5, 7) << 9, 10, 11, 10, 9, 10, 11,
9         10, 9, 95, 95, 95, 10, 11,
10        10, 95, 10, 11, 10, 95, 11,
11        10, 9, 95, 95, 95, 13, 11,
12        10, 10, 10, 10, 10, 13, 1);
13
14     Mat sobelx, sobely;
15     Sobel(src, sobelx, CV_64F, 1, 0, 3, 1, 0, BORDER_REPLICATE);
16     Sobel(src, sobely, CV_64F, 0, 1, 3, 1, 0, BORDER_REPLICATE);
17     cout << "Sobel x = " << endl << " " << sobelx << endl << endl;
18     cout << "Sobel y = " << endl << " " << sobely << endl << endl;
19     waitKey();
20 }
```



```
Microsoft Visual Studi...  
Sobel x =  
[2, 91, 86, -6, -85, -78, 4;  
84, 172, 88, -2, -86, -165, -81;  
168, 170, 4, 0, 1, -166, -169;  
83, 170, 88, 0, -77, -176, -100;  
-1, 85, 86, 0, -73, -111, -38]  
  
Sobel y =  
[2, 83, 252, 340, 257, 86, 0;  
88, 170, 84, 2, 88, 171, 85;  
0, 0, 0, 0, 3, 6, 3;  
-85, -170, -86, -2, -83, -174, -112;  
1, -83, -254, -340, -255, -95, -30]
```

Figure 5. Sobel filtering result from OpenCV