MODULE: PROGRAMMING METHODS FOR ROBOTICS

PROJECT REPORT: IMAGE PROCESSING WITH DIFFERENT KERNELS

January 14, 2019

Student : Yang YOU
Cranfield University
MSc in Robotics

Contents

1 ABSTRACT

Image processing is a widely used technique in many areas such as machine learning, medical inspection and camera applications. In this experiment, various kernels are used to perform the image processing task. This report aims at describing the approaches which are used, discussing the advantages and disadvantages among those filters.

2 INTRODUCTION

Usually, image processing refers to digital image processing which computer is used to analysis those images. Those digital images are defined by matrix, the element in the matrix is called Pixel, which can be converted to standard Red Green Blue type and their value is called Gray-Scale Value. By manipulating the grey value of pixels, different image results can be obtained .

Generally, there are two types of process, smoothing and gradient operator respectively [5]. The purpose of smoothing is reducing the noise and other disturbances, a similar idea can be observed in signal processing is using the Fourier transform to eliminate high frequency signals. Gradient operator is based on the information of the image. By finding the highest gradient value's parts in a image function, the area which have the most evident difference can be detected, some typical methods which use gradient operator are image sharpening and edge detection. However, smoothing and gradient operator process also have drawbacks. The smoothing process will smooth and blur the edges of image which contain important information. The process based on gradient operators will increase the noise level as the noise also have a high frequency proprieties which means they also have a high gradient value.

In this paper, different kernels are used to perform smoothing, sharpening and edge detecting tasks to verify the two types of process. The structure of this paper is designed as follows. Description of different kernels and their applications is presented in the next section. The Section 3 will give the approaches used in this experiment and presenting the results. In the Section 4, the results will be analyzed and concluded.

3 IMAGE PROCESSING

3.1 Image smoothing

The principal of image smoothing is using the average of one pixel's neighbours to replace that pixel. The reason why this method can reduce the noise followed by statistical principals. The variance of the sampling distribution of the mean is lower than the variance of original data. However, in the image processing, if the noise spot is too large, this method of smoothing wll not have a promising result.

We assume in every image, there are specific pixel values $g(i, j)$, i and j are the indices for rows and columns respectively. Then a filter (or convolution mask) $h(m, n)$ can be applied, the convolution process is defined as follows, where $f(i, j)$ is the output pixel:

$$f(i,j) = \sum_{m=-N}^{N} \sum_{n=-N}^{N} h(m,n)g(i+m, i+n) \tag{1}$$

3.2 Image Sharpening

Image sharpening [1] is a process which can increase the apparent sharpness of an image. Basically, this process uses a high pass filter to an image, which will increase the contrast between the dark and bright area. The function of sharpening can be defined as equation 2, the output image f obtained from original image$h$ by a subtraction of coefficient C multiply by a measure function of sharpness $S(i, j)$, which computed by gradient operators [5].

$$f(i,j) = g(i,j) - CS(i,j) \tag{2}$$

Laplacian, which is a differential operator given by the divergence of the gradient of a function can be used in this process. Its definition is presented as follows, where x and y are two directions in the image.

$$\nabla^2 g(x,y) = \frac{\partial^2 g(x,y)}{\partial x^2} + \frac{\partial^2 g(x,y)}{\partial y^2} \tag{3}$$

For the image with discrete pixel values, an approximation for the second order partial differential equation can be made for Equation 3. The approximate functions are presented as follows.

$$\frac{\partial^2 g(x,y)}{\partial x^2} = (g(i+1,j) - g(i,j)) - (g(i,j) - g(i-1,j)) = g(i+1,j) + g(i-1,j) - 2g(i,j) \tag{4}$$

$$\frac{\partial^2 g(x,y)}{\partial y^2} = (g(i,j+1) - g(i,j)) - (g(i,j) - g(i,j-1)) = g(i,j+1) + g(i,j-1) - 2g(i,j) \quad (5)$$

By summing up those two equations, the gradient estimation can be deduced as follows.

$$\nabla^2 g(x,y) = [g(i+1,j) + g(i-1,j) + g(i,j+1) + g(i,j-1)] - 4g(i,j) \quad (6)$$

3.3 Image Edge Detection

Gradient information methods are also used in image edge detection task. Most of them can be divided into two groups, First Order Methods and Second Order Methods respectively. The first order method is using operators to estimate the first-order image derivatives. Operators such as Sobel [4], Roberts [3] and Prewitt [2] are widely used for the first-order estimation. The Laplacien operator is commonly used in estimation of the second-order of image edge's amplitude. However, Unlike the image sharpening strengthen the contrast around edges in the image, the edge detection only extract the edges and omit other information of one image.

In this project, first order methods are also implemented. The gradient for image function g(x,y) is defined as equation 7.

$$\nabla g(x,y) = [\frac{\partial g(x,y)}{\partial x}, \frac{\partial g(x,y)}{\partial y}] \quad (7)$$

The norm of the gradient is used for measuring the rate of difference among pixels, which is defined as follow.

$$|\nabla g(x,y)| = \sqrt{(\frac{\partial g(x,y)}{\partial x})^2 + (\frac{\partial g(x,y)}{\partial y})^2} \quad (8)$$

4 Experiments and Results

4.1 PPM file processing

In this experiment, a particular image format which called ppm was used. The ppm image format consists of a header followed by the image pixel data. The header contains the information of : 1. The type of storage used for the pixel values of the image. If the number is P3 it means that the pixel data is stored in ascii text format. If the number is P6 it means that the pixel data is stored in compressed binary format. 2. Comment which begins with the #. 3. Width, height and maximum colour value of the image (usually 255) The pixel

data that follows consists of RGB values in the range 0-maximum colour value. Function *openIOFiles* was used to open the binary ppm file and create an ascii type file for writing, *convertP6ToP3* and *writeP3Image* functions are dealing with converting binary image to ascii type and writing ascii data to the output file respectively.

4.2 Programming method

To realize the smoothing, sharpening and edge-detection tasks, *smooth*, *sharpen* and *edgedetection* functions were designed to process the image data. The pixel has three types of values which are Red, Green and Blue. All of them shared the same proprieties such as the minimal and maximum values, they also treated by the same processes. Thus, the <Class> type in C++ was introduced to defined the pixel type. The class type was widely used in the object oriented programming in c++. After introduced the class type for the pixel, new operators need to be defined for calculations, as the new <Pixel type> didn't have any operator rules previously. To prevent the overlap of data caused by reading the pixel value and also doing the calculation in the meantime, two pixel type variables *inputpixel* and *outputpixel* were initialized. Moreover, the reading and writing of data were based on the stream of input of output, which are namely *ifstream* and *ofstream*. Those functions relayed on libraries, which are <iostream>, <fstream> and <vector> respectively. In the end, to prevent the overflow the pixel value, which was limited between the minimal and maximal values, a mechanism of auto-changing in-allowed values was designed in every process.

The functions mentioned above are provided in the Appendix. For the time efficiency, all the filters which were used in the following tasks had $3 * 3$ dimensions.

4.3 Smooth filter

First, a filter which defined as Table 1 was used for smoothing task. After a convolution process by *smooth* function, the result is present in shown in Fig 2 which compared with the original image in Fig 1. Smoother and more blurred effects can be observed in Fig 2 compared with Fig 1, in the meantime, the edges are also more indistinct in the Fig 2.

Table 1: Smoothing filter

0	1	0
1	0	1
0	1	0

Figure 1: Original image

Figure 2: Image after smoothing

To exam how significant a smoothing filter can reduce the noise level, noises which is defined by Gaussian distribution in 2-D (Equation. 9) was added into the image as Fig 3 .

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{9}$$

Then, same process was applied to that image which has noise. A result in Fig 4 which has a clearly decreased noise level can be can be observed.

Figure 3: Image with noises

Figure 4: Image after smoothing

4.4 Sharpen filter

According to the Equation 6 of Laplacian operator, the filter which is defined as Table 2 using two directions (x, y) can be used for sharpening part. By adding more rotations to the Equation 3, more filters can be made for the sharpening process. The first sharpening filter (Table 3), which contains more information with horizontal, vertical and rotations was

Table 2: Original Laplacian filter

0	1	0
1	-4	1
0	1	0

Table 3: Sharpen filter 1: Laplacian filter with 8 neignbours

1	1	1
1	-8	1
1	1	1

implemented. Moreover, a second sharpening filter (Table 4) which has a higher center value compared with the first sharpening filter was implemented to test the difference in results.

Table 4: Sharpening filter 2

-1	-1	-1
-1	12	-1
-1	-1	-1

One clown image (Fig 5) was introduced to this task. Those two sharpening filters would be used as convolution kernels in the convolution process.

Figure 5: 2D Gaussian distribution graph

The convolution process was run by *sharpen* function, results are presented as below.

In the Figure 6, edges were more evident compared with the original image (Fig 5) and Figure 7. However, the colors became significantly bright in Figure 6. This result could be explained by Equation 2, which indicated the sharpening output is obtained by the original pixel value minus the factor of coefficient and Laplacian operator processed value, the more greater the coefficient C was, the brighter the image was. This method (Eq 2) would be also supported in the next task, $S(i, j)$ could measure the edges' information. Moreover, compared with Figure 6, the result in Figure 7 showed a better grasp of color information but also has a lower contrast along the edges.

In the meantime, to exam to what extent does the sharpen process can increase the noise

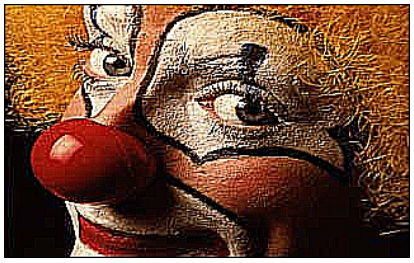

Figure 6: Image after sharpening filter 1 processed, C = 1.0

Figure 7: Image after sharpening filter 2 processed

level, a Gaussian distributed noise (Eq 9) was also introduced to this task. The image with noise (Fig 8) and sharpened one with sharpening filter 2 (Fig 9) are as follows. A dramatic increase of noise level could be observed in the Figure 9 which supported the argument mentioned above.

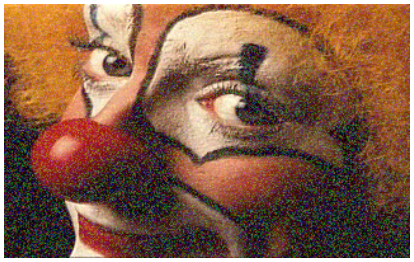

Figure 8: Image with noise

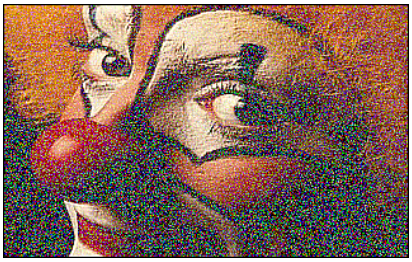

Figure 9: Image with noise after sharpening

4.5 Edge-Detection filter

In this part, the filters with Sobel Operator [4] was introduced. Those filters were defined in Table 5 and Table 6.

Table 5: Sobel Vertical filter

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

Table 6: Sobel horizontal filter

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

Those filters were also used as kernels for convolution process in function *edgedtection*. The Figure 10 was introduced as the original image for this task. After the convolution process, two output images (Fig 12 and Fig 13) were created which represented the vertical edges' information and horizontal edges' information respectively.

Figure 10: Original Image

Figure 11: Image after Sobel filer process

Figure 12: Image after Sobel vertical filter process

Figure 13: Image after Sobel horizontal filer process

Then, accroding the Equation 8 mentioned above, a combination of vertical and horizontal output was needed to complete the whole edges' information. This step was defined by the square root of the sum by vertical and horizontal values' square. The final output by Sobel operator filter was presented in Figure 15. Noise in the background of the original image could be also observed in the output image, but not as evident as the sharpening process output. Moreover, due to the complexity of calculation in Equation 8, this process could be time consuming. If a less time consuming need is raised up, the Equation 10 can be used as a approximation of Equation 8 to reduce the processing time with a slight loss of the accuracy about image edges' information.

$$|\nabla g(x,y)| = |\frac{\partial g(x,y)}{\partial x}| + |\frac{\partial g(x,y)}{\partial y}| \tag{10}$$

Similar method like Prewitt Operator [2] also used vertical and horizontal filters to extract

the edges' information. Both Sobel and Prewitt operator are first order methods

Furthermore, second order methods which are mentioned in the section 3 can be also used to detect the edges of one image. Thus, the last experiment tested the Laplacian operator to measure the edges' information ($S(i, j)$ in Eq 2) and then compared with the first order method's result (Fig 15). The result is presented in Figure 14 by a comparison with Sobel filter processed result (Fig 15).

Compared with Figure 15 which processed by Sobel filter, the result of Laplacian filter processed has a better detailed edges' information. This situation could be supported by the definition of Laplacian, which a second order process can extract more information compared with first order process. Last but not least, the Laplacian process used only one filter which was more time efficient compared with Sobel filter process.

Figure 14: Image after Laplacian filter process

Figure 15: Image after Sobel filer process

5 CONCLUSION

In this experiment, various operators such as Sobel, Laplacian and variations were used as convolution kernels and compared their advantages and disadvantages.The difference between first order method process and second order method process has been tested and presented. To process the image data, the C++ Standard library and STL such as <iostream>, <fstream> and <vector> has been used. Features such as Object Oriented Programming, stream input and output were implemented. Techniques like pointer, reference and class has been used in the program to realize the desired goals. In the end, this project provided an

interesting and useful practise in image processing, programming skills were also practised and strengthened during the experiment.

REFERENCES

[1] Behar Cheung, , Frank C, and Malek Adjouadi. A. rosenfeld and a. c. kak, digital image processing, vol. 2, 2nd ed. new york: Academic, 1982.

[2] J. M. S. Prewitt. *Object enhancement and extraction.* Picture Processing and Psychopictorics, 1970.

[3] Lawrence G. Roberts. *Machine Perception of Three-Dimensional Solids.* Outstanding Dissertations in the Computer Sciences. Garland Publishing, New York, 1963.

[4] Irwin Sobel. An isotropic 3x3 image gradient operator. *Presentation at Stanford A.I. Project 1968*, 02 2014.

[5] Milan Sonka, Vaclav Hlavac, and Roger Boyle. *Image Processing, Analysis, and Machine Vision.* Thomson-Engineering, 2007.

A APPENDIX

```
 // smooth function (Yang)
void smooth(vector<vector<Pixel> >& image, vector<vector<Pixel> >& output)
{
        int h = image.size();
        int w = image[0].size();

        // allocate memory for the output piexel
        output.resize(h); // allocate h rows
        for (int i = 0; i < h; i++)
        {
                output[i].resize(w);    // for each row allocate w columns
        }

        Pixel sum;
        for (int i=1; i<h-1; i++)
                for (int j = 1; j < w - 1; j++)
                {
                        sum =  image[i + 1][j] + image[i - 1][j] + image[i][j + 1]
```

```
                        sum = sum / 4;
                        if (sum.getBlue() >= 255) {
                                sum.setBlue(255);
                        }
                        else if (sum.getBlue() <= 0) {
                                sum.setBlue(0);
                        };
                        if (sum.getGreen() >= 255) {
                                sum.setGreen(255);
                        }
                        else if (sum.getGreen() <= 0) {
                                sum.setGreen(0);
                        };
                        if (sum.getRed() >= 255) {
                                sum.setRed(255);
                        }
                        else if (sum.getRed() <= 0) {
                                sum.setRed(0);
                        };
                        output[i][j] = sum;
                }
}

//Sharpen function (Yang)
void sharpen(vector<vector<Pixel> >& image, vector<vector<Pixel> >& output)
{
        int h = image.size();
        int w = image[0].size();

        // allocate memory for the output piexel
        output.resize(h); // allocate h rows
        for (int i = 0; i < h; i++)
        {
                output[i].resize(w);    // for each row allocate w columns
        }
```

```
Pixel sum;
for (int i = 1; i < h - 1; i++) {
        for (int j = 1; j < w - 1; j++)
        {
        //First sharpen filter: Laplacian operator. (f(i,j) = g(i,j) - c*s(
        //int C = 1.0; // Cofficient
        //int blue = image[i][j].getBlue() + C*( image[i][j].getBlue() * 8
        //int green = image[i][j].getGreen() + C* (image[i][j].getGreen() *
        //int red = image[i][j].getRed()+ C*(image[i][j].getRed() * 8 - (im

        //Laplacian operator could be also used for edge detection. -S(i,j)
        int blue = (image[i][j].getBlue() * 8 - (image[i - 1][j - 1] + imag
        int green = (image[i][j].getGreen() * 8 - (image[i - 1][j - 1] + im
        int red = (image[i][j].getRed() * 8 - (image[i - 1][j - 1] + image[

        //Second sharpen filter
        //int blue = image[i][j].getBlue() * 12 - (image[i - 1][j - 1] + im
        //int green = image[i][j].getGreen() * 12 - (image[i - 1][j - 1] +
        //int red = image[i][j].getRed() * 12 - (image[i - 1][j - 1] + imag
        //blue = blue / 4;
        //green = green / 4;
        //red = red / 4;

        //prevent overflow
        if (blue >= 255) {
                blue = 255;
        }
        else if (blue <= 0) {
                blue = 0;
        };
        if (green >= 255) {
                green = 255;
        }
        else if (green <= 0) {
```

```
                        green =0 ;
                };
                if (red >= 255) {
                red = 255;
                }
                else if (red <= 0) {
                        red = 0;
                };
                output[i][j].setBlue(blue);
                output[i][j].setGreen(green);
                output[i][j].setRed(red);
                }
        }
}

// edge detection function (Yang)
void edgedetection(vector<vector<Pixel> >& image, vector<vector<Pixel> >& output, v
{
        int h = image.size();
        int w = image[0].size();

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

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

        int sum_blue;
```

```
int sum_green;
int sum_red;
// allocate memory for the output piexel
output.resize(h); // allocate h rows
vectorial.resize(h);
horizontal.resize(h);

for (int i = 0; i < h; i++)
{
        output[i].resize(w);    // for each row allocate w columns
        vectorial[i].resize(w);    // for each row allocate w columns
        horizontal[i].resize(w);    // for each row allocate w columns
}

Pixel sum_vertical, sum_horizontal;

for (int i = 1; i < h - 1; i++) {
        for (int j = 1; j < w - 1; j++)
        {
        //Vertical value variables' definition
        int blue_vertical;
        int green_vertical;
        int red_vertical;

        //Horizontal value variables' definition
        int blue_horizontal;
        int green_horizontal;
        int red_horizontal;

        //Initialize sum variables by 0
        int sum_blue_vertical = 0;
        int sum_green_vertical = 0;
        int sum_red_vertical = 0;
        int sum_blue_horizontal = 0;
        int sum_green_horizontal = 0;
```

```
int sum_red_horizontal = 0;

//filter process
for(int row =i - 1; row <= i+1;row++)
        for (int col = j - 1; col <= j + 1; col++) {
                int a = row - (i -1);
                int b = col - (j - 1);
                //verical filter
                blue_vertical = image[row][col].getBlue() * vertica
                green_vertical = image[row][col].getGreen() * verti
                red_vertical = image[row][col].getRed() * vertical_

                sum_blue_vertical = sum_blue_vertical + blue_vertic
                sum_green_vertical = sum_green_vertical + green_ve
                sum_red_vertical = sum_red_vertical + red_vertical;

                //horizontal filter
                blue_horizontal = image[row][col].getBlue() * horiz
                green_horizontal = image[row][col].getGreen() * ho
                red_horizontal = image[row][col].getRed() * horizo

                sum_blue_horizontal = sum_blue_horizontal + blue_ho
                sum_green_horizontal = sum_green_horizontal + green
                sum_red_horizontal = sum_red_horizontal + red_horiz
        }
//prevent vertical process overflow
if (sum_blue_vertical >= 255) {
        sum_blue_vertical = 255;
}
else if (sum_blue_vertical <= 0) {
        sum_blue_vertical = 0;
};
if (sum_green_vertical >= 255) {
        sum_green_vertical = 255;
}
```

```
else if (sum_green_vertical <= 0) {
        sum_green_vertical = 0;
};
if (sum_red_vertical >= 255) {
        sum_red_vertical = 255;
}
else if (sum_red_vertical <= 0) {
        sum_red_vertical = 0;
};
//prevent horizontal process overflow
if (sum_blue_horizontal >= 255) {
sum_blue_horizontal = 255;
}
else if (sum_blue_horizontal <= 0) {
        sum_blue_horizontal = 0;
};
if (sum_green_horizontal >= 255) {
        sum_green_horizontal = 255;
}
else if (sum_green_horizontal <= 0) {
        sum_green_horizontal = 0;
};
if (sum_red_horizontal >= 255) {
        sum_red_horizontal = 255;
}
else if (sum_red_horizontal <= 0) {
        sum_red_horizontal = 0;
};

//vertical filter process result
vectorial[i][j].setBlue(sum_blue_vertical);
vectorial[i][j].setGreen(sum_green_vertical);
vectorial[i][j].setRed(sum_red_vertical);
//horizontal filter process result
horizontal[i][j].setBlue(sum_blue_horizontal);
```

```
            horizontal[i][j].setGreen(sum_green_horizontal);
            horizontal[i][j].setRed(sum_red_horizontal);

            sum_blue = sqrt((pow(sum_blue_horizontal,2)) + (pow(sum_blue_vertic
            sum_green = sqrt((pow(sum_green_horizontal , 2)) + (pow(sum_green_
            sum_red = sqrt((pow(sum_red_horizontal , 2)) + (pow(sum_red_vertica

            if (sum_blue >= 255) {
                    sum_blue = 255;
            }
            else if (sum_blue <= 0) {
                    sum_blue = 0;
            };
            if (sum_green >= 255) {
                    sum_green = 255;
            }
            else if (sum_green <= 0) {
                    sum_green = 0;
            };
            if (sum_red >= 255) {
                    sum_red = 255;
            }
            else if (sum_red <= 0) {
                    sum_red = 0;
            };
            output[i][j].setBlue(sum_blue);
            output[i][j].setGreen(sum_green);
            output[i][j].setRed(sum_red);

            sum_blue = 0;
            sum_green = 0;
            sum_red = 0;

            }
    }
```

```
}
```