Kyungpook National University

Artificial Brain Research - ABR

BEP

Lesson and Assignment 5

Filtering

Gwenaelle Cunha Sergio

Winter 2014

1 Summary of Lecture Material

Filtering, edge detection and morphology are all image pre-processing methods. The first one removes image noises, the second one is used to find boundaries or contours in an object and the third one is used to increase or decrease the relation between two parts of an object. They all use similar methods, or mask, and ways (convolution).

An image convolution is represented by the following formula, where h is the filter, $*$ denotes the convolution and m is the filter size:

$$p(x,y) = q * h = \sum_{i=-m/2}^{m/2} \sum_{j=-m/2}^{m/2} h(j,i)q(x-j,y-i)$$

1.1 Noise

Image noise is any information that should not have been included in the original image. The most common types of noise are: uniform, salt and pepper and gaussian, **Fig. 1**.

(a) Original image (b) Salt and pepper noise (c) Gaussian noise

Figure 1: Image noises

1.2 Filtering

To remove noise in an image, it's necessary to use filtering [1] methods. The way it functions is by taking each pixel and calculating its new value using a spatial mask and its neighboring pixels.

Usually the mask size is an odd number, for example, 3x3, 5x5, 7x7, etc; and the way the mask moves around the picture is from the up-left corner to the down-right corner, covering all the pixels, that is, an image convolution.

There a few techniques that allows the analysis of pixels in the border of the image, like adding a new boundary to the original image with the color black and copying the the same boundary again, **Fig. 2**.

[1] Spatial filtering

(a) Original image

(b) Set boundary pixel as 0

(c) Copy original boundary

Figure 2: Boundary

1.2.1 Types of filtering/filter

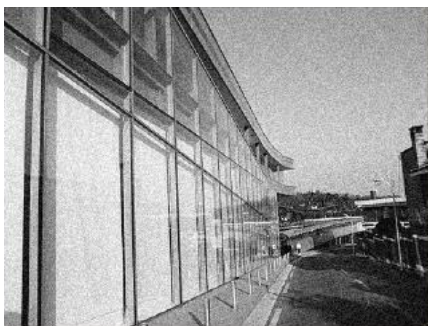
Low-pass Filtering (LPF): only low frequency data (meaning that the intensity changes slowly) can pass; the high frequency parts will appear blurred; and removes high frequency noise, like the salt and pepper one.

High-pass Filtering (HPF): only high frequency data (meaning that the intensity changes quickly) can pass; the high frequency parts will be emphasized, that is, it's a good method for showing boundaries in an image.

Average Filter: calculates the average value of the current pixel considering the ones adjacent to it. The mask looks like that:

$$M_{3x3} = \begin{pmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{pmatrix}$$

The values in the mask are $1/9$ because 9 pixels will be used in this specific calculation. If the mask were to be 5x5, the value would be $1/25$. From **Fig. 3**, it can be seen that the resulting image is blurred, so it's very similar to the LPF.



(a) Original image

(b) Result image for average filter

Figure 3: Average Filter

Median Filtering/Filter: used for understory edge part to remove impulse noise. It's a representation of an order filter, since it sorts out the mask values and takes the middle element as the pixel's new value. See **Fig. 4(b)**.

Maximum Filter: similar to median filter, but instead of taking the middle value, it takes the largest value as the pixel's new value. See **Fig. 4(c)**.

(a) Original image

(b) Result image for median filter

(c) Result image for maximum filter

Figure 4: Median and Maximum Filter

Minimum Filter: opposite of maximum filter, since it takes the smallest value as the pixel's new value. See **Fig. 5**.

(a) Original image

(b) Result image for minimum filter

Figure 5: Minimum Filter

Midpoint Filter: sorts the mask values and takes the average value between the largest and smallest values, **Fig. 6**.

(a) Original image

(b) Result image for midpoint filter

Figure 6: Midpoint Filter

Alpha-trimmed Mean Filter: opposite of midpoint filter, it takes all the values, except the smallest and largest to calculate the average value, according to alpha ($\frac{\sum_{\alpha}^{N-\alpha} elements}{N-2\alpha}$). See **Fig. 7**.

(a) Original image

(b) Result image for alpha-trimmed mean filter

Figure 7: Alpha-trimmed mean Filter

2 Assignment

This assignment consists of answering the following questions.

2.1 Investigate image noises

Image noise is a random variation in brightness or color information in images. It's not present in the object imaged and it can be anything, but it's usually an aspect of electronic noise. See **Fig. 8** [2].

Figure 8: Noise in an image taken from a digital camera

Noise is an unwanted signal or an undesirable part of the image that we want to get rid of in order to properly process the image. However, not all noise is bad, sometimes it can be used to increase acutance to make an image apparently sharper in a technique called *dither*.

[2] $http://en.wikipedia.org/wiki/Image\_noise$

2.2 Investigate sort methods and implement

2.2.1 Bubble Sort

This method sorts an array of data by changing adjacent values and thus bringing the ones with greater value to the end of the vector.

The implementation is as follows:

```
void bubblesort(int vector[], int size) {
    bool switched = true;
    while (switched) {
        switched = false;
        for (int i = 0; i < size - 1; i++) {
            if (V[i] > V[i + 1]) {
                aux = V[i];
                V[i] = V[i + 1];
                V[i + 1] = aux;
                switched = true;
            }
        }
    }
}
```

2.2.2 Quick Sort

This method sorts an array of data by dividing and conquering. First, it's chosen an element in the middle of the array, called *pivot*, by adding the initial position in the array to the final position and dividing the result by 2. The array is then organized such as that all the elements to the left of the pivot is smaller than it and all the ones to its right is greater. The array is then divided into two parts, the one that's to the left of the pivot and the that's to the right. The last step is to do a recursion with the two arrays obtained until the array is completely sorted out.

The implementation is as follows:

```
void quicksort(int vector[], int inivet, int endvet) {
    int i = inivet;
    int j = endvet;
    int pivo = vector[(int) ((inivet + endvet) / 2)];
    int aux;
    while (i < j) {
        while (vector[i] < pivo) i++;
        while (vector[j] > pivo) j--;
        if (i <= j) {
```

```
            aux = vector[i];
            vector[i] = vector[j];
            vector[j] = aux;
            i++;
            j--;
        }
    }
    if (j > inivet) quicksort(vector, inivet, j);
    if (i < endvet) quicksort(vector, i, endvet);
}
```

2.3 Implementation of every filter

"Use Average, Median, Maximum, Minimum, Midpoint and Alpha-trimmed Mean filter to test noise images. Implement of multiply size mask and process the image boundary."

All of the images in this section are organized in the following way. From top left to bottom right: original, 3x3 filter, 5x5 filter and 7x7 filter.

2.3.1 Average

For this filter, you can use the OpenCV function *filter2D*. This function has 7 parameters: the original image; the result image; the depth that should be used for the new image, that's -1 if the depth should be the same as in the original image; the mask, or kernel; delta, which is a value to be added to each pixel during the convolution and is 0 by default; and the type of border. For each mask size, follow the example for a 3x3 mask:

```
cv::Mat mat = imread("TestFilter.png", CV_LOAD_IMAGE_GRAYSCALE);
Mat kernel, filter3x3;
cv::Point anchor = cv::Point(-1,-1);
double delta = 0;
int ddepth = -1;
int kernel_size = 3;
kernel = Mat::ones(kernel_size, kernel_size, CV_32F) /
        (float) (kernel_size*kernel_size);
filter2D(mat, filter3x3, ddepth, kernel, anchor, delta, BORDER_DEFAULT);
```

2.3.2 Median

This filter requires one to sort an array. This sorting function can be anything, from the bubble sort to the quick sort, as seen before. We'll call this function *sort*. For each mask size, follow the example for a 3x3 mask:

Figure 9: Average Filter

```
cv::Mat mat = imread("TestFilter2.png", CV_LOAD_IMAGE_GRAYSCALE);
Mat filter3x3; mat.copyTo(filter3x3);
int kernel_size = 3, kernelArea = kernel_size*kernel_size;
int middleIdx = (kernelArea-1)/2, radius = 1;
int vectorPixelsValue[9]; //kernel_size*kernel_size];
for (int i = 0; i < mat.rows; i++){
    for (int j = 0; j < mat.cols; j++){
        int cont = 0;
        for (int ki = i-radius; ki <= i+radius; ki++){
            for (int kj = j-radius; kj <= j+radius; kj++){
                if ((ki < 0 || kj < 0) ||
                    (ki >= mat.rows || kj >= mat.cols)) {
                    vectorPixelsValue[cont++] = 0;
                } else {
                    vectorPixelsValue[cont++] = mat.at<uchar>(ki,kj);
                }
            }
        }
        sort(vectorPixelsValue, kernelArea);
        filter3x3.at<uchar>(i,j) = vectorPixelsValue[middleIdx];
    }
}
```

Figure 10: Median Filter

2.3.3 Maximum

The difference between this filter and the Median filter is the last line of the code, since instead of choosing the median value, we choose the maximum value in the array. Since the array is already sorted, then the maximum value is the last value.

```
filter3x3.at<uchar>(i,j) = vectorPixelsValue[kernelArea-1];
```

Figure 11: Maximum Filter

2.3.4 Minimum

Like the Maximum filter, to implement this filter, only the last line of the code changes to choose the minimum, or the first, value.

```
filter3x3.at<uchar>(i,j) = vectorPixelsValue[0];
```

Figure 12: Minimum Filter

2.3.5 Midpoint

This filter calculates an average between the minimum and the maximum value in the sorted window of the image.

```
filter3x3.at<uchar>(i,j) = cvRound((vectorPixelsValue[0]+
    vectorPixelsValue[kernelArea-1])/2);
```

Figure 13: Midpoint Filter

2.3.6 Alpha-trimmed Mean

This filter calculates an average between all of the values in the sorted window of the image between α and $N - \alpha$. The last line of the code now becomes this:

```
for (int k = 0; k < 3; k++){
    alpha = k*2;
    int elementValue = 0;
    for (int vi = alpha; vi < kernelArea-alpha; vi++){
        elementValue += vectorPixelsValue[vi];
    }
    int N = kernelArea-2*alpha; //all elements from alpha to N-alpha-1
    filter3x3[k].at<uchar>(i,j) = cvRound(elementValue/N);
}
```

In **Fig. 14** it is shown the result of a 3x3 filter with α equals to 0, 2 and 4, respectively.

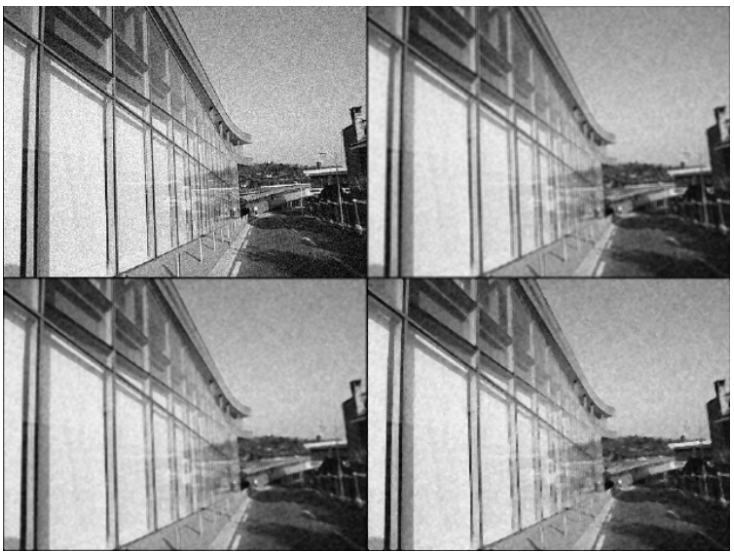


Figure 14: Original image and result images using α = 0, 2 and 4, respectively