WS 2019/20	Exercises Digital Image Processing	Solution Exercise No. 7

Simple Image Processing Operations

Goal:

General

- Sammary of neighboorhood operations in the spatial domain
- calculation of filtering in the spatial domain
- Boundary treatment methods
- Parctical aspects of filtering in the spatial domain

Solution for task 7.1a

Task:

Implement a function to perform a filtering in the spatial domain with any given filter mask! Note: Save the given filter mask in a form of an image (GrayImage) which has the same size of the filter mask!

Defining the interface:

```
// Filters the input image <input> with the filter mask <mask>. The result
// will be written in <output> . Boundary treatment by tiling the image.
void filter (GrayImage& input, GrayImage& output, GrayImage& mask)
```

Implementation:

```
//
// (6.1a)
//
//
void filter (GrayImage& input, GrayImage& output, GrayImage& mask)
{
    int     mwidth   = mask.getWidth();
    int     mheight  = mask.getHeight();
    float*  mdata    = mask.getData();

    int     mwidth2  = mwidth / 2;
    int     mheight2 = mheight / 2;

    int     iwidth   = input.getWidth();
    int     iheight  = input.getHeight();
    float*  idata    = input.getData();
    float*  odata    = output.getData();

    float   sum;

    for (int y=0; y<iheight; y++)
    {
        for (int x=0; x<iwidth; x++)
        {
            sum = 0.0;
            for (int t=0; t<mheight; t++)
            {
                for (int s=0; s<mwidth; s++)
                {

                    int y2 = mod(y+(t-mheight2), iheight);
                    int x2 = mod(x+(s-mwidth2), iwidth);

                    sum += mdata[t*mwidth+s] * idata[y2*iwidth+x2];
                }
            }
            odata[y*iwidth+x] = sum;
        }
    }
}
```

Defining the interface:

```
// Filters the input image <input> with the filter mask <mask>. The result
// will be written in <output> . Boundary treatment by copying or omission.
void filter2 (GrayImage& input, GrayImage& output, GrayImage& mask)
```

Implementation:

```
//
// (6.1a)
//
//
void filter2 (GrayImage& input, GrayImage& output, GrayImage& mask)
```

```
{
    int     mwidth   = mask.getWidth();
    int     mheight  = mask.getHeight();
    float*  mdata    = mask.getData();

    int     mwidth2  = mwidth / 2;
    int     mheight2 = mheight / 2;

    int     iwidth   = input.getWidth();
    int     iheight  = input.getHeight();
    float*  idata    = input.getData();
    float*  odata    = output.getData();

    float   sum;

    for (int y=mheight2; y<iheight-mheight2; y++)
    {
        for (int x=mwidth2; x<iwidth-mwidth2; x++)
        {
            sum = 0.0;
            for (int t=0; t<mheight; t++)
            {
                for (int s=0; s<mwidth; s++)
                {
                    int y2 = y+(t-mheight2);
                    int x2 = x+(s-mwidth2);

                    sum += mdata[t*mwidth+s] * idata[y2*iwidth+x2];
                }
            }
            odata[y*iwidth+x] = sum;
        }
    }
}
```

Results:

(a) (b) (c)

Figure 1: Results of the boundary treatment at a screen size of 21: tiled (a), image content copied (b) and omitted (c).

Solution for task 7.2b

Define a mask for the Gaussian filter in the spatial domain and describe the functions of this filter.

Implementation:

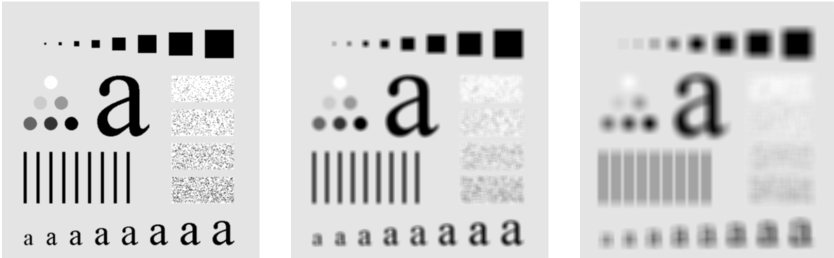
```
void createMask (MaskType maskType, GrayImage& mask)
{
    int     width   = mask.getWidth();
    int     height  = mask.getHeight();
    float*  data    = mask.getData();

    switch (maskType)
    {
        // 9er Box Filter
```

```
        case AVERAGING:
        {
            int     size  = mask.getSize();
            float   value = 1.0 / size;

            for (int i=0; i<size; i++)
            {
                data[i] = value;
            }
            break;
        }

        // 2D-Gauss-Filtermaske mit der Standardabweichung <sigma>
        case GAUSS:
        {
            float sigma;
            cout << "Sigma: ";
            cin >> sigma;

            int     x0      = width / 2;
            int     y0      = height / 2;
            float   factor1 = 1.0 / (2.0*M_PI*sigma*sigma);
            float   factor2 = -1.0 / (2.0*sigma*sigma);

            for (int y=0; y<height; y++)
            {
                for (int x=0; x<width; x++)
                {
                    int x1 = x-x0;
                    int y1 = y-y0;
                    data[y*width+x] = factor1 * expf(factor2 * (x1*x1 + y1*y1));
                }
            }
            break;
        }

        // 1D-Gauss-Filtermaske mit der Standardabweichung <sigma>
        case GAUSS1D:
        {
            float sigma;
            cout << "Sigma: ";
            cin >> sigma;

            int     x0      = width / 2;
            int     y0      = height / 2;
            float   factor1 = 1.0 / (sqrtf(2.0*M_PI)*sigma);
            float   factor2 = -1.0 / (2.0*sigma*sigma);

            for (int y=0; y<height; y++)
            {
                for (int x=0; x<width; x++)
                {
                    int x1 = x-x0;
                    int y1 = y-y0;
                    data[y*width+x] = factor1 * expf(factor2 * (x1*x1 + y1*y1));
                }
            }
            break;
        }
    }
}
```

Solution for task 7.2c

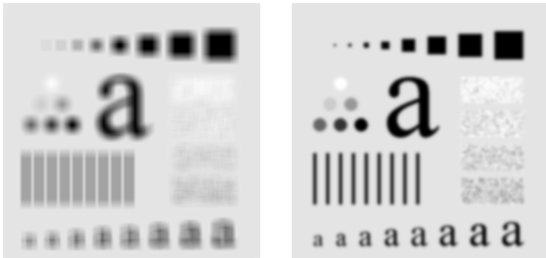
Compare the mean filter and Gaussian filter

The following images shows that the result of the mean and Gaussian filters with the same mask size but it is not necessarily to have the same degree of smoothing. The properties of the filter responses are equal.
The degree of smoothing in the Gaussian filter is determined by the standard deviation σ and not by the mask size. Smoothing results are approximately equal in strength when the mask



(a) (b) (c)

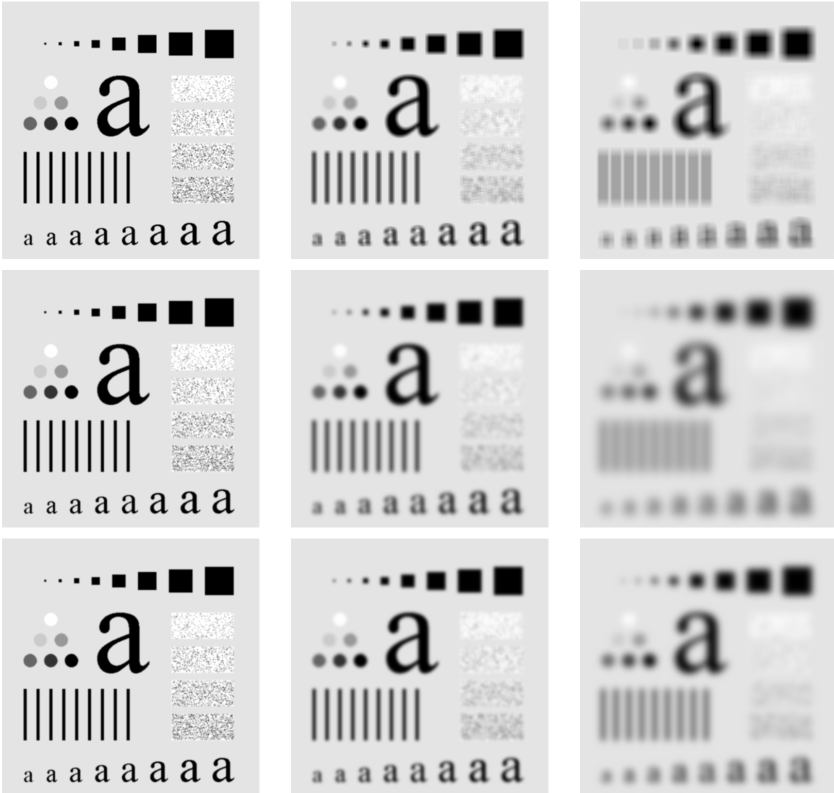
Figure 2: Results of a mean filter with sizes 3, 9 and 21.



(a) (b)

Figure 3: Comparison of mean and Gaussian filter with the same mask size. (a) Mean filter with mask size 21. (b) Gaussian filter with mask size 21 ($\sigma = 3$).

size of the mean filter is about twice to three times the size of the standard deviation of the Gaussian filter.



(a) (b) (c)

Figure 4: Top: Mean filter with sizes 3, 9 and 21. Middle: Gaussian filter with the standard deviations of 1, 4 and 10. Bottom: Gaussian filter with the standard deviations of 1, 3 und 7. (The mask sizes of the Gaussian filter in each case are $n = 6\sigma + 1$. It assumes that the calculation of (theoretically) unlimited Gaussian filter mask in the range $[-3\sigma, +3\sigma]$ is sufficient for the approximation.

Solution for task 7.2a

Task: *What is the idea behind the median filter? How does it works and what is output of the filter?*

The median filter sorts all gray values in the filter mask and replaces the center pixel by the *Median* (the mean value that found by sorting). Disruptive elements such as noise will be eliminated, because extreme gray values (eg, black or white) will be removed. In addition, a smoothing is performed without having to remove rough edges. Depending on the filter size edges will retain a certain size or eliminated.

Solution for task 7.2b

Implementation:

```
#include "dip.h"

enum SortType
{
    SELECTIONSORT = 1,
    BUBBLESORT    = 2,
    QUICKSORT     = 3
};
```

```
void exchange (float& value1, float& value2)
{
    float temp = value1;
    value1     = value2;
    value2     = temp;
}

int minIndex (float* data, int startIdx, int size)
{
    int minIdx = startIdx;
    for (int i=startIdx+1; i<size; i++)
    {
        if (data[i] < data[minIdx]) minIdx = i;
    }
    return minIdx;
}
void selectionsort (float* data, int size)
{
    int minIdx;
    for (int i=0; i<size; i++)
    {
        minIdx = minIndex(data, i, size);
        exchange(data[i], data[minIdx]);
    }
}
void bubblesort (float* data, int size)
{
    bool sortiert = false;
    while (!sortiert)
    {
        bool getauscht = false;
        for (int i=0; i<size-1; i++)
        {
            if (data[i] > data[i+1])
            {
                exchange (data[i], data[i+1]);
                getauscht = true;
            }
        }
        if (!getauscht) sortiert = true;
    }
}
void quicksort(float* data, int leftIdx, int rightIdx)
{
    int   l     = leftIdx;
    int   r     = rightIdx;
    float pivot = data[rightIdx];
    if (l < r)
    {
        while (l<=r)
        {
            while (data[l] < pivot) l++;
            while (data[r] > pivot) r--;
            if (l<=r)
            {
                exchange(data[l], data[r]);
                l++;
                r--;
            }
        }
        quicksort(data, leftIdx, r);
        quicksort(data, l, rightIdx);
    }
}

void sort (GrayImage& input, GrayImage& output, int xpos, int ypos,
           const int xsize, const int ysize, SortType sortType)
{
    int    width  = input.getWidth();
    float* idata  = input.getData();
    float* odata  = output.getData();

    int    links  = xpos - xsize/2;
    int    oben   = ypos - ysize/2;
    int    rechts = xpos + xsize/2;
```

```
    int     unten  = ypos + ysize/2;

    float* sortdata = new float[xsize*ysize];

    int     count = 0;
    for (int y=oben; y<=unten; y++)
    {
        for (int x=links; x<=rechts; x++)
        {
            sortdata[count] = idata[y*width+x];
            count++;
        }
    }

    switch (sortType)
    {
        case SELECTIONSORT:
            selectionsort(sortdata, count);
            break;
        case BUBBLESORT:
            bubblesort(sortdata, count);
            break;
        case QUICKSORT:
            quicksort(sortdata, 0, count-1);
            break;
    }
    odata[ypos*width+xpos] = sortdata[count/2];
    delete[] sortdata;
}
void filter (GrayImage& input, GrayImage& output, int xsize, int ysize, SortType sortType)
{
    int     width    = input.getWidth();
    int     height   = input.getHeight();

    for (int y=ysize/2; y<height-ysize/2; y++)
    {
        for (int x=xsize/2; x<width-xsize/2; x++)
        {
            sort(input, output, x, y, xsize, ysize, sortType);
        }
    }
}

// ---------------------------------------------------------------------------

int main (int argc, char** argv)
{
    /* ********************************** */
    /* Uebung 06                          */
    /* ********************************** */

    /* ********************************** */
    /* Eingabe                            */
    /* ********************************** */

    cout << "Folgende Funktionen waehlen:\n";
    cout << "1 - Selectionsort anwenden\n";
    cout << "2 - Bubblesort anwenden\n";
    cout << "3 - Quicksort anwenden\n";

    int choice;
    cin >> choice;

    string filename;
    cout << "Bildname: ";
    cin >> filename;

    int xsize, ysize;
    cout << "Filtergroesse in x (ungerade): ";
    cin >> xsize;
    cout << "Filtergroesse in y (ungerade): ";
    cin >> ysize;

    // Abfangen, falls Filtergroesse nicht stimmt
    if (xsize%2 == 0 || ysize%2 == 0)
    {
        cout << "Filtergroesse nicht ungerade!\n";
```

```
        cout << "FINISHED.\n";
        return 0;
    }

    GrayImage input;
    input.load(filename);

    /* ************************************ */
    /* Datenverarbeitung / Funktionsaufruf  */
    /* ************************************ */

    // Bildinhalt kopieren, damit ist die Randbehandlung erledigt
    GrayImage result = GrayImage(input.getWidth(), input.getHeight());

    switch (choice)
    {
        case 1:
            cout << "Selectionsort gestartet - ";
            filter (input, result, xsize, ysize, SELECTIONSORT);
            cout << "fertig" << endl;
            break;
        case 2:
            cout << "Bubblesort gestartet - ";
            filter (input, result, xsize, ysize, BUBBLESORT);
            cout << "fertig" << endl;
            break;
        case 3:
            cout << "Quicksort gestartet - ";
            filter (input, result, xsize, ysize, QUICKSORT);
            cout << "fertig" << endl;
            break;
    }

    /* ************************************ */
    /* Ausgabe                              */
    /* ************************************ */

    input.show();
    result.show();
    result.save();

    /* ************************************ */
    /* Programmende                         */
    /* ************************************ */
    cout << "FINISHED.\n";
    return 1;
}
```

Task: *Test your implementations for images lena_gauss.bmp and lena_int.bmp.*

The image *lena_int.bmp* contains intensity noise which is characterized by black and white Stï¿œrpixel (noisy pixel). Therefore, these must be removed. The image *lena_gauss.bmp* has the Gauss noise, that means it contains normally distributed grey valued Stï¿œrpixel (noisy pixel). In case the image contains the black border, one can use the edge detection.

Figure 5: (a): Original image: *lena_int.bmp*, (d): Original image: *lena_gauss.bmp*, (b) and (e): Median filter responses are greater 3×3, (c) and (f): Median filter responses 9×9.