Digital Image Processing
Lab 4

Group 25:
Henry Maathuis (S2589540)
Sebastian Wehkamp (S2589907)

January 14, 2018

Contents

1 Exercise 1

1.1 Partial reconstruction

This exercise is a continuation of exercise 2 of the previous lab in which we created a reconstruction form the Laplacian pyramid. This is done by creating a function *IPpyr_recon(g, J, K, sigma)* in which $g, J, sigma$ are the reconstruction parameters and K is partial reconstruction level. The function reads the pyramid and converts the pyramid to separate images which are stored in a list. Based on the reconstruction level K a number of difference levels are set to zero since these are not to be added. The last step is to expand the image for every level and add the differences. This can be safely done since the difference levels which are not needed are set to zero. The result is an image of the same size as the original image with partial reconstruction level K. The final version of the code can be seen in Listing 1.

```
% INPUT: laplacian pyramid g, level J, sigma s
% OUTPUT: reconstructed image im

function [im] = IPpyr_PartRecon(g, J,K, s)
    g = im2double(g);
    [M, N] = size(g);

    current_height = 1;
    sz = N;

    center = N / 2;
    G = {};

    % Convert the pyramid to a list of images G
    for idx = 1:J
        height_pos = current_height:current_height + sz - 1;
        width_pos = center - (0.5 * sz) + 1 : center + (0.5 * sz);
        G{idx} = g(height_pos, width_pos);
        current_height = current_height + sz;
        sz = sz / 2;
    end

    % Set starting image to bottom level of pyramid
    im = G{J};

    % Set levels to zero
    if K~=0
        for j=1:K
            G{j} = 0;
        end
    end

    % For every level expand the image and add the differences
    for idx = 1:J-1
        im = IPexpand(im, s) + G{J-idx};
    end
```

```
37
38      % Scale values between 0 and 1
39      minV = min(im(:));
40      maxV = max(im(:));
41
42      im = im - minV;
43      im = im / (maxV - minV);
44
45      imshow(im);
46  end
```

Listing 1: Function which computes the partial reconstruction.

1.2 Partial reconstruction test script

The goal of this exercise was to create a test script which tests Listing 1 and displays the results. After reading the image the test script creates a LaPlacian pyramid g of the image. From this pyramid g a partial reconstruction *g2* is made. Lastly the differences are computed and all images are shown in the same figure. These results for reconstruction level K = 2 can be seen in Figure 1. The error using K = 2 is equal to 0.0494.

Figure 1: Comparison between the original image, partial reconstruction, and difference image.

```
1  % Specify reconstruction level
2  K = 2;
3
4  % Read original image
5  image = imread('plant.tif');
6  image = im2double(image);
7
8  % Apply scaling to original image
9  minV = min(image(:));
10 maxV = max(image(:));
11 image = image - minV;
12 image = image / (maxV - minV);
13
14 % Decompose the image
15 g = IPpyr_decomp(image, 3, 1);
16
```

```
% Reconstruct the image with specified level K
g2 = IPpyr_PartRecon(g, 3, K, 1);

% Compute difference image
dif = abs(g2 - image);

% Plot images properly
subplot(1,3,1);
imshow(image);
subplot(1,3,2);
imshow(g2);
subplot(1,3,3);
colormap(gray);
imagesc(dif);
axis off;

% Compute the error
sum_diff = 0;

for i = 1:N
    for j = 1:N
        diff = abs(g2(i, j) - image(i, j));
        sum_diff = sum_diff + diff;
    end
end

error = sum_diff / (M * N)
```

Listing 2: Test script used for testing the Partial Reconstruction function shown in Listing 2

1.3 Test for different levels of K

The results for reconstruction level K = 1 can be found in Figure 2. The error for K = 1 is equal to 0.0266. In the difference image you can see that the reconstruction using K=1 is better than the reconstruction of K=2 since the shapes in the difference image are more clear. The error measure of K=1 is about half of the error of K=2 which also indicates that the reconstruction K = 1 is better. This is in line with the expectations since a reconstruction level of k=1 displays more information than K=2.

Figure 2: Comparison between the original image, partial reconstruction, and difference image.

2 Exericse 2

2.1 Dilation

For this exercise we had to implement the binary dilation operation with an arbitrary structuring element. To do this we used the duality between dilation and erosion. This means that the dilation of an image is equal to an erosion on the complement of the original image with a 180 degrees rotated structuring element. So all we have to do is calculate the complement of the original image, rotate the structuring element, perform the erosion, and calculate the complement again on the image obtained after erosion. The implementation can be seen in Listing 3.

```
function [dilation] = IPdilate(image,structEl)
    % Similar implementation but now using the duality property.
    % This states that dilating is the same as performing erosion with
    % an inverted image and a rotated (180 degrees) structuring element
    image_comp = imcomplement(image);
    structEl_rot = rot90(structEl, 2);

    dilation = IPerode(image_comp, structEl_rot);

    % Invert image back to original "color"
    dilation = imcomplement(dilation);
end
```

Listing 3: Function that performs a binary dilation with an arbitrary 3x3 structuring element.

2.2 Erosion

For this exercise we had to implement the binary erosion operation with an arbitrary structuring element. We assume that the origin of the structuring element lies in the center, and that binary images have type logical in Matlab. The implementation of our erosion is given in Listing 4. This function pads the image with ones in order to not influence the erosion. Then we start shifting the structuring element over the padded image. For every shift we check the equality between the structuring element and the corresponding part of the image. Since we only want to check if the image has ones on the same place as the structuring element we perform a point wise multiplication.

```
erosion(i,j) = isequal(structEl , padded  .*  structEl);
```

This means that whenever the structuring element has a 0 on a pixel location, it does not matter whether the input image has a 0 or a 1 on that location. Using this point wise multiplication we only look whether the image has ones on the same place as the structuring element. If this is the case, we set the center to one.

```
function [erosion] = IPerode(image,structEl)
    [M,N] = size(image);

    % Pad with 1s since we want to match as much as possible with our
    % structuring element (eroding). This is also Matlabs convention.
    paddedIm = padarray(image,[1,1], 1);
    erosion = zeros(M,N);
    for i=1:M
```

```
9           for j=1:N
10              padded = paddedIm(i:i+2,j:j+2);
11              erosion(i,j)= isequal(structEl, padded .* structEl);
12          end
13      end
14  end
```

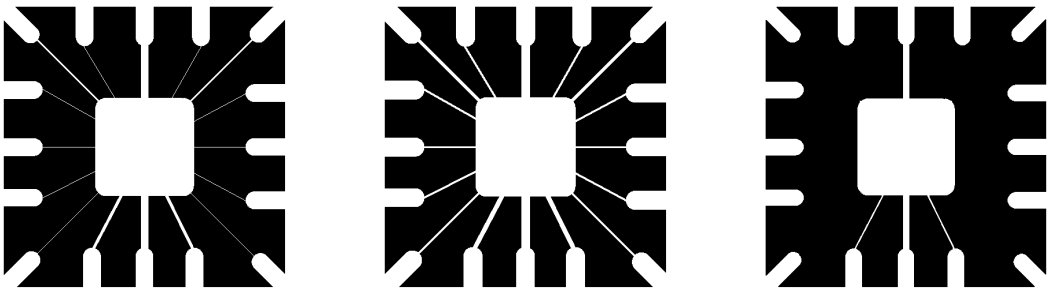
Listing 4: Function that performs a binary erosion using the duality of erosion and dilation.

2.3 Testing with wirebondmask.tif

We tested our erosion and dilation functions using a mask consisting completely out of ones and by using a non-symmetric mask shown in Table 1. The results for dilation and erosion with constant masks is shown in Figure 3. The output of the function using the mask shown in Table 1 is displayed in Figure 4. For both functions we compared our results with the built-in Matlab function and the outputs are exactly the same.

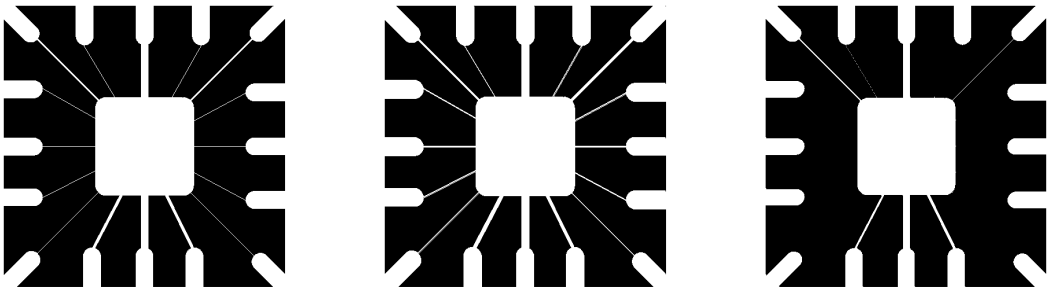
Table 1: non-symmetric mask

1	0	0
1	0	0
0	1	0



(a) Original image. (b) Dilation with constant mask. (c) Erosion with constant mask.

Figure 3: Comparison between the original image, dilation, and erosion with constant mask.



(a) Original image. (b) Dilation with constant mask. (c) Erosion with constant mask.

Figure 4: Comparison between the original image, dilation, and erosion with a non-symmetric mask.

3 Exercise 3

3.1 Grayscale morphology implementation

In this exercise we had to implement both erosion and dilation again but this time for grayscale images instead of binary images. Erosion can be implemented by doing as the book states; in order to find the erosion of a grayscale image you have to find the minimum of each value in the region image coincident with the structuring element. This is similar to finding the dilation of a grayscale image only to find the dilation you have to find the maximum value in the region image coincident with the structuring element. The formulas necessary for this implementation are shown in Equation 1 and Equation 2. In our implementation we made use of the duality of erosion and dilation as described later on. The dilation function for grayscale images is listed in Listing 5 and the function performing erosion on gray scale images is listed in Listing 6. Similarly to the binary case we have to multiply the structuring element point wise with the corresponding input region. This means that we only look at the pixel values of the input image where the structuring element is 1.

$$[f \ominus](x,y) = min_{(s,t)\in b_N}[f(x+s, y+t) - b_N(s,t)] \tag{1}$$

Figure 5: Formula for erosion.

$$[f \oplus](x,y) = max_{(s,t)\in b_N}[f(x-s, y-t) - b_N(s,t)] \tag{2}$$

Figure 6: Formula for dilation.

```
function [dilation] = IPgdilate(image,structEl)
    [M,N] = size(image);
    image = im2double(image);
```

```
4
5       % Pad image by 1 row and column on each side.
6       % Needed since the origin is in center of image
7       % Padding with 1 is necessary since we intend to dilate
8       % This means that we would like to match as much as possible
9       paddedIm = padarray(image,[1,1], 1);
10      dilation = zeros(M,N);
11
12      for i=1:M
13          for j=1:N
14              % As described in the book we need to take the maximum
15              % value of each value in the region image coincident
16              % with the structuring element.
17              padded = paddedIm(i:i+2,j:j+2);
18              max_elem = max(max(padded .* structEl));
19              dilation(i,j) = max_elem;
20          end
21      end
22  end
```

Listing 5: Function that performs a grayscale dilation.

We can compute the grayscale erosion also in terms of dilation. To perform erosion we can do dilation on the complement of the image with a structuring element which is rotated 180 degrees. Afterwards we take the complement again to obtain our final result. Note that this is similar to how we computed dilation in the binary case. The grayscale erosion operation is implemented in the listing below.

```
1   function [erosion] = IPgerode(image,structEl)
2       % Similar implementation but now using the duality property.
3       % This states that eroding is the same as performing dilation with
4       % an inverted image and a rotated (180 degrees) structuring element
5       image_comp = imcomplement(image);
6       structEl_rot = rot90(structEl, 2);
7
8       erosion = IPgdilate(image_comp, structEl_rot);
9
10      % Invert image back to original "color"
11      erosion = imcomplement(erosion);
12  end
```

Listing 6: Function that performs a grayscale erosion.

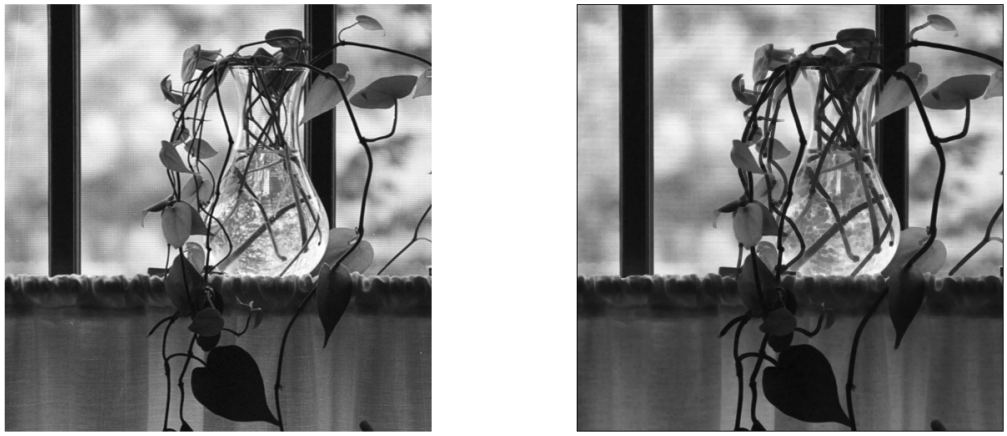
3.2 Testing with vase.tif

In this exercise we have to test the functions created above. This is done by performing a dilation, erosion, opening, and a closing on the grayscale images. As a mask we should use a box, so a 3x3 matrix consisting of all ones. First we performed an erosion and a dilation on vase.tif. The dilation results are shown in Figure 7 and the erosion results are displayed in Figure 8. The results for the opening and closing are displayed in Figure 9 and Figure 10 respectively. Note that closing is implemented by first performing dilation and then erosion on an image. Opening is implemented

by first performing erosion after which dilation is performed. The code used to generate the images for each operation is given in Listing 7, Listing 8, Listing 9 and Listing 10.

(a) Original image vase.tif (b) Grayscale dilation of vase.tif

Figure 7: Comparison between the original image and the dilation.

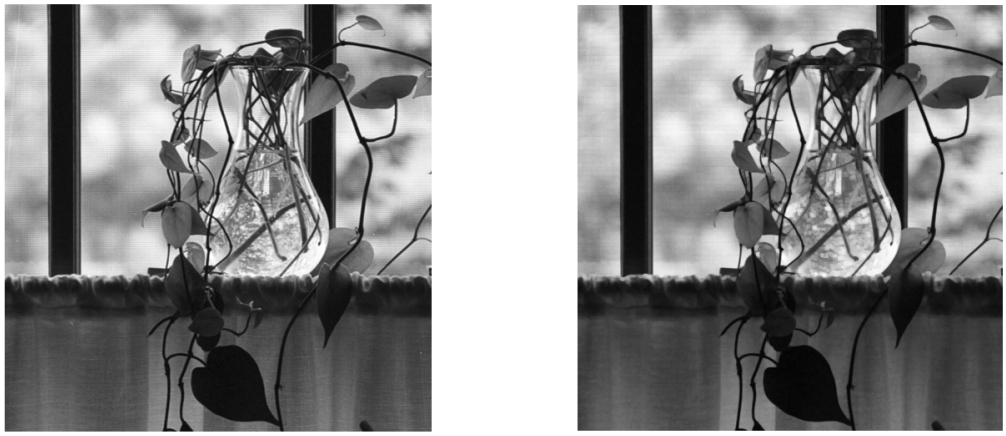


(a) Original image vase.tif (b) Grayscale erosion of vase.tif

Figure 8: Comparison between the original image and the erosion.

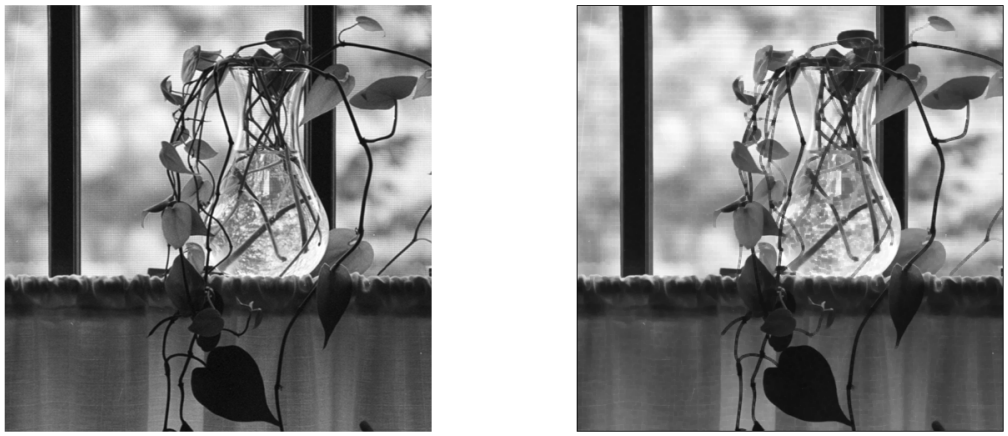
When looking at the results of erosion and dilation we observe that in the case of a dilation some of the bright parts got even brighter and larger. This can be clearly seen in the vase itself

in Figure 7b. The opposite is the case for the erosion in which some of the darker parts got a bit darker and larger as can be seen in Figure 8b. In Figure 8b the darker particles in the vase became larger. In the case of dilation we can also see that thin black structures (i.e. twigs) get thinner due to this operation. In the case of erosion we see that the twigs get thicker.



(a) Original image vase.tif (b) Grayscale opening of vase.tif

Figure 9: Comparison between the original image and the opening.



(a) Original image vase.tif (b) Grayscale closing of vase.tif

Figure 10: Comparison between the original image and the closing.

In Figure 9 you can see that some small bright peaks have been removed while still maintaining the overall brightness in the vase. Since opening is performed by first doing erosion and afterwards dilation, we see that small white structures are removed. If you look really closely you can see that a white stripe is removed in the rightmost leaf. The closing does the opposite, by removing small dark spots. The results of the closing can also be seen in the rightmost leaf. The darker line across the leaf has been removed as a result of the closing. Note that the difference is hard to see in the images in the report but on originals the difference is much clearer. A larger structuring element can help to visualise the differences even better.

4 Appendix

```
img = imread('vase.tif');

% Show original image
figure;
imshow(img);

% Define a structuring element
struct_elem = ones(3, 3);

g = IPgerode(img, struct_elem);

% Normalization (force values to be within 0 and 1)
minV = min(g(:));
maxV = max(g(:));

g = g - minV;
g = g / (maxV - minV);

% Visualize result
figure;
imshow(g);
```

Listing 7: Script that computes an erosion on the vase image.

```
img = imread('vase.tif');

% Visualize original image
figure;
imshow(img);

% Define a structuring element
struct_elem = ones(3, 3);

g = IPgdilate(img, struct_elem);

% Normalization (force values to be within 0 and 1)
minV = min(g(:));
maxV = max(g(:));
```

```
15
16 g = g - minV;
17 g = g / (maxV - minV);
18
19 % Visualize result
20 figure;
21 imshow(g);
```

Listing 8: Script that computes a dilation on the vase image.

```
1  img = imread('vase.tif');
2
3  % Show original image
4  figure;
5  imshow(img);
6
7  % Define a structuring element
8  struct_elem = ones(3, 3);
9
10 g = IPgerode(img, struct_elem);
11 g = IPgdilate(g, struct_elem);
12
13 % Normalization (force values to be within 0 and 1)
14 minV = min(g(:));
15 maxV = max(g(:));
16
17 g = g - minV;
18 g = g / (maxV - minV);
19
20 % Visualize result
21 figure;
22 imshow(g);
```

Listing 9: Script that computes an opening on the vase image.

```
1  img = imread('vase.tif');
2
3  % Show original image
4  figure;
5  imshow(img);
6
7  % Define a structuring element
8  struct_elem = ones(3, 3);
9
10 g = IPgdilate(img, struct_elem);
11 g = IPgerode(g, struct_elem);
12
13 % Normalization (force values to be within 0 and 1)
14 minV = min(g(:));
15 maxV = max(g(:));
16
```

```
g = g - minV;
g = g / (maxV - minV);

% Visualize result
figure;
imshow(g);
```

Listing 10: Script that computes a closing on the vase image.

5 Code previous exercises

```
function [output_img] = IPdownsample(img, fac)
    img = im2double(img);
    [m, n] = size(img);
    %Create list of indices with steps of size fac
    m_ind = 1:fac:m;
    n_ind = 1:fac:n;

    %Using list of indices place the original pixels in a new image
    output_img = img(m_ind, n_ind);
end
```

Listing 11: Function that downsamples an image

```
function [g] = IPexpand(f, sigma)
    g = imgaussfilt(IPzoom(f, 2), sigma);
end

%%mask = fspecial('gaussian',3,sigma);
    %g = IPfilter(IPzoom(f, 2), mask);
    %g = imresize(imgaussfilt(f, sigma), 0.5);
    %g = im2double(g);
```

Listing 12: Function which applies Gaussian filtering

```
% INPUT: image f, level J, sigma s
% OUTPUT: resulting laplacian pyramid image g,
% gaussian pyramid images G,
% laplacian images L

function [g, G, L] = IPpyr_decomp(f, J, s)
    f = im2double(f);
    M = size(f, 1);

    sum = 0;

    % Determine the height of the resulting image
    for idx = 1:J
        sum = sum + (0.5^(idx - 1));
    end

```

```
    P = M * sum;

    % Construct the matrix
    g = ones(P, M);

    % Gaussian pyramid
    G{1} = f;
    for idx = 1:J - 1
        f = IPreduce(f, s);
        G{idx + 1} = f;
    end

    % Use the size of the output image to determine where
    % each L_n image should be placed
    [g_h, g_w] = size(g);
    offsetY = 0;

    % Expand the guassian images and subtract it from the original G_n
    for idx = 1:J
        if (idx == 1)
            f =  G{J - idx + 1};
        else
            G{J - idx + 1}
            IPexpand(G{J - idx + 2}, s)
            f =  G{J - idx + 1} - IPexpand(G{J - idx + 2}, s);
        end

        % Store Laplacian images
        L{idx} = f;

        [p, q] = size(f);
        offsetY = offsetY + p;

        % Compute where the image should be placed in the resulting image
        height =  g_h - offsetY + 1 : g_h - offsetY + p;
        width = g_w / 2 - 0.5 * p + 1: g_w / 2 + 0.5 * p;
        g(height, width) = f;
    end

    % Scale values between 0 and 1
    minV = min(g(:));
    maxV = max(g(:));

    g = g - minV;
    g = g / (maxV - minV);
end
```

Listing 13: Function which creates a pyramid decomposition

```
function [g] = IPreduce(f, sigma)
    g = IPdownsample(imgaussfilt(f, sigma), 2);
```

```
end

%    %mask = fspecial('gaussian',3,sigma);
   % g = imresize(imgaussfilt(f, sigma), 0.5);
    %g = imresize(IPfilter(f, mask), 0.5);
    %g = im2double(g);
```

Listing 14: Function which downsamples an image

```
function [output_img] = IPzoom(img, fac)
    img = im2double(img);
    %Replicate a pixel using the repelem function
    output_img = repelem(img, fac, fac);
end
```

Listing 15: Functon which zooms in on an image using pixel replication.