Digital Image Processing
Lab 5

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

January 21, 2018

Contents

1 Exercise 1

1.1 Compute the gradient image

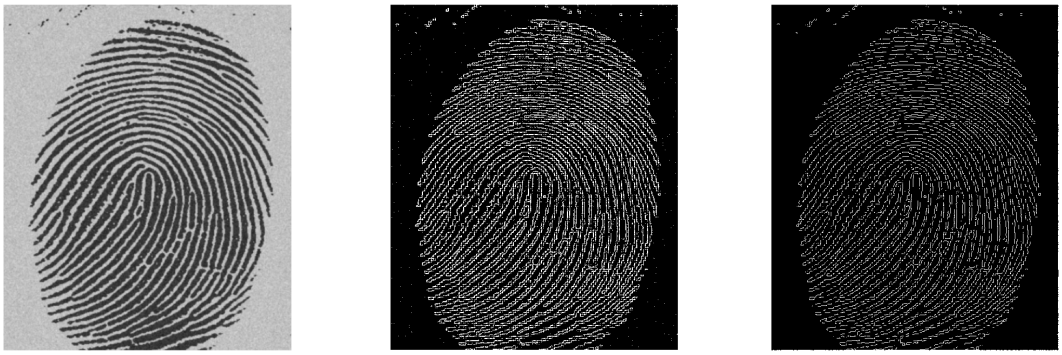
In this exercise we had to create a function *IPgradientthresh(f,p)* which computes the gradient image $g = |fx| + |fy|$ for an input image f. On the resulting image a threshold should be applied. The first thing we have to do is to pad the image with ones on the right and bottom. We decided to pad with ones since the edge values already have relatively large values. Padding with zeros would result in subtracting zero from the already high value thus messing up with the scaling of the threshold value. Padding with ones solves this problem partially however there are still high values at the edges. After padding we compute for every pixel $|fx|$ by subtracting the current pixel from the pixel on the right. The same is done for $|fy|$ only with the pixel below. By adding $|fx|$ and $|fy|$ for each pixel we obtain g. After this operation, we scale the pixels to be within the range 0 to 1. Then the only thing left to do is apply the threshold to create a binary image. The final version of the code is listed in Listing 1.

```
function [g] = IPgradientthresh(f,p)
    f = im2double(f);
    [M, N] = size(f);
    % Create an all ones matrix
    padded = ones(M + 1, N + 1);
    % Create a padded version of the image
    padded(1:M, 1:N) = f;
    % allocate space for the new image
    g = zeros(M, N);
    for i = 1:M
        for j = 1:N
            % Compute |fx|
            abs_x = abs(padded(i + 1, j) - padded(i, j));
            % Compute |fy|
            abs_y = abs(padded(i, j + 1) - padded(i, j));
            % Compute g
            g(i, j) = abs_x + abs_y;
        end
    end
    % Normalization (force values to be within 0 and 1)
    minV = min(g(:));
    maxV = max(g(:));

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

    % Compute threshold
    p = p/100*max(max(g));

    % Apply threshold
    g(g < p) = 0;
    g(g >= p) = 1;
end
```

Listing 1: Function which computes a thresholded gradient image.

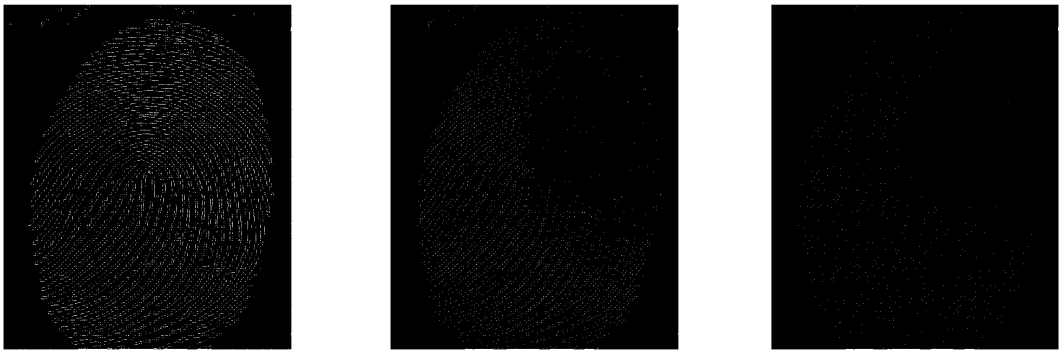
1.2 Testing with fingerprint.tif

In this exercise we have to test the above created function listed in Listing 1 on fingerprint.tif for varying threshold values. In Figure 1 and Figure 2 you can clearly see that when we increase the threshold the lines get thinner. From 20% on wards, as seen in Figure 1c a lot of details are removed and the quality decreases. The 30% image in Figure 2a still contains some sort of recognisable fingerprint but with a lot less details. Starting at 40% all details are lost as seen in Figure 2. A test script can be found in Listing 6.



(a) Original image (b) 10% (c) 20%

Figure 1: Images obtained by applying Listing 1 to fingerprint.tif.



(a) 30% (b) 40% (c) 50%

Figure 2: Images obtained by applying Listing 1 to fingerprint.tif.

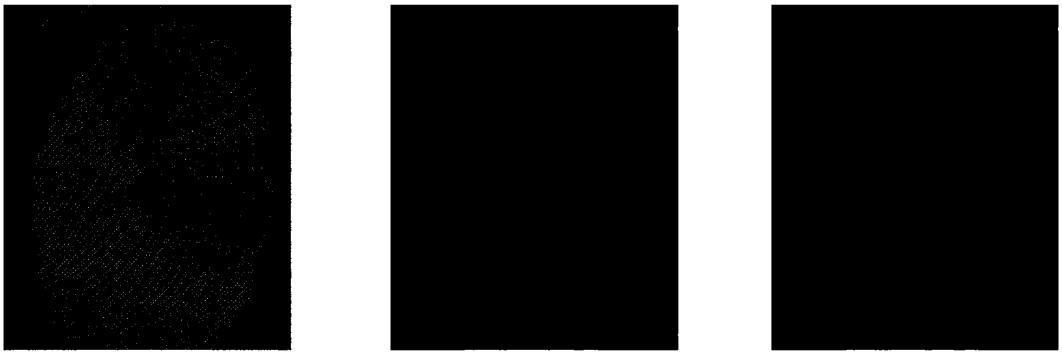
1.3 Smooth image first

In Figure 3 and Figure 4 you can see the results of applying Listing 1 on a uniformly smoothed 'fingerprint.tif'.

In Figure 3 and Figure 4 you can see the results of applying Listing 1 on a uniformly smoothed 'fingerprint.tif'. We smoothed the image using a uniform 3x3 mask consisting of ones. The results for 20% as seen in Figure 3c seem to be a bit clearer than Figure 1c. However when we smooth the image pretty much all details are gone after 30% thresholding while the image without smoothing does have some details left. A test script can be found in Listing 6.

(a) Smoothed image (b) 10% (c) 20%

Figure 3: Images obtained by applying Listing 1 to a uniformly smoothed fingerprint.tif.



(a) 30% (b) 40% (c) 50%

Figure 4: Images obtained by applying Listing 1 to a uniformly smoothed fingerprint.tif.

2 Exercise 2

2.1 Implement the Marr-Hildreth edge detector

In this assignment we implemented the Marr-Hildreth edge detector. We follow the implementation as listed in the book. The detector can be described in three steps.

a. Filter the image using a Gaussian lowpass filter

b. Compute the Laplacian of the image

c. Find zero crossings in the image.

The first step is filtering which is needed because the 2nd derivative is very sensitive for noise. This noise must be filtered out which can be done using a uniform filter. The second step is to compute the mask using the equation below and perform a convolution with the mask and the image. The convolution is performed as a multiplication in frequency space. The last step is, as the assignment states, threshold the resulting image at p% of the maximum value in the image. The final version of our code can be seen in Listing 2.

$$K_\sigma(x,y) = -\frac{2}{\pi\sigma^4}(1-\frac{r^2}{2\sigma^2})e^{\frac{-r^2}{2\sigma^2}} \\ (r^2 = x^2+y^2) \tag{1}$$

```
function [g] = IPMarrHildreth (f, sigma, p)
    % Apply uniform filtering
    f = IPfilter(f, 1/9 * ones(3,3));
    % Zero padding is applied to avoid boundary overlap effects
    M=size(f,1); N=size(f,2); % nr of rows/columns of image f
    C = 6 * sigma; D = 6 * sigma; % nr of rows/columns of kernel h

    if mod(C, 2) == 0
        C = C + 1;
        D = D + 1;
    end
    P=M+C-1; Q=N+D-1; % nr of rows/columns after padding
    fp=zeros(P,Q); % zero padding: start with zeroes
    fp(1:M,1:N)=f; % insert f into image fp
    hp=zeros(P,Q); % Construct filter matrix hp, same size as fp.

    % Compute mask with formula
    for i = 1:C
        for j = 1:D
            x = -ceil(C / 2) + i;
            y = -ceil(D / 2) + j;
            LoG_val = -2.0 / (pi * sigma^4) ...
                * (1.0 - ((x * x + y * y) / (2.0 * sigma^2))) ...
                * exp((-(x * x + y * y) / (2.0 * (sigma^2))));
            hp(i, j) = LoG_val;
        end
    end

    % Shift mask over image
    hp = circshift(hp, [-floor(C / 2), -floor(D / 2)]);

    % Convert to frequency space for efficient convolution
```

```
33      Fp=fft2(double(fp), P, Q); % FFT of image fp
34      Hp=fft2(double(hp), P, Q); % FFT of kernel hp
35
36      % Perform convofultion
37      Gp = Hp .* Fp;
38
39      % Return to spatial domain
40      gp=ifft2(Gp); % Inverse FFT
41      gp=real(gp); % Take real part
42      g=gp(1:M, 1:N); % Undo zero padding
43
44      % Compute threshold
45      p = p/100*max(max(g));
46
47      % Apply threshold
48      g(g < p) = 0;
49      g(g >= p) = 1;
50
51      % Scale image back
52      minV = min(g(:));
53      maxV = max(g(:));
54
55      g = g - minV;
56      g = g / (maxV - minV);
57
58      % Plot the image
59      imshow(g);
60  end
```

Listing 2: Function which implement the Marr Hildreth edge detector

2.2 Testing the Marr-Hildreth edge detector

We tested our Marr-Hildreth edge detector by applying it to 'house.tif' with varying thresholds and a sigma of 1.0 and 5.0. First we will discuss the results of sigma 1.0. The edge detector seems to do a good job although with a threshold of 10% there is a bit too much noise in the image. A threshold of around 20% to 30% seems to be optimal. There is little noise in those images while still maintaining the important edges. Using higher thresholds results in losing important edges. Next up we tested the edge detector with sigma 5.0. We now observe that the edges are much thicker than sigma 1.0 as expected and there is way less noise in the images. The general outline of the house remains sort of clear until 40% but 20% seems to be the best threshold for sigma 5. The results of Sigma 1 however are way better, the edges are smaller and closely resemble the size of the edges in the original image. A test script can be found in Listing 8.

(a) 10% (b) 20% (c) 30%

Figure 5: Images obtained by applying Listing 2 to house.tif with varying thresholds and sigma 1.

(a) 40% (b) 50%

Figure 6: Images obtained by applying Listing 2 to house.tif with varying thresholds and sigma 1.

(a) 10% (b) 20% (c) 30%

Figure 7: Images obtained by applying Listing 2 to house.tif with varying thresholds and sigma 5.

(a) 40% (b) 50%

Figure 8: Images obtained by applying Listing 2 to house.tif with varying thresholds and sigma 5.

3 Exercise 3

3.1 Implement contour tracing

In this exercise we have to implement a function IPcontour which takes a binary image containing a single object as input, determines the starting point, and uses bwtraceboundary to trace the contour. The first thing we have to do is to determine the starting point which is done in the function given in Listing 4. It simply loops over the binary image and returns the location of the first pixel whose value is a logical 1. The next step is to trace the boundary starting in the previously found starting point. The resulting boundary is displayed and the coordinates of points on the boundary are returned. The final function is shown in Listing 3. Due to us misreading the assignment we also implemented IPbwtraceboundary ourselves. The file (IPbwtraceboundary.m) can be found in the zip file if you are interested.

```
% Function tracing countr of object
function [X, Y] = IPcontour (f)
    % Calculate starting point and show results as requested
    [X,Y] = IPstartingpoint(f)
    % Trace boundary
    res = bwtraceboundary(f, [X,Y], 'W');
    imshow(f)
    hold on;
    Y = res(:,1);
    X = res(:,2);

    % Plot the contour, note that the color and linewidth can be tweaked
    % below
    plot(X,Y,'g','LineWidth',1);
end
```

Listing 3: Function implementing countour tracing algorithm

```
% Determine starting location
function [x, y] = IPstartingpoint (f)
```

```
    [M, N] = size(f);
    for i = 1:M
        for j = 1:N
            if f(i, j) == 1
                x = i;
                y = j;
                return
            end
        end
    end
end
```

Listing 4: Function which determine the starting point for tracing the contour

3.2 Test countour tracing on lincoln.tif

Figure 9 shows the results when tracing the contour on the lincoln image. The starting point we found was ($x = 15$, $y = 83$). We observe that the algorithm effectively captured the contour of Lincoln. We adapted the thickness of the line for visualisation purposes, obviously the boundary we obtained has a thickness of 1 pixel. A test script can be found in Listing 9.

Figure 9: Contour tracing on Lincoln.

3.3 Implement fourier descriptor scheme

In this exercise we had to implement the fourier descriptor scheme. our function accepts an array of boundary points and the number of Fourier descriptors, P, to retain. First we create a contour

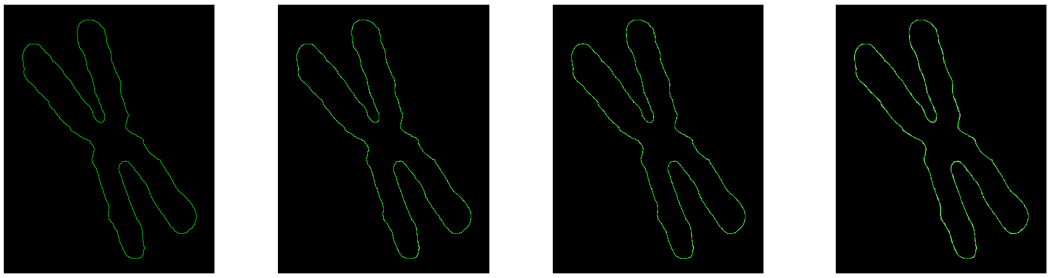
of the image and convert the points to complex numbers. After performing a fourier transform we calculate the center location and the range of frequencies we want to set to zero. We want to keep P descriptors which means we want to retain the P lowest frequencies and the DC component. All of the other descriptors are set to zero. The resulting function can be seen in Listing 5.

```
function [] = IPfourierdesc (f, P)
    % Get contour values
    [X,Y] = IPcontour(f);
    % convert to complex
    S_k = complex(X, Y);
    % Perform FFT
    d_f = fft(S_k);

    % Set all values to zero except for DC component (center) and the
    % lowest P frequencies
    center = round(length(S_k) / 2);
    range = round(center - P/2);
    for idx = 1:range
        d_f(center - idx + 1) = 0;
        d_f(center + idx + 1) = 0;
    end

    % Inverse fourier transform
    d_f = ifft(d_f);

    % Plot the result
    plot(round(real(d_f)), round(imag(d_f)),'b','LineWidth',1);
end
```

Listing 5: Function which implements fourier descriptor reconstruction

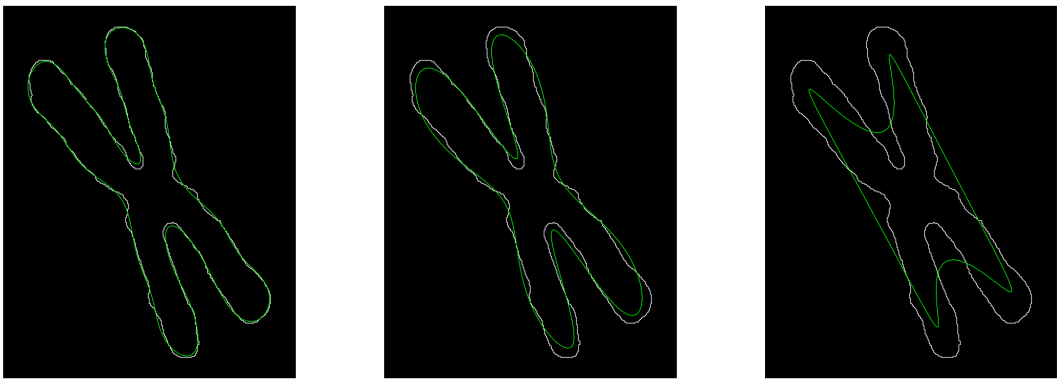
The assignment wanted us to recreate the images shown in the book in figure 11.20. The images created by us using Listing 5 can be seen in Figure 10 and Figure 11. The results are identical to results shown in the book thus our IPfourierdesc works correctly. When using all 1434 descriptors we get a perfect reconstruction of the original contour as can be seen in Figure 10a. Reducing the number of descriptors by ten-fold to 144 results still in a very good reconstruction of the original contour. From 36 descriptors and onward we lose some detail especially in the sharp corners. However even with just 18 descriptors we still get a general shape from the original chromosome. There is still some shape recognisable using 8 descriptors as can be seen in Figure 11c but a lot of details are lost.

A test script can be found in Listing 10.



(a) 1434 descriptors (b) 286 descriptors (c) 144 descriptors (d) 72 descriptors

Figure 10: Images obtained by applying Listing 5 on 'chromosome_boundary.tif' with a varying number of descriptors. Green is the reconstructed boundary and white the original boundary. Recreation of image 11.20 in the book.



(a) 36 descriptors (b) 18 descriptors (c) 8 descriptors

Figure 11: Images obtained by applying Listing 5 on 'chromosome_boundary.tif' with a varying number of descriptors. Green is the reconstructed boundary and white the original boundary. Recreation of image 11.20 in the book.

3.4 Apply your algorithm to the boundary

We also tested our function on 'lincoln.tif' to find out the minimum number of descriptors needed in order to still obtain a recognisable silhouette. The results can be seen in Figure 12 and Figure 13. Until 72 descriptors almost no details are lost and the silhouette is still very recognisable. From 36 descriptors onward some of the sharper edges are lost as can be seen in Figure 12c, but the general shape of the silhouette is still visible. With 20 descriptors pretty much all the details are lost and just a very general outline remains. The optimal number of descriptors seems to be 72 if you want all details and if you can work with a little bit less details 36 descriptors will do.

(a) 144 descriptors (b) 72 descriptors (c) 36 descriptors

Figure 12: Images obtained by applying Listing 5 on 'lincoln.tif' with a varying number of descriptors. Green is the reconstructed boundary.

(a) 28 descriptors (b) 20 descriptors (c) 18 descriptors

Figure 13: Images obtained by applying Listing 5 on 'lincoln.tif' with a varying number of descriptors. Green is the reconstructed boundary.

4 Test scripts

```
1 img = imread('fingerprint.tif');
2
3 % Create 5 images without smoothing
```

```
for idx = 10:10:50
    figure;
    g = IPgradientthresh(img, idx);
    imshow(g);
end

% Create 5 images with smoothing
img = IPfilter(img, 1/9 * ones(3,3));

for idx = 10:10:50
    figure;
    g = IPgradientthresh(img, idx);
    imshow(g);
end
```

Listing 6: Test script used for testing gradient threshold function outputting a set of images

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

%img = IPfilter(img, 1/9 * ones(3,3));

images = cell(40,1);

for idx = 10:50
    g = IPgradientthresh(img, idx);
    images{idx - 9} = uint8(g);
end

 % create the video writer with 1 fps
 writerObj = VideoWriter('gradient_no_filter.avi');
 writerObj.FrameRate = 10;
 % set the seconds per image
 secsPerImage = ones(1,41) * 1;
 % open the video writer
 open(writerObj);
 bw = colormap(gray(2));
 % write the frames to the video
 for u=1:length(images)
     % convert the image to a frame
     frame = im2frame(images{u}, bw);
     for v=1:secsPerImage(u)
         writeVideo(writerObj, frame);
     end
 end
 % close the writer object
 close(writerObj);
```

Listing 7: Test script used for testing gradient threshold function outputting a video.

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

```

```
sigma = 1;
p = 50;

IPMarrHildreth(img, sigma, p);
```

Listing 8: Test script for Marr Hildreth edge detector

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

img = IPcontour(img);
```

Listing 9: Test script for IPcontour

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

IPfourierdesc(img, 14);
```

Listing 10: Test script for IPfourierdesc

5 Code previous exercises

```
function [g] = IPfilter (f, filter)
    A = im2double(f); % convert image to double
    nr = size(A,1); % number of rows
    nc = size(A,2); % number of columns
    % Shift A cyclically in four directions
    A_up = [A(2:nr,:); A(1,:)]; % one row up
    A_down = [A(nr,:); A(1:nr-1,:)]; % one row down
    A_left = [A(:,2:nc) A(:,1)]; % one column to left
    A_right = [A(:,nc) A(:,1:nc-1)]; % one column to right

    % Shift A cyclically in other 4 diagonal directions
    A_down_right = circshift(A, [1 1]);
    A_up_right = circshift(A, [-1 1]);
    A_up_left = circshift(A, [-1 -1]);
    A_down_left = circshift(A, [1, -1]);

    B = filter(2,2) * A + filter(1,2) * A_up + filter(3,2) * A_down ...
    + filter(2,1) * A_left + filter(2,3) *  A_right ...
    + filter(3,3) * A_down_right + filter(1,3) * A_up_right ...
    + filter(1,1) * A_up_left + filter(3,1) * A_down_left;

    g = im2uint8(B); % convert to 8-bit
end
```

Listing 11: IP filter function.