Digital Image Processing
Lab 1

Group 50:
Henry Maathuis (S2589540)
Sebastian Wehkamp (S2589907)

December 3, 2017

Contents

1 Exercise 1 — Downsampling, upsampling, and zooming

1.1 a

For this exercise we created a matlab function that downsamples an image by a given factor. Downsampling is done by removing every Nth pixel where N depends on the factor provided by the user. The code for the function is found in Listing 1. The function takes an image and a factor as input after which it loops over the pixels in every row and column with steps of factor size. These pixels are stored in the output image which is returned.

```matlab
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 1: Code to downsample image using a given factor

1.2 b

The result of Listing 1 with a factor of 4 on the image *cktboard.tif* can be seen in Figure 1.

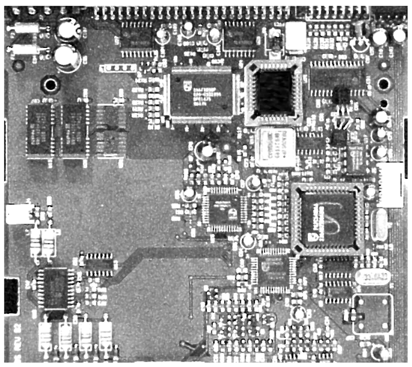


Figure 1: Downsampled image by factor 4

Since it is a bit hard to see if we actually succeeded in downsampling we can simply look at the sizes of the images. The original image is of size 594x671 while our image is of size 149x168 which is the expected size for our image.

1.3 c

The function listed in Listing 2 upsamples the provided image by placing 0 pixels between existing pixels. The number of zeroes entered between the pixels can be set by the upsampling factor. The first step is to create a matrix with zeroes having the size of the output image. The dimensions of the output matrix are $m * fac - (fac - 1)$ and $n * fac - (fac - 1)$, where fac is the upsample factor, m is the number of rows in the original image and n is the number of columns in the original image. The last step is to place the pixels from the input image into the output image taking into account the upsampling factor.

```matlab
function [output_img] = IPupsample(img, fac)

    img = im2double(img);

    [m, n] = size(img);

    % Initialize with 0s.
    output_img = zeros(m * fac - (fac - 1), n * fac - (fac - 1));

    % Put the original values back in taking into account the 0s.
    output_img(1:fac:end,1:fac:end) = img(:,:);
end
```

Listing 2: Code to upsample image using a given factor

1.4 d

Since showing the complete image in this report does not work due to the high resolution we decided to show the image at a high zoom level in Figure 2. You can see that in both rows and columns three out of four pixels is black as expected. This is also shown when we look at the image sizes, the original image was 594x671 and the new image is 2373x2681. This is the expected result as we chose not to add three empty extra pixels at the bottom and on the right.

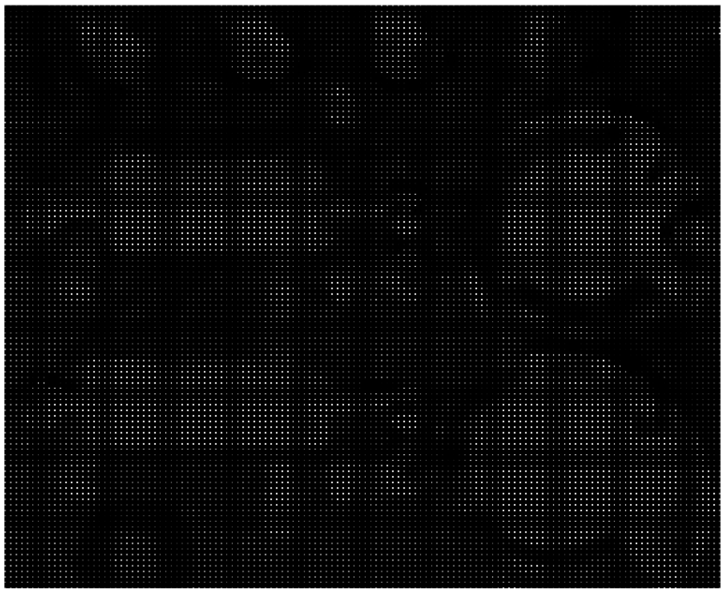


Figure 2: Zoom in on result of upsampling cktboard.tif with factor 4

1.5 e

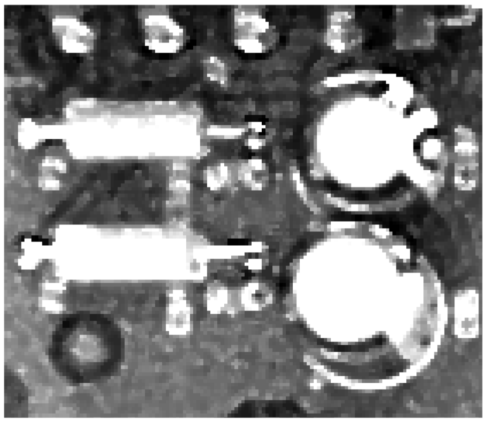
The function listed in Listing 3 zooms in on an image by using pixel replication, also known as nearest neighbour interpolation. To do this we used the matlab function *repelem* which does both column and row wise replication. The function caller can provide a factor by which the image should be zoomed. This factor is used in repelem to take care of the pixel replication.

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

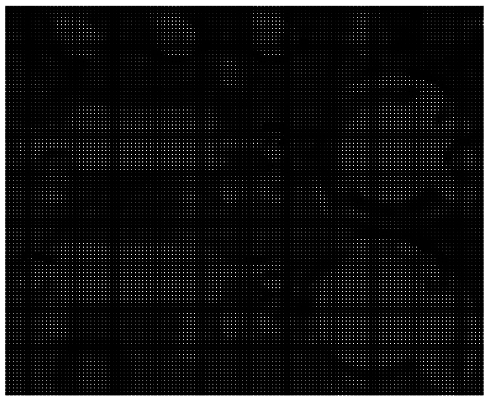
Listing 3: Code to zoom an image using pixel replication

1.6 f

A comparison between both images can be seen in Figure 3. You can see that in Figure 3a the image gets more pixelated. This is due to the implementation in which the nearest pixel simply gets replicated a given number of times creating a larger "super pixel". This in contrast to the upsampling example shown in Figure 3 in which all of the pixels in between are just made black.



(a) Zoom using pixel replication

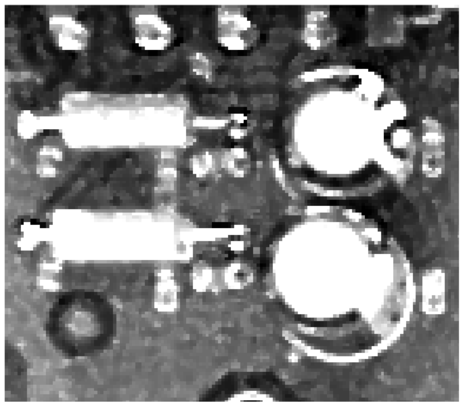


(b) Zoom using upsampling

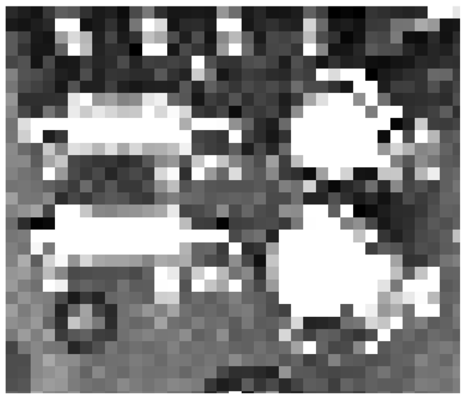
Figure 3: Image comparison between zooming in with pixel replication and upsampling

1.7 g

In Figure 4 you can clearly see that the original image in Figure 4a is much sharper than Figure 4b. This is due to the fact that in each dimension three out of four pixels are removed with downsampling after which they are filled up again with pixel replication resulting in 4x4 pixel areas with the same value.



(a) Original



(b) Downsampling and zooming applied

Figure 4: Image comparison between original image (Figure 4a) and the image after downsampling and zooming with factor 4 (Figure 4b)

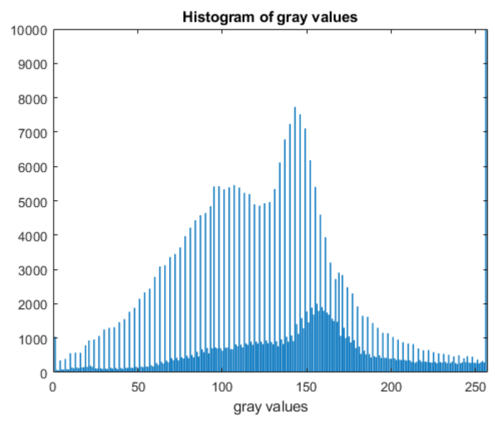
2 Exercise 2 — Histogram equalization

2.1 a

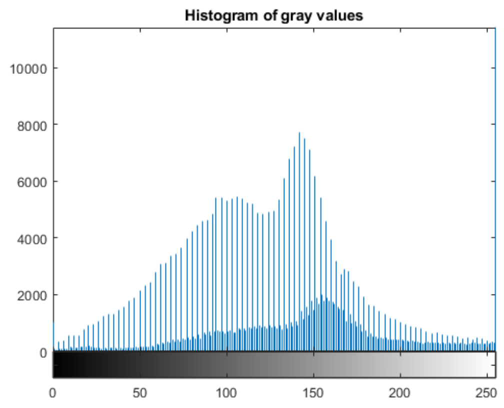
The function below creates a histogram from an image by iterating over each pixel and computing the frequency of each intensity value (in range 0 255) in an image. Then a vector is returned of size 1x256 where the indices that we use are the particular intensity values and the values stored are the frequencies with which each intensity value is stored. An histogram created using Listing 4 can be seen in Figure 5a. For verification purposes we also created a histogram using the Matlab function *imhist* which can be seen in Figure 5b.

```
function [h] = IPhistogram(img)
    img = im2double(img);
    img = img * 255;
    %Allocate space for histogram
    h=zeros(1, 256);
    index=1:1:256;
    %Count number of pixels for every gray value
    for n=1:256
        h(n) = length(find(img==n-1));
    end
    %Plot the histogram
    bar(index,h)
    ylim([0 100000]);
    title('Histogram of gray values')
    xlabel('gray values')
end
```

Listing 4: Function which creates a histogram of an image



(a) Our Histogram of cktboard.tif



(b) Histogram of original image

Figure 5: Comparison between our histogram function and matlabs histogram function imhist

2.2 b

In this exercise we had to implement the histogram equalisation technique discussed in Section 3.3.1. We did this by creating a function which accepts an image and creates a histogram of this image. Then for every pixel in this image a helper function is called which computes $\frac{1}{N} * \sum_{x=1}^{P_i}(h(x))$ where P_i is equal to the grey value of the current pixel, N the total number of pixels, and h is the histogram. This value is stored in the new image. The main function can be seen in Listing 5 and the helper function can be found in Listing 6.

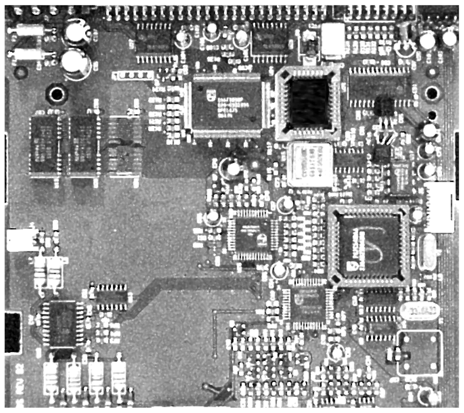
```
function [new_img] = IPhisteq(img)

    [m,n] = size(img);
    d_img = im2double(img);
    %Create histogram
    h = IPhistogram(d_img);
    %Allocate space for new image
    new_img = zeros(m,n);
    for i=1:m
        for j=1:n
            %Apply formula 3.3-8 from the Image Processing book
            new_img(i,j) = IPhisteqhelper(d_img,i,j,h);
        end
    end
    new_img = uint8(new_img);
end
```

Listing 5: Code that uses a histogram equalization technique

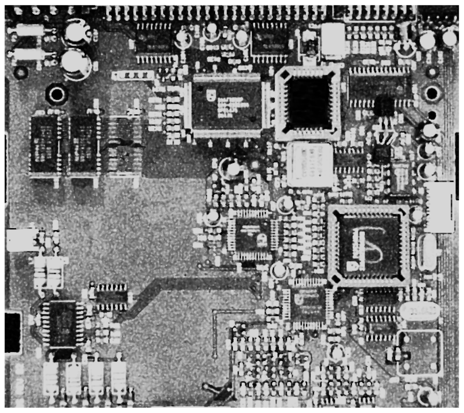
```
function [s_k] = IPhisteqhelper(img,i,j,h)
    [m, n] = size(img);

    % Number of pixels in original image
    N = m * n;

    % Compute original intensity of image
    gl = img(i,j) * 255;

    % (L - 1) / number of pixels * sum of number of pixels
    % up till and including that intensity in the original image
    s_k = 255 / N * sum(h(1:gl+1));

    % Round to nearest integer
    s_k = round(s_k);
end
```

Listing 6: Helper function called in Listing 5

The result from Listing 5 can be found in Figure 6b. The most obvious difference between Figure 6a and Figure 6b is that the image got a bit darker. This can also be seen in Figure 7. In the original image there were a small amount of darker pixels and in the new image there are quite a lot of them. You can clearly see that the histogram of Figure 7b is much flatter than Figure 7a.

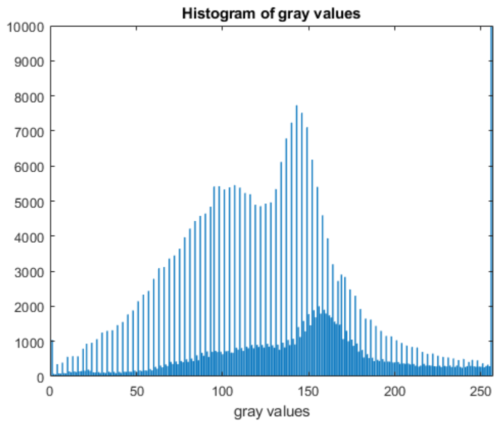


(a) Original image

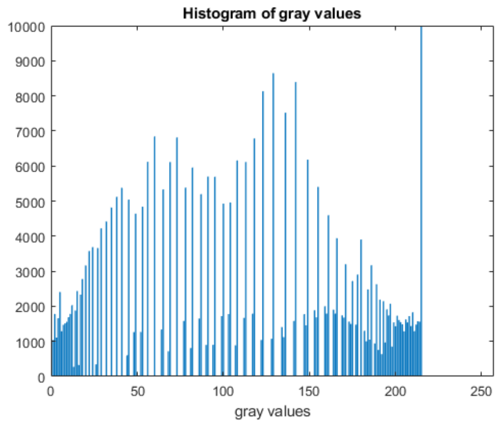


(b) Equalized image

Figure 6: Comparison between the original image and the image normalised created using Listing 5



(a) Histogram of original image

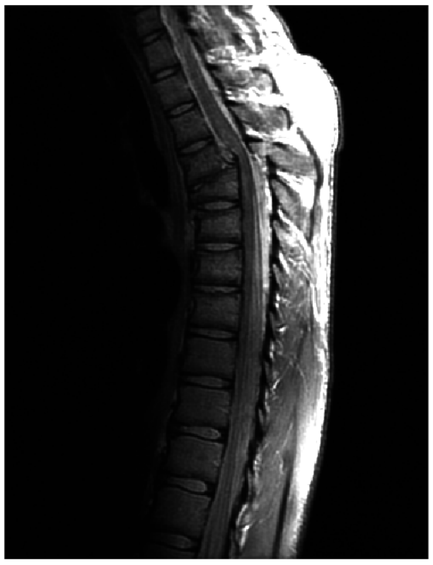


(b) Histogram of equalized image

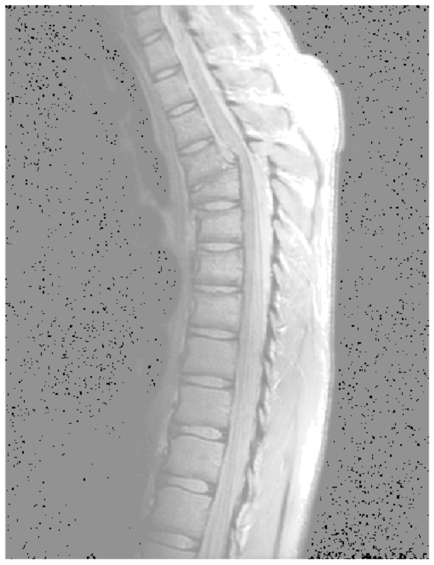
Figure 7: Comparison between the histograms of the original image and the image normalised created using Listing 5

2.3 c

As can be seen in Figure 9 the original image contains a lot of very dark, almost black, pixels. There are not many pixels distributed over the middle and then there is a peak at clear white pixels. The normalised result improves the clarity of the image hugely as can be seen in Figure 8. Since there is a huge peak of black pixels the equalisation technique tries to compensate this by brightening all pixels in the middle while still maintaining the most important details. The equalised histogram located in Figure 9b shows this as well. Note that to create this histograms we used a different y-scale of 100000 in contrast to the 10000 shown in Listing 4 in order to show the huge peak of black pixels.

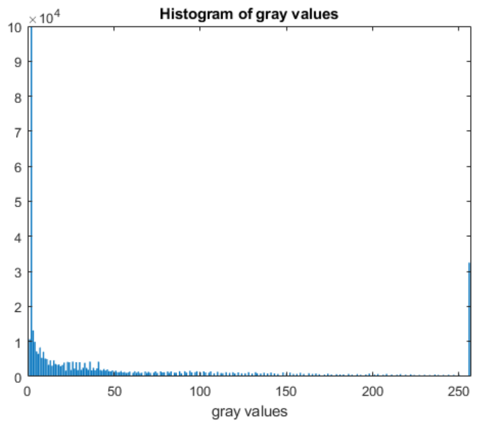


(a) Original image

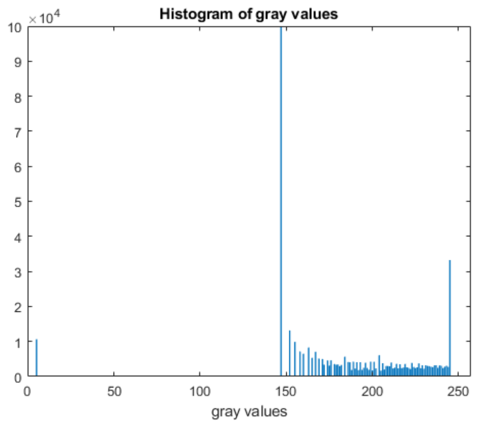


(b) Equalized image

Figure 8: Comparison between the original image and the image normalised created using Listing 5



(a) Histogram of original image



(b) Histogram of equalised image

Figure 9: Comparison between the histograms of the original image and the image normalised created using Listing 5

3 Exercise 3 — Spatial filtering

3.1 a

For this exercise we created a function named IPfilter, Listing 7 which applies a given 3x3 mask to the image. First the image is shifted cyclically to get *A_up*, *A_down*, *A_left*, and *A_right*. To get the diagonal values we used *circshift* to shift the values cyclically.

```
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 7: Function performing spatial filtering on image

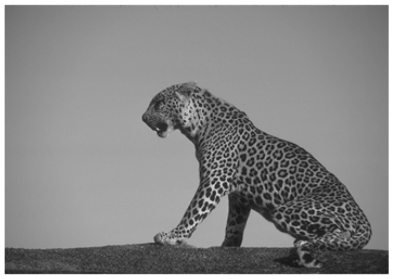
3.2 b

The laplacian filter code comes down to simply passing a 3x3 mask to the IPfilter function. The mask should have -1 in all places except for the centre which has a value of 8. The code can be seen in Listing 8.

```
function [g] = IPlaplacian (f)
    filter = [-1 -1 -1; -1 8 -1; -1 -1 -1];
    %filter = [1/9 1/9 1/9; 1/9 1/9 1/9; 1/9 1/9 1/9];

    %Apply the filter
    g = IPfilter(f, filter);
end
```

Listing 8: Function implementing Laplacian enhancement technique

The results of laplacian filtering can be seen in Figure 10. A LaPlacian filter should highlight areas of rapid change. Our LaPlacian filtering clearly does this and would thus be very suitable for edge detection. The tiger is clearly visible and separated from the background, even the individual spots on the tigers' body are nicely separated.


(a) Original tiger image

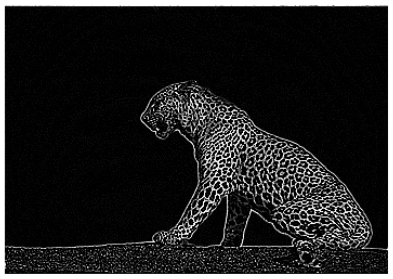

(b) Tiger image with LaPlacian filtering applied

Figure 10: Comparison between the original image and the image with LaPlacian filtering applied

4 Individual contributions

The work was mainly split in a report and a programming part. Although we both contributed to both parts, Henry spent most of his time on the programming and Sebastian more time on the report.