# Image Processing
# Lab 1

Kevin Gevers (s25595987)
Jeroen Overschie (s2995697)

January 20, 2021

Note, that all used source code can be found attached next to this report's pdf file. Its structure should be self-explanatory; all requested functions are named accordingly and any extra functions are explained in the report. Note that (almost) every function has a corresponding test script, which is named just like its function, but with a suffix '_test'. Where possible, we followed the terminology from the book [1] for variable naming.

## Exercise 1

In this exercise, **downsampling**, **upsampling** and **zooming** functions are requested. Although we at first implemented them separately, we realized all three are spatial operations that share the same way of geometrically transforming the original image pixel coordinates, just with different scaling factors and/or interpolation method. For example, we can use a scaling constant, say $factor$, to both shrink- (downsampling) using $factor < 1$ and grow an image (upsample) using $factor > 1$. Given these similarities, what then just differs is the *interpolation* method, which can be passed as an argument.

For this reason, we built a generic transformation function, IPSCALING_TRANSFORMATION. This function takes in an image, an (affine) transformation matrix (must be of a certain form), and a parameter controlling which interpolation method to use. Because our requested scaling implementations use this function, we explain this function first.

### IPSCALING_TRANSFORMATION

As described in the book [1] section 2.6 *'Geometric Transformations'*, it is possible to compute new coordinates of some various geometric transformations using just a matrix. Multiplying a coordinate vector by this matrix then produces its new mapped location. Like said in the text, what can be particularly useful of this method is that one can 'stack' various transformations in one matrix, by multiplying various transformation matrices. Even though we do not need this feature in our implementation, we still thought of this mapping step as an elegant way to approach the problem. The scaling transformation matrix $A$ is defined as:

$$A = \begin{bmatrix} cx & 0 & 0 \\ 0 & cy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

With $cx$ and $cy$ the horizontal and vertical scaling factors, respectively. We can now compute the transformed coordinates by multiplying the coordinate vector with the transformation matrix (Eq. 2-45):

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = A \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Even though we can now map from the original image space to the transformed space, it is not trivial how now to determine intensity levels for the unknown pixel values in the transformed space; i.e. interpolation. To make this interpolation process easier later on, we chose to use an *inverse mapping* approach. With inverse mapping, we do not map coordinates from the original space, but rather, we take the transformed coordinates directly and compute the original image position to compute its new intensity value. Given a coordinate in the transformed space, we can obtain its original coordinates using the inverse transformation matrix $A^{-1}$: hence 'inverse' mapping.

❖ For the implementation of IPSCALING_TRANSFORMATION, see Listing 1. We determine the scaled image dimensions by forward mapping the bottom-right coordinate, which is equal to the original image dimensions vector. This indeed returns the scaled image dimensions, but assumes the image only scales, but does not rotate or sheer. This is an assumption we can make, however, because we only perform image scaling indeed.

Now that we have the scaled image dimensions, each output pixel's corresponding original coordinates are computed. Note that we offset the original coordinates by some amount $1/2(1-A^{-1})$ in order to let the original pixels point to the **centers** of their respective output location. Given these coordinates, we can estimate its new intensity value: using **interpolation**. We created the IPINTERPOLATE function for this.

## IPINTERPOLATE

The interpolation function has the job of estimating unknown pixel coordinate intensity levels. Having inverse mapped the transformed coordinates, we obtain their position in the original space. Because a scaling operation took place, these probably have decimals: e.g. when growing a 2x2 image to 4x4 size, a pixel in transformed space at $(3,3)$ would be mapped to $(3/2, 3/2)$ like so:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} A^{-1} = \begin{bmatrix} 3 \\ 3 \\ 1 \end{bmatrix} A^{-1} = \begin{bmatrix} 3 \\ 3 \\ 1 \end{bmatrix} \begin{bmatrix} 1/2 & 0 & 0 \\ 0 & 1/2 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 3/2 \\ 3/2 \\ 1 \end{bmatrix}$$

Thus, the challenge lies in estimating values for pixels with unknown intensity values, like $(3/2, 3/2)$. We can do this by several means: *Nearest Neighbor interpolation*, *Bilinear interpolation* or *Bicubic interpolation*, among others. Applying no interpolation at all would yield zero-valued pixels in the output image. Even though the assignment only asked for Nearest Neighbor interpolation and zero-padding, we also implemented Bilinear interpolation out of curiosity.

❖ See Listing 2 for interpolation implementation. Most notably, **nearest neighbors** works by finding the first available pixel value below the transformed coordinate by using ROUND. Remember that the inverse mapped original coordinates probably have decimals; the rounding operation rounds these coordinates to the nearest pixel in the original space. Note that we are handling images and we can assume we are operating in a *rectilinear grid* with all spacing equal. Using the **no interpolation** setting just returns a zero-intensity value unless the output pixel can be **exactly mapped** to an original pixel.

## (a)

Now onto the requested functions, which are now easy to implement given the newly available functions. The down-sampling function takes in an image and a scaling factor, $factor$. To down-sample, all we have to do is pass a value $factor < 1$ to IPSCALING_TRANSFORMATION, i.e. to shrink with a factor of 4 we pass $factor = 1/4$.

❖ See implementation: Listing 3. Nearest neighbor interpolation is used and the image is cast to an integer image before saving the result.

## (b)

See Figure 1. It can be observed that the x4 down-sampled image still looks pretty nice, despite the simple interpolation method used. When down-sampling x15, however, it can be clearly seen that nearest neighbors is not an optimal interpolation method.

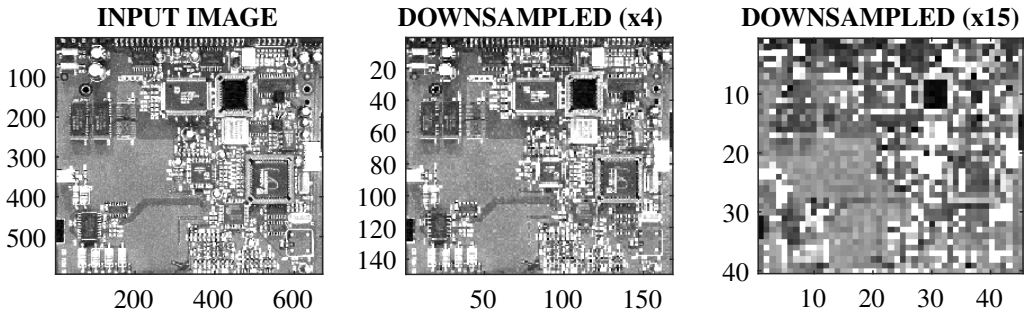



Figure 1: Comparison plot of original- and downsampled cktboard image. Middle image shows a factor x4 downsampling and the rightmost image a factor of x15 downsampling.

## (c)

Up-sampling is done easily too, using a value of $factor > 1$. Note that it is requested to interpolate by 'introducing zeros'. In our implementation, this corresponds to using IPINTERPOLATE using $interpolation =' none'$, meaning an output pixel is only assigned an intensity value if its original pixel value could be linearly mapped. This results in unknown pixels having no intensity value, i.e. black pixels. ❖ Implementation: Listing 4.

## (d)

See Figure 2. It can be observed that the image is very dark since a lot of pixels did not get assigned an (interpolated) intensity value.

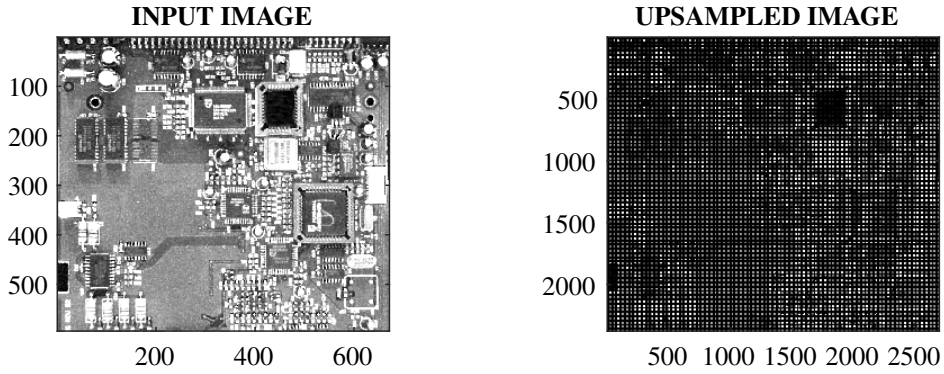



Figure 2: Comparison plot of original- and upsampled cktboard image.

## (e)

Finally, we implement a zooming function. This is upsampling but with interpolation: 'pixel replication' or better, nearest-neighbor interpolation. Again we use our generic function, using $factor > 1$ and 'nearest' interpolation.

❖ Implementation: Listing 5.

## (f)

See Figure 3 for a x4 factor zooming of the cktboard image. The zoomed image is 4 times the size, but looks the same, due to pixel replication. In comparison to (d), this image looks properly like the original. The image in (d) was distorted by the introduced zeros.

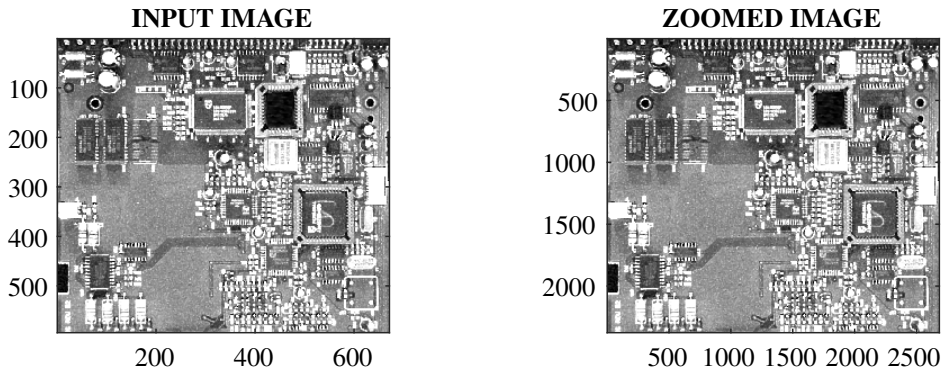



Figure 3: Comparison plot of original- and zoomed cktboard image.

## (g)

In this last exercise, we downsample and zoom back in. See Figure 4. Even though the images are back to the same size, we definitely lost some detail. See the bottom image. It can be observed that in the right-bottom 'reconstructed' image, pixel values are indeed *replicated*: looking at the axes one can see that multiple pixel values of the same intensities span multiple coordinates.

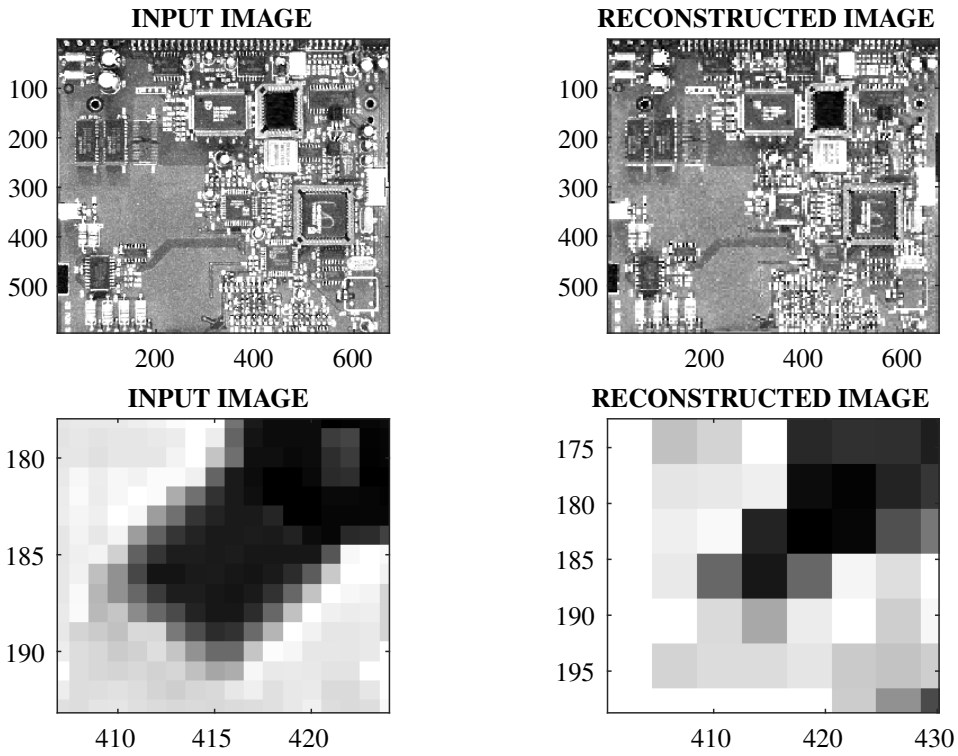



Figure 4: Comparison plot of original- and recontructed cktboard image. The bottom image presents a zoomed-in view of the original and reconstructed image.

## Bilinear interpolation extra

Out of curiosity, we compared our nearest neighbor interpolation with **Bilinear interpolation**. It is a bit more complicated but still relatively easy. It uses the 4 nearest-neighbors to compute the weighting each pixel has on the estimated

intensity value. It can be expressed as $v(x,y) = ax + by + cxy + d$. Since we are estimating intermediate pixel values, we get a much smoother output image. See Figure 5. It can be observed that our method performs very similarly using both interpolation methods to Matlab's built-in function.

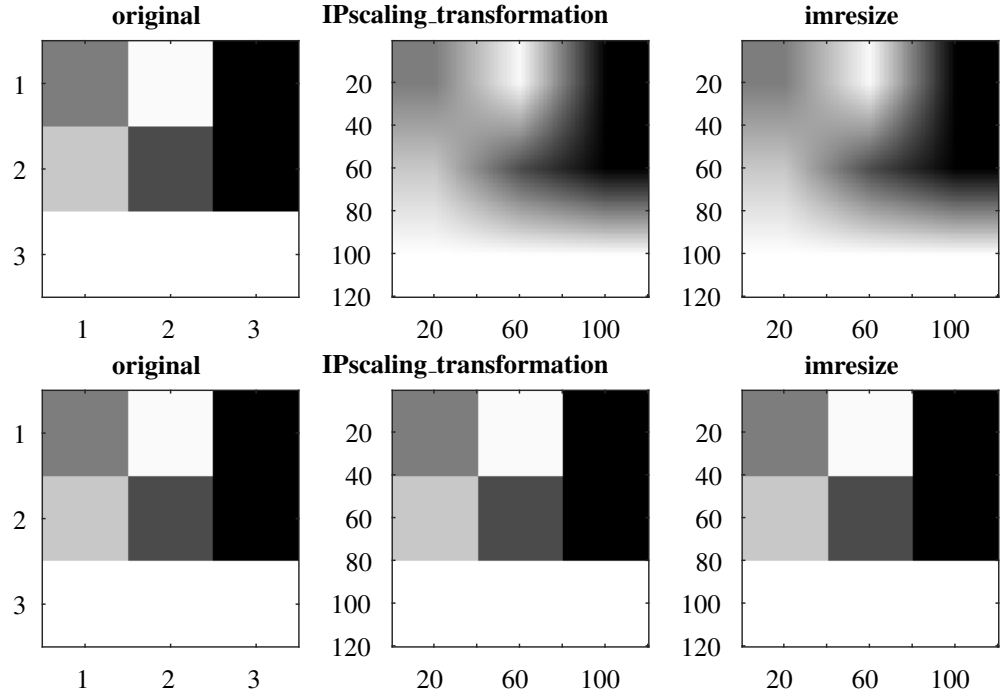



Figure 5: Interpolation using IPINTERPOLATE. The top row shows a zooming operation of x40 using *Bilinear interpolation*; the bottom row also does a x40 magnification, but using *nearest neighbors interpolation*.

# Exercise 2

## (a)

Histogram computation was discussed in the book section 3.3, and was defined as follows. Let $r_k$ denote the image intensities of some L-level digital image, e.g. where $k = 0,1,2,...,L-1$. We define the intensity levels of our digital image with the function $f(x,y)$, that outputs an intensity level given a set of coordinates. Now, we the unnormalized histogram of $f$ is defined as:

$$h(r_k) = n_k \text{ for } k = 0,1,2,...,L-1$$

Where $n_k$ denotes the number of pixels present in the image that are at intensity level $k$. So, what is requested in this exercise is an implementation of the function $h$. Note that one can subdivide the intensity scale in an amount of *histogram bins*, though we left the number of bins equal to the number of intensity levels, that is $L$. Specifically, our image is 8-bit, so $L = 2^8 = 256$. Let us compare our implementation of the histogram with the built-in Matlab one, see Figure 6.

We observe that the blurry image has the majority of its pixels in the lower value ranges of $[0,255]$, i.e. dark or black pixels. This corresponds to what we see in the image.

❖ Implementation: Listing 6.

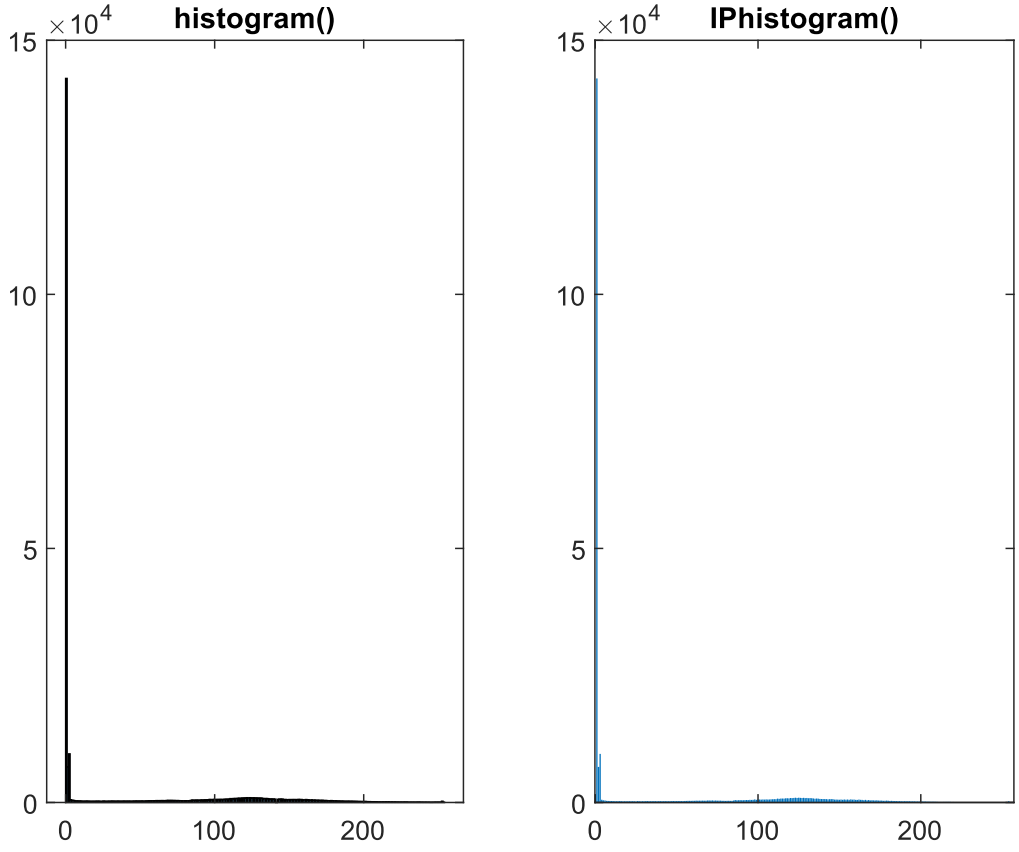



Figure 6: Example of the result of applying our histogram function, IPHISTOGRAM versus Matlab's built-in one.

## (b)

Next, we implement histogram equalization. In histogram equalization, we can increase an image's contrast by distributing intensity values better across the histogram value range. We do this by mapping each intensity level to a new value using the *Cumulative Distribution Function* (CDF) of the intensity value and multiplying it by the number of intensity levels, $L-1$. This can be expressed as (Eq 3-11):

$$s = T(r) = (L-1)\int_0^1 p_r(w)dw$$

Where $p$ is the *normalized* histogram, which is defined as $p(r_k) = \frac{h(r_k)}{MN} = \frac{n_k}{MN}$, for which the sum for all values of $k$ should always be 1. We can think of the normalized histogram as expressing the 'probability' of encountering that pixel intensity value - and thus the total probability of encountering any pixel intensity value should be one: $\sum_{k=0}^{L-1} p(r_k) = 1$. Now, to compute the integral shown above in a discrete way, we can use the following formula (Eq 3-15):

$$s_k = T(r_k) = (L-1)\sum_{j=0}^{k} p_r(r_j) \text{ where } k = 0,1,2,...,L-1$$

Where $s_k$ will be the histogram-equalized pixel intensity value at $k$, given an intensity value $r$. Now onto the implementation. See ❖ Listing 7. The function uses the previously built IPHISTOGRAM to compute the image histogram

and normalizes it. Next, we re-compute each intensity value using histogram equalization and saves it in a vector $T$. Now we only have to map all image pixel intensities to their new values in $T$ by looping the image.

## (c)

The result of this operation can be seen in Figure 7. The histogram equalization effect on the blurry moon image did not have much effect, however. It appears that the image histogram already had a fair distribution, and was not significantly enhanced by our equalization technique.

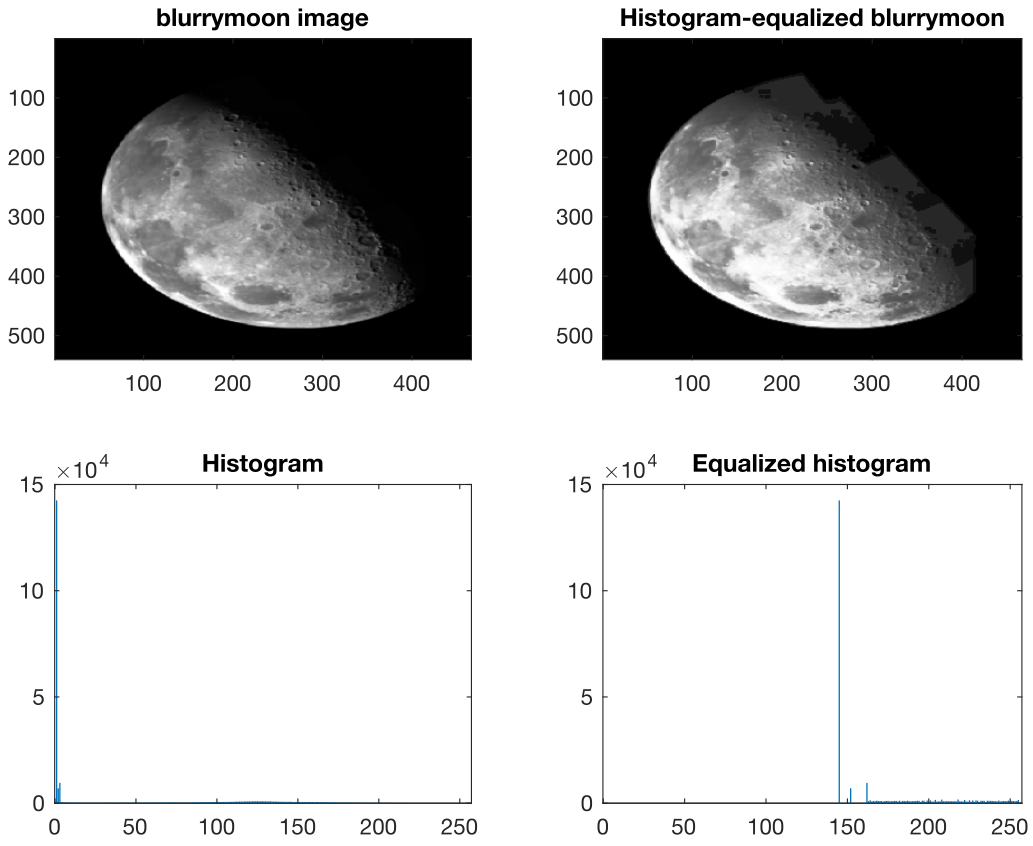



Figure 7: Example of the result of applying our histogram function, IPHISTOGRAM versus Matlab's built-in one.

A case where the effects of histogram equalization can be seen more clearly is in Figure 8, which shows histogram equalization applied on a different image, taken from an online article [2]. It can be observed that this time around, the histogram equalization technique has a clearly positive effect on enhancing the image contrast.

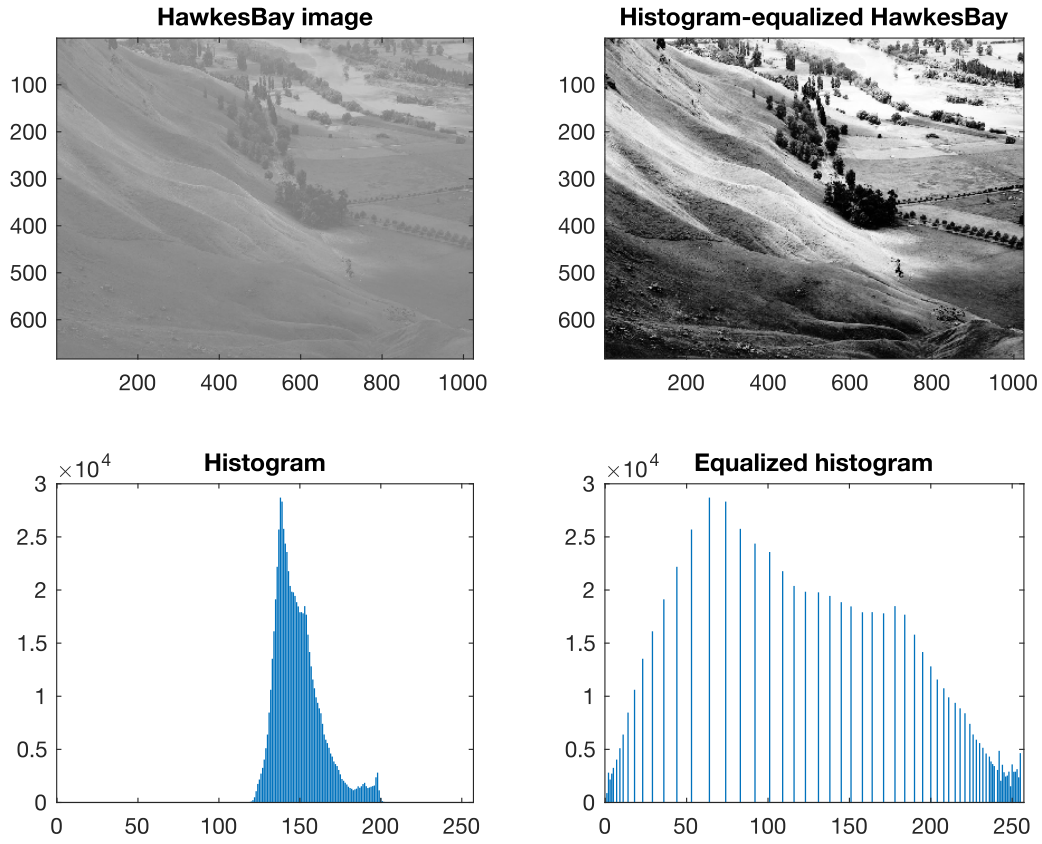



Figure 8: Example of the result of applying our histogram function, IPHISTOGRAM versus Matlab's built-in one.

# Exercise 3

## (a)

For this exercise, we are tasked to implement a function that performs spatial filtering of an image using a 3$x$3 mask (or kernel). We are not allowed to use the Matlab functions `filter`, `filter2`, `conv` or `conv2`, seeing as those already do this or part of it.

While there are different types of filtering techniques for images, but for this exercise, we only focus on spatial filtering. Spatial filtering of an image is done by moving the center of a kernel over the image and computing the sum of products at each pixel. This process follows Formula 1 (Eq $3-30$ in the book) for each individual pixel when using a 3$x$3 mask.

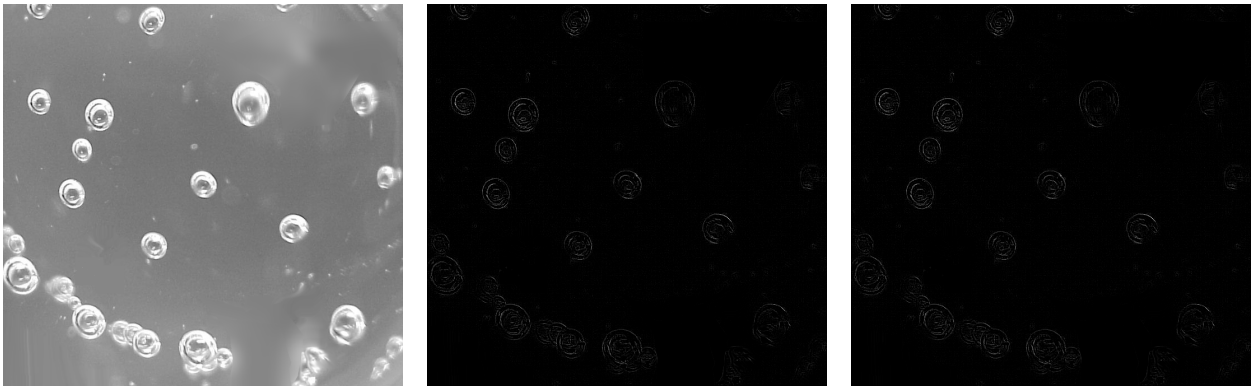
$$\begin{aligned} g(x,y) = w(-1,-1)\cdot f(x-1,y-1) + w(-1,0)\cdot f(x-1,y) + ... \\ + w(0,0)\cdot f(x,y) + ... + w(1,1)\cdot f(x+1,y+1) \end{aligned} \tag{1}$$

While the exercise requires us to implement it for a 3$x$3 kernel, we decided to use the more general formula that is able to handle an arbitrary sized kernel. We did this in view of future exercises and to make it more robust. This formula uses the dimensions of the kernel by looping from $-a$ to $a$ and looping from $-b$ to $b$. For a kernel of size $m \times n$ the value of $a$ and $b$ is determined by $m = 2a+1$ and $n = 2b+1$. We will use Formula 2 (Eq $3-31$ in the book) for our function.

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x+s, y+t) \tag{2}$$

Important to note is that the kernel its center is placed on the pixel we are modifying. So the center of the kernel is indexed with $(0,0)$, due to how the indexing in Matlab works we had to compensate for this using the variables `kernelCenterX` and `kernelCenterY`. The implementation can be found in ❖ Listing 8 and is very self-explanatory.

Figure 9: Results of testing IPFILTER



(a) Input image *"bubbles"*

(b) The output image using out own function IPFILTER.

(c) The output image using matlab's function `filter2`.

To see if our implementation is working correctly we used the provided image *"bubbles.tif"* and compared the result of using our function IPFILTER to the results of using the matlab function `filter2`, as can be seen in Figure 9. Both Figure 9b and 9c are exactly the same, thus showing our implementation is correct.

## (b)

For this part of the exercise, we needed to implement the Laplacian (image) enhancement technique. This technique involves creating a Laplacian filter and then using the filter to enhance the original image. The Laplacian filter is the second derivative of the image, using a kernel. Formula 3 (Eq $3-53$ in the book) shows how this can be calculated for each pixel. It is easy to deduce here that the Laplacian filter is simply a regular filter with a specific kernel applied. The book provides 4 different kernels that can be used, which can be seen in Figure 10.

$$\nabla^2 f(x,y) = f(x+1,y) + f(x-1,y) + f(x,y+1) + f(x,y-1) - 4f(x,y) \tag{3}$$

| | | |
|---|---|---|
| 0 | 1 | 0 |
| 1 | −4 | 1 |
| 0 | 1 | 0 |

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 1 | −8 | 1 |
| 1 | 1 | 1 |

| | | |
|---|---|---|
| 0 | −1 | 0 |
| −1 | 4 | −1 |
| 0 | −1 | 0 |

| | | |
|---|---|---|
| −1 | −1 | −1 |
| −1 | 8 | −1 |
| −1 | −1 | −1 |

Figure 10: Kernels that can be used for Laplacian filters [1]

Once the Laplacian filter is made the filter needs to be subtracted or added to the original image, depending on which kernel was used. This is done by adding a variable $c$ to the equation, where $c$ is set to $-1$ for the left 2 kernels from Figure 10 and $c$ is set to 1 for the right 2 kernels from Figure 10. Formula 4 (Eq $3-54$ in the book) shows the now complete formula for the Laplacian enhancement technique.

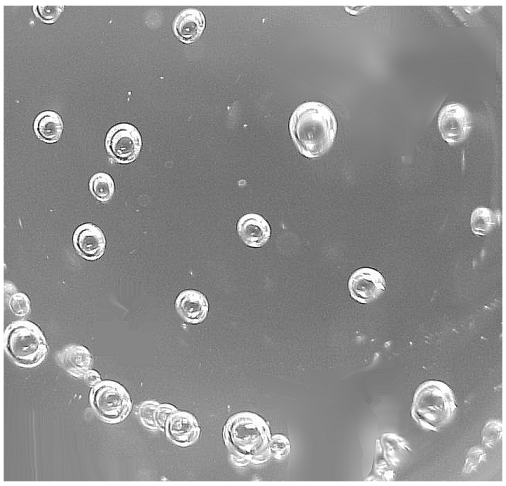
$$g(x,y) = f(x,y) + c\left[\ \nabla^2 f(x,y)\ \right] \tag{4}$$

The implementation of this function can be found in ❖ Listing 9.

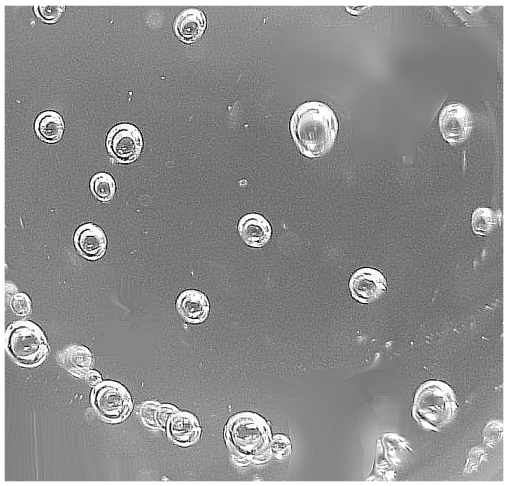
## (c)

For this last part we had to apply our function IPLAPLACIAN to "*bubbles.tif*" and discuss the results. We used the 2 left-most kernels from Figure 10 and in Figure 11 the Laplacian filter and the enhanced image can be seen for both.

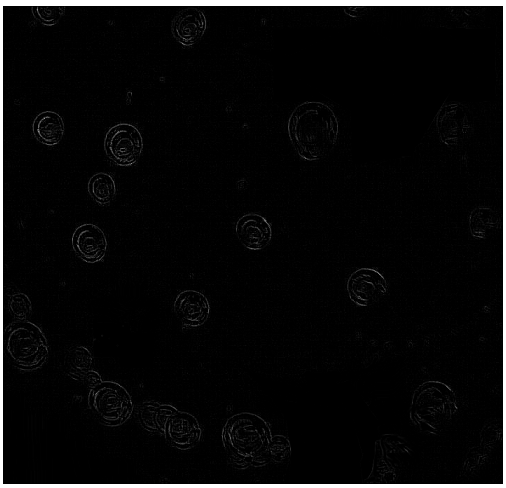
Figure 11: Results of testing IPLAPLACIAN



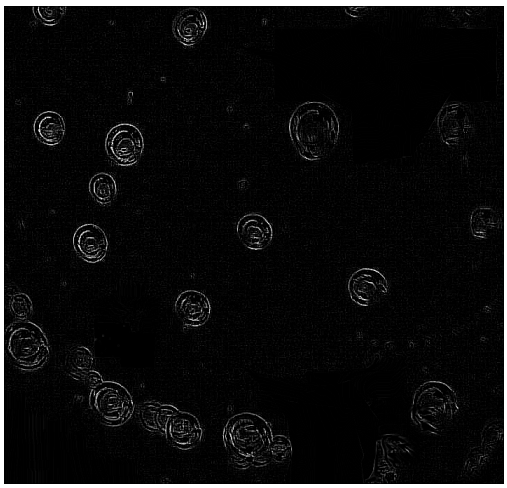
(a) The enhanced image using the left-most kernel from Figure 10.



(b) The enhanced image using second from the left kernel from Figure 10.



(c) The Laplacian filter used in creating the enhanced image using the left-most kernel from Figure 10.



(d) The Laplacian filter used in creating the enhanced image using the second from the left kernel from Figure 10.

In the filter images (Figure 11c and 11d) it clearly shows the edges from the bubbles. Since we use the left 2 kernels from Figure 10 they are subtracted from the original image and thus make the resulting images sharper. In Figure 11a and 11b it is clear that the edges are more defined compared to the original image (Figure 9a). The difference between the resulting images of the two kernels is also quite clear. Especially in the Laplacian filter images, the edges are much clearer as can be seen in Figure 11d. This does bring along more background noise from the initial image, which does not look as nice in the enhanced image. This shows us that it is important to select the right kernel for the image you are trying to enhance.

# References


[1] Rafael C. Gonzalez and Richard E. Woods. *Digital image processing*. Prentice Hall, Upper Saddle River, N.J., 2008.

[2] Wikipedia. Histogram equalization — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Histogram%20equalization&oldid=982455984`, 2020. [Online; accessed 30-November-2020].


# A Code

## A.1 Exercise 1

Listing 1: `IPscaling_transformation.m: perform geometric transformations of the scaling kind, given an affine transformation matrix or a scaling constant.`

```
function It = IPscaling_transformation(I, A, interpolation)
% IPscaling_transformation Computes an image scaling operation
% using an affine transformation matrix.
% Arguments:
%       I: Input image
%       A: Affine transformation matrix [3x3] of form
%           [cx 0 0; 0 cy 0; 0 0 1;] or a scalar value, then cx=cy.
%       interpolation: interpolation method. See IPinterpolate.m
if isnumeric(A)
    A = [A 0 0; 0 A 0; 0 0 1;];
end
if ~exist('interpolation', 'var')
   interpolation = 'none';
end
I = im2double(I);

% Image size
[M, N] = size(I); % height, width
D = [M, N, 1];  % dimensions

% Transformed dimensions
Dt = D * A;
Mt = round(Dt(1));
Nt = round(Dt(2));

% Map coordinates to new values
It = zeros(Mt, Nt);

%% Inverse mapping
% Perform _inverse mapping_ instead of forward mapping. Compute
```

```
% the inverse affine transformation matrix $A^{-1}$. See
% section 2.6 of (DIP, 42 - Gonzalez, Woods) book, page 102.
for y = 1:Mt
    for x = 1:Nt
        Pt = [x, y, 1];
        P = Pt / A; % original coordinate (same as Pt * inv(A)).
        offset = diag(0.5 * (1 - inv(A)))'; % inverse mapping centering offset
        It(y, x) = IPinterpolate(I, P, offset, interpolation);
    end
end

end
```

Listing 2: IPinterpolate.m: interpolate an unknown pixel intensity value using one of the supported methods.

```
function v = IPinterpolate(I, P, offset, interpolation)
% IPinterpolate Interpolate image pixel for image using given method.
% Arguments:
%       I: Input image
%       P: Coordinates of pixel with unknown intensity value; in original
%           (i.e. non-transformed) coordinate space. e.g. coordinates might
%           have decimals.
%       interpolation: one of
%           ('none' | 'nearest' | 'bilinear')
%           meaning no interpolation, nearest neighbor interpolation and
%           bilinear interpolation, respectively.
[M, N] = size(I);
switch interpolation
    %% Nearest neighbor interpolation
    % Finds the nearest pixel in the inverse mapping using `ceil`.
    case 'nearest'
        % Make sure within bounds
        Po = P + offset;
        x = min(max(1, Po(1)), N);
        y = min(max(1, Po(2)), M);
        v = I(round(y), round(x));
    case 'bilinear'
    %% Bilinear interpolation
    % See Bilinear interpolation Wikipedia page for used terminology and
    % also https://bit.ly/3o3vcxD for parts of implementation.
        Po = P + offset;
        x = Po(1);
        y = Po(2);
        % Any values out of acceptable range
        x(x < 1) = 1;
        x(x > N - 0.001) = N - 0.001;
        x1 = floor(x);
        x2 = x1 + 1;
        y(y < 1) = 1;
        y(y > M - 0.001) = M - 0.001;
        y1 = floor(y);
        y2 = y1 + 1;
        % Neighboring Pixels
        Q = [I(y1,x1); I(y1,x2); I(y2,x1); I(y2,x2);];
```

```
        % Pixels Weights
        b = [(y2-y)*(x2-x); (y2-y)*(x-x1); (x2-x)*(y-y1); (y-y1)*(x-x1);];
        v = dot(b, Q);
    otherwise % no interpolation, i.e. 'none'
        % if this pixel maps to an original pixel
        if mod(P(1), 1) == 0 && mod(P(2), 1) == 0
            v = I(P(2), P(1));
        else % otherwise pad with zeros; no interpolation
            v = 0;
        end
end
end
```

### A.1.1 Exercise 1 (a)

Listing 3: IPdownsample.m: downsample images using IPscaling_transformation.

```
function downsampledImage = IPdownsample(I, factor)
% IPdownsample Down-samples an image by a factor of `factor`, which
% must be a (positive) integer, i.e. factor >= 1.
% Arguments:
%   I: input image to downsample
%   factor: factor to shrink with. Must be an integer.
assert(isinteger(factor), 'factor must be an integer (was %d)', factor);
It = IPscaling_transformation(I, 1/double(factor), 'nearest');
downsampledImage = uint8(It * 2^8); % normalize back to 8-bit int image.
end
```

### A.1.2 Exercise 1 (c)

Listing 4: IPupsample.m: upsample images using IPscaling_transformation.

```
function upsampledImage = IPupsample(I, factor)
% IPupsample Up-samples an image by a factor of `factor`, which
% must be a (positive) integer, i.e. upsamplingFactor >= 1. Function up-
    samples
% image by introducing zeros.
% Arguments:
%   I: input image to up-sample
%   factor: factor to grow with. Must be an integer.
assert(isinteger(factor), 'factor must be an integer (was %d)', factor);
It = IPscaling_transformation(I, double(factor), 'none');
upsampledImage = uint8(It * 2^8); % normalize back to 8-bit int image.
end
```

### A.1.3 Exercise 1 (e)

Listing 5: IPzoom.m: zoom images using IPscaling_transformation.

```
function zoomedImage = IPzoom(I, factor)
% IPzoom Zooms an image by a factor of `factor`, which
% must be a (positive) integer, i.e. upsamplingFactor >= 1. Function zooms
% image by replicating pixels.
% Arguments:
%   I: input image to zoom
%   factor: factor to zoom with. Must be an integer.
assert(isinteger(factor), 'factor must be an integer (was %d)', factor);
```

```
It = IPscaling_transformation(I, double(factor), 'nearest');
zoomedImage = uint8(It * 2^8); % normalize back to 8-bit int image.
end
```

## A.2 Exercise 2

### A.2.1 Exercise 2 (a)

Listing 6: IPhistogram.m: construct an image's histogram.

```
function h = IPhistogram(I)
% Image size
M = size(I, 1); % height
N = size(I, 2); % width

% Histogram bins
h = zeros(1, 2^8); % 8-bit image

% Count pixels of some intensity value, `r_k`, in the bins
for i = 1:M
    for j = 1:N
        r_k = floor(I(i, j)) + 1; % use floor() for integer casting
        h(r_k) = h(r_k) + 1;
    end
end
end
```

### A.2.2 Exercise 2 (b)

Listing 7: IPhisteq.m: histogram equalization.

```
function [I_eq, h_eq] = IPhisteq(I)
% Histogram equalization algorithm. From (Gonzalez, Woods - Digital Image
% Processing) chapter 3.3, Eq. 3-15. Follows book terminology.
% Arguments:
%   I: input image to apply histogram-equalization on

% Image size
M = size(I, 1); % height
N = size(I, 2); % width

% Compute histogram and normalized histogram
h = IPhistogram(I);
L = size(h, 2); % amount of intensity levels
assert(sum(h) == M * N);
p_r = h / (M * N); % normalized histogram

% Histogram mapping, i.e. transformation T
T = zeros(1, 2^8); % 8-bit image

% Map each intensity level
for k = 1:L
    % Cumulative probabilities (normalized intensity levels) up to k
    cdf_k = sum(p_r(1:k));
    T(k) = (L - 1) * cdf_k;
end
```

```
26
27 % Map input image intensities to transformed intensities
28 % Find average pixels over neighborhood
29 s = zeros(M, N);
30 for i = 1:M
31     for j = 1:N
32         r_k = floor(I(i, j)) + 1; % use floor() for integer casting
33         s(i, j) = T(r_k);
34     end
35 end
36 I_eq = uint8(s);
37 h_eq = IPhistogram(I_eq);
38 end
```

## A.3 Exercise 3

### A.3.1 Exercise 3 (a)

Listing 8: IPfilter.m: image filter.

```
1  % Function that performs a general image filter
2  % It takes the orginal image and a chosen kernel as input
3  % It outputs the enhanced image
4  function result = IPfilter(f, kernel)
5      [M, N] = size(f); %M = height, N = width
6      result = zeros(M,N);
7      [kernelHeight, kernelWidth] = size(kernel);
8      a = (kernelHeight-1)/2;
9      b = (kernelWidth-1)/2;
10     kernelCenterX = (kernelHeight+1) / 2;
11     kernelCenterY = (kernelWidth+1) / 2;
12     % loop over all the pixels in the image
13     for x=1:M
14         for y=1:N
15             % loop over the kernel size
16             for s=-a:a
17                 for t=-b:b
18                     if (x+s < 1 || y+t < 1 || x+s > M || y+t > N)
19                         % skip anything outside the orginal image.
20                         % has the same effect as zero padding.
21                         continue;
22                     end
23                     result(x,y) = result(x,y) + kernel(kernelCenterX+s, ...
24                         kernelCenterY+t) * f(x+s, y+t);
25                 end
26             end
27         end
28     end
29 end
```

### A.3.2 Exercise 3 (b)

Listing 9: IPlaplacian.m: Laplacian enhancement technique.

```
1 % Function that performs the laplacian enhancement technique
2 % It takes the orginal image and a chosen kernel as input
```

```
% It outputs the enhanced image
function result = IPlaplacian(f, kernel)
    laplacian = IPfilter(f, kernel); % create laplacian image

    % substract or add laplacian image from orginal image,
    % depending on kernel used. See book Figure 3.45, section 3.6.
    if (isequal(kernel, [0,1,0; 1,-4,1; 0,1,0]) || ...
        isequal(kernel, [1,1,1; 1,-8,1; 1,1,1]))
        c = -1;
    else
        c = 1;
    end

    result = f + c * laplacian; % add filter to image
end
```