Image Processing
lab 1

Mattijs Meiboom (s1398342)

December 11, 2011

Exercise 1 – Reducing the number of intensity levels

a. The intensity levels in our 8-bit image are represented on a scale of 0-255 and we thus have 256 levels of intensity. To resample the image to a fewer number of levels, we will take the following steps.

(a) First we convert the image-representation to use floats on the interval [0,1]

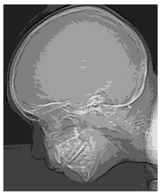
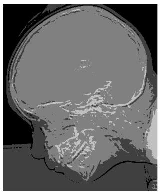
(b) We then determine the size of each level (delta, by dividing [0,1] by the number of levels)

(c) For each pixel in the image, we determine in which level the value falls (by dividing by our delta)

(d) Finally, we map each level to a value in the domain [0,255]

```matlab
function [ i ] = ipreduce( i, k )
%IPREDUCE Reduces the intensity levels of an image to 2^k
%     The output intensity can be any power of 2 between
%       2 (k = 1) and 256 (k = 8).

img = im2double(i);          % convert image to double with range [0,1]
nr  = size(img,1);           % number of rows
nc  = size(img,2);           % number of columns

levels   = 2 ^ k;
interval = 1 / levels;        % delta in [0,1] to form 2^k levels

for r = 1:1:nr
    for c = 1:1:nc

        % pixel (r,c) is at the qth level
        q = min(floor(img(r,c) / interval), levels - 1);

        % map q from [0,levels-1] to 0-255
        i(r,c) = q * (256 / levels);

    end
end

end
```

b.

			
256 levels	128 levels	64 levels	32 levels
			
16 levels	8 levels	4 levels	2 levels

Exercise 2 – Bilineair interpolation

a. The following is a function for scaling an image (using bilineair interpolation). To reach this result, we take the following steps. This should work for both scaling and shrinking, as it uses the ratio between source and target image.

(a) First, we determine the spacing of pixels in the target image, by stretching the original in the x and y direction.

(b) Then, using this spacing information, we obtain for each pixel in the target image a (non-discrete) position in the source image and it's position with respect to the pixels surrounding it.

(c) We interpolate the value of the target pixel by multiplying the value of and distance to the 4 surrounding pixels and summing these 4 values.

```
function [ i ] = ipresizebl( i, sx, sy)
%IPRESIZEBL Zooming by bilineair interpolation
%       Capable of zooming and shrinking an image by bi-linear
%       interpolation. The input to the function are the scaling
%       factors in the x- and y-dimensions, where x = y = 1.0
%       represents no scaling.

A  = im2double(i);    % convert image to double with domain [0,1]
sr = size(A,1);       % source number of rows
sc = size(A,2);       % source number of columns

tr = ceil(sr * sy);  % target image number of rows
tc = ceil(sc * sx);  % target image number of columns

dx = (sc - 1) / (tc - 1);    % spread of pixels in x direction
dy = (sr - 1) / (tr - 1);    % spread of pixels in y direction

% starting at (1,1) we distribute A over B and interpolate intermediate
```

```
% pixels using weighted bilineair interpolation.

    % TODO: http://tech-algorithm.com/articles/bilinear-image-scaling/
    % create target image
    B = zeros(tr,tc);

    % Loop. If (x,y) not at a source pixel, then interpolate
    for x = 1:1:tr
        for y = 1:1:tc

            % find 'pixel' in (sx,sy), starting at 1,1
            sx = 1 + dy * (x - 1);
            sy = 1 + dx * (y - 1);

            % = 1 + dy * 560.
            % find surrounding 4 pixels based on dx,dy
            tl = A(floor(sx),floor(sy));
            tr = A(floor(sx),ceil(sy));
            bl = A(ceil(sx),floor(sy));
            br = A(ceil(sx),ceil(sy));

            % Y = A(1-w)(1-h) + B(w)(1-h) + C(h)(1-w) + Dwh
            offset_x = sx - floor(sx);
            offset_y = sy - floor(sy);

            B(x,y) = tl * (1 - offset_x) * (1 - offset_y) ...
                   + tr * offset_x * (1 - offset_y) ...
                   + bl * (1 - offset_x) * offset_y ...
                   + br * offset_x * offset_y;
        end
    end

i = im2uint8(B);                          % convert to 8-bit

end
```

b.

c. When viewing the results at 100%, you will notice that the restored image has some artifacts introduced with respect to the original image. By first shrinking the original, we loose detail in areas that have rapidly changing values (such as edges). When restoring to the original size (especially when scaling by a large factor) these areas can become jaggy due to the lineair nature of the interpolation and the lack of detailed information.

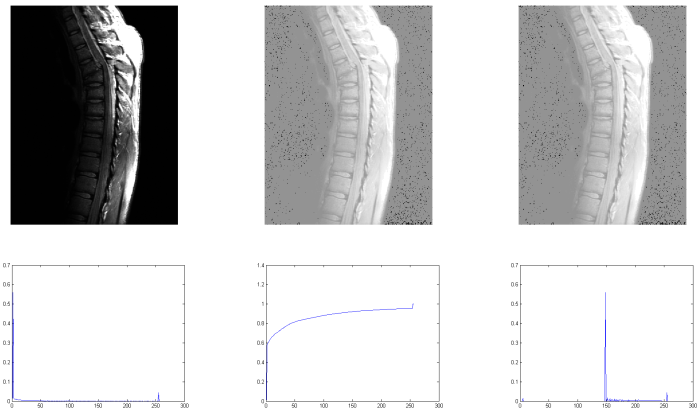
Exercise 3 – Histogram equalization

a. For the histogram equalisation, we will use a transformation function which is actually a summation based on the normalized histogram of the image. This transformation has the property that for a certain intensity level r, the output of the transformation function is the probability that a pixel has intensity level equal or lower then r. This way, intensity levels are distributed as equally as possible over the domain of r. As a result, the equalized image will consist of a more distributed set of the intensity domain (unless, as in the case of our image, one value is very dominant which will then shift towards the center of the domain).

```
function [ hist ] = iphistogram( img )
%IPHISTOGRAM yields a histogram of an 8-bit grayscale image
%       The output is normalized wrt the number of pixels in the source
%       image.
%

nr   = size(img,1);  % number of rows
nc   = size(img,2);  % number of columns
hist = zeros(1,256);  % result placeholder
frac = 1 / (nr * nc); % weight of 1 pixel

for x = 1:1:nc
    for y = 1:1:nr
        intlvl = img(y,x) + 1;              % map 0-255 to 1-256
        hist(intlvl) = hist(intlvl) + frac;
    end
end

end
```

```
function [ img ] = iphisteq( img )
%IPHISTEQ Summary of this function goes here
%   Detailed explanation goes here

% Image-size and histogram
nr = size(img, 1);
nc = size(img, 2);
pr = iphistogram(img);
sk = zeros(1,256);

% Calculate transformation mapping by summation
sk(1) = pr(1);
for x = 2:1:256
    sk(x) = sk(x - 1) + pr(x);
end

% Tranform image
for x = 1:1:nc
    for y = 1:1:nr
        intlvl = img(y,x) + 1;
        img(y,x) = 256 * sk(intlvl);
```

```
        end
end

end
```

b. 

c. The reason that the result for equalizing once or twice are identical is that, in the equalized image, intensity levels have already been distributed according to the transformation function and thus when calculating a transformation for the second pass, the probability of a pixel having intensity level smaller or equal to r (r in [0..1]) is actually r.