Image Processing
lab 1
Group 20

Zhang Yuanqing(s3811468) Loran Oosterhaven

December 2, 2019

Exercise 1 – Downsampling, upsampling, and zooming

a. The function is implemented by removing the pixels which the mod of the index and factor is not zero in both row and column directions. This operation could reduce the resolution of an original image. The factor and image name are two input parameters of the function. The illustration in Figure1 shows how the downsampling is done with the factor is 2. Listing1 shows our implementation of the downsampling function. There are two input parameters. The first one is the name of the image to upsample, and the second one is the factor used in upsampling. After sampling the function uses `imwrite` to generate the upsampled image to local disk.

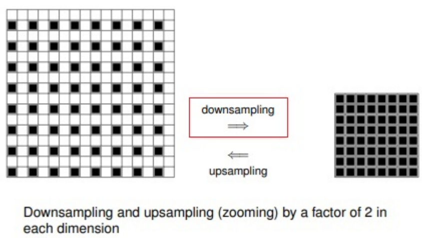


Figure 1: Downsampling

```
%This is the implementation of function IPdownsample

function res = IPdownsample(imName, factor)

toShrink = imread(imName);

%shrink the original image with a facor
%picking pixels every factor from the original one.
shrunkImage = toShrink(factor:factor:end, factor:factor:end);
imwrite(shrunkImage, 'shrunk.tif');

res = shrunkImage;

```

```
end
```

Listing 1: IPdownsample

b. Applying the function `IPdownsample` on Figure2 with factor is 4, we call `IPdownsample('cktboard.tif', 4)`. We got the shrunk image shown in Figure3. The length and height of the shrunk image are only $\frac{1}{4}$ of the original image becaue $\frac{3}{4}$ of the original pixels are removed.

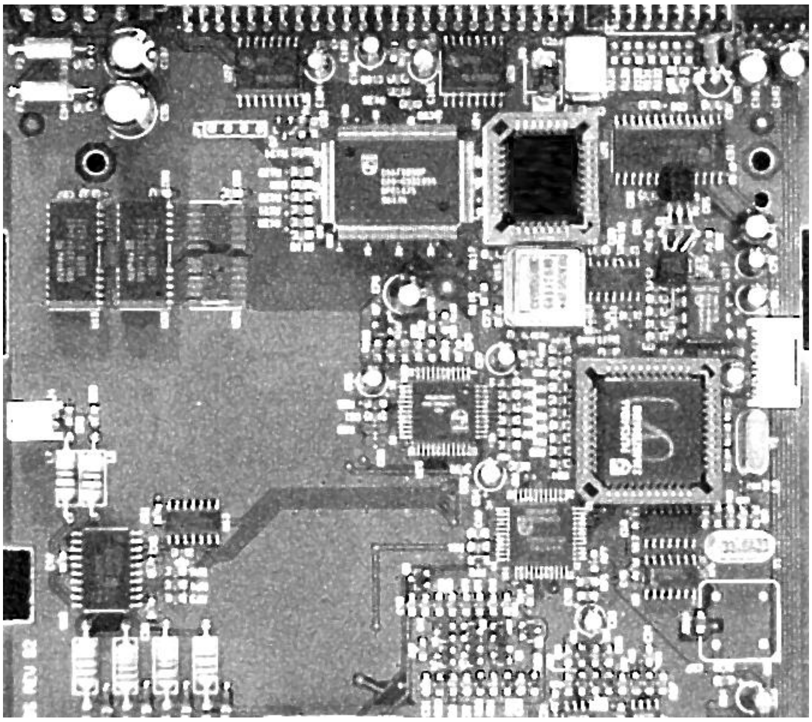


Figure 2: Original cktboard image

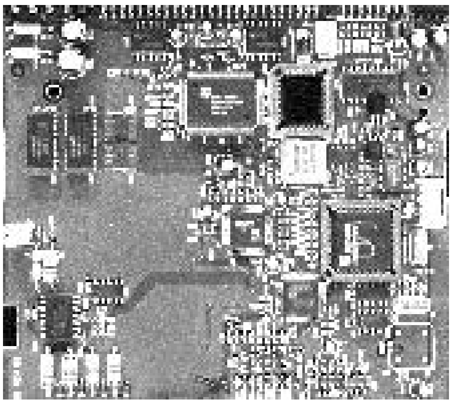


Figure 3: Shrunk cktboard.tif by factor 4

c. The IPupsample is implemented by inserting (factor -1) numbers of pixels of zero value into the between every original pixels in both row and column directions. Figure4 shows the process of upsampling of an image by using factor2. Listing shows our implementation of the upsampling function. Two input parameters are required. The first one is the image to make upsampling on it and the second one is the factor. After upsampling the generated image will be stored to local disk by using `imwrite`. Listing2 shows the implementation of `IPupsample`.

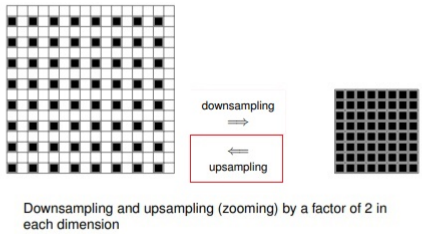


Figure 4: Upsampling

```
%This is the implementation of function IPupsample

function res = IPupsample(imName , factor)

toUp = imread(imName);

uppedImage = repelem(toUp, factor, factor);

zoomedSize = size(uppedImage);

for i=1:zoomedSize(1)
    for j=1:zoomedSize(2)

        if mod(i, factor) ~= 0 || mod(j, factor) ~= 0
            uppedImage(i,j) = 0;
        end

    end
end

imwrite(uppedImage, 'upped.tif');

res = uppedImage;
end
```

Listing 2: IPupsample

d. To apply the function `IPupsample` on Figre2 by using factor 4 , we call IPupsample('cktboard.tif', 4). We got the upsampled image shown in Figure5. The length and height of the shrunk image are 4 times of the original image. Because in front of each pixel, 3 pixels with zero value are inserted in both row and column direction.

The details of the upsampled image is not clear here. So in appendix I put a scaled upsampled cktboard.tif to give a clearer display of the result.

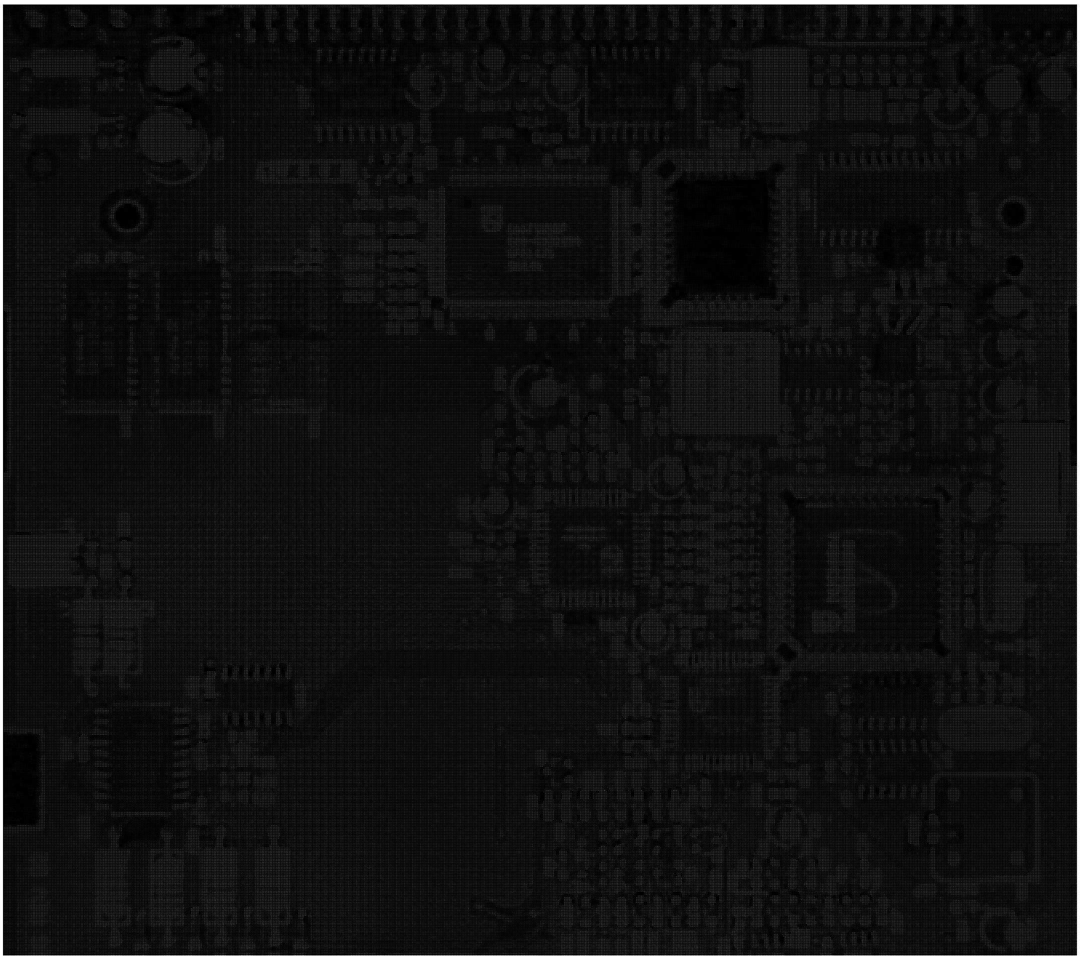


Figure 5: Upsampled cktboard.tif by factor 4

e. The `IPzoom(imName, factor)` takes two parameters: `imName` to indicate the image name to process and `factor` to indicate the factor used to downsample. It's implemented by replicating pixels to the number indicated by factor in row and column. So the final area of the zoomed image is (factor * factor) times bigger than the original one. Both factor times longer length than the original image in both row and column directions. In Figure6, there is an example of zoom operation with factor 2. Listing 3 shows the implementation of `IPzoom(imName, factor)`.

Here the function repelem is used to replicate the pixels in an efficient way.

```
%This is the implementation of IPZoom




```

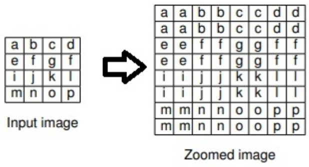


Figure 6: Zooming operation

```
function res = IPzoom(imName, factor)

toZoom = imread(imName);

zoomedImage = repelem(toZoom, factor, factor);

imwrite(zoomedImage, 'zoomed.tif')

res = toZoom

end
```

Listing 3: IPzoom

f. Applying IPzoom on Figure2 with factor is 4, we call `IPzoom('cktboard.tif', 4)`. The result image is shown in Figure7. The length and height of the zoomed image are also 4 times of the original image.

The difference between Figure5 and Figure7 is mainly the values of the inserted pixels. Because in Figure5 the inserted pixel values are 0 so the image is very dark. But in Figure7 the inserted pixel values are determined by the original pixels to be replicated so the zoomed image is much more clearer than the upsampled one. Obviously the zoomed one is much more readable than the upsampled one.

g. Resolution refers to the number of pixels in an image. So by using the code IPzoom('shrunk.tif', 4), we can zoom the shurnk.tif back to the original resolution with the factor 4. The zoomed shrunk.tif is shown in Figure8.

From Figure8, compared to the original cktboard.tif, obviously we can see it is grainy, not clear, and many details are lost. I think this happens because firstly we used IPdownsample('cktboard.tif', 4) to generate shrunk image. In this function we only pick the pixels whose indices mod factor is zero. So only 1 of (factor * factor) of the original pixels are reserved in shrunk.tif. Then after we use the IPzoom('shrunk.tif', 4), we only replicate the pixels in the shrunk.tif which contains much less details than the origianl one. So finally though we got the image which has same resolution as the original one, the quantity of the new zoomed one is worse than the origianl one.

So we could say that the downsampling and zooming are not inversable.

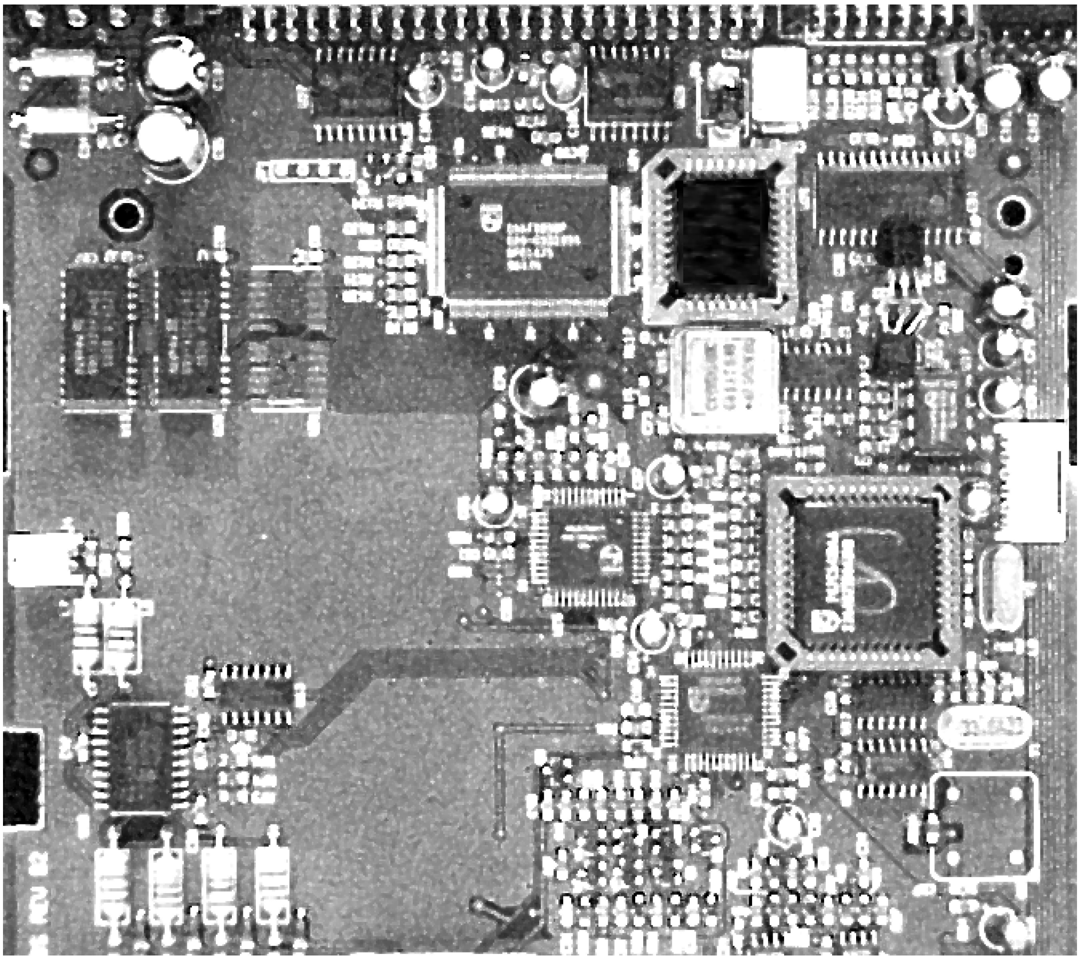


Figure 7: Zoomed cktboard.tif by factor 4

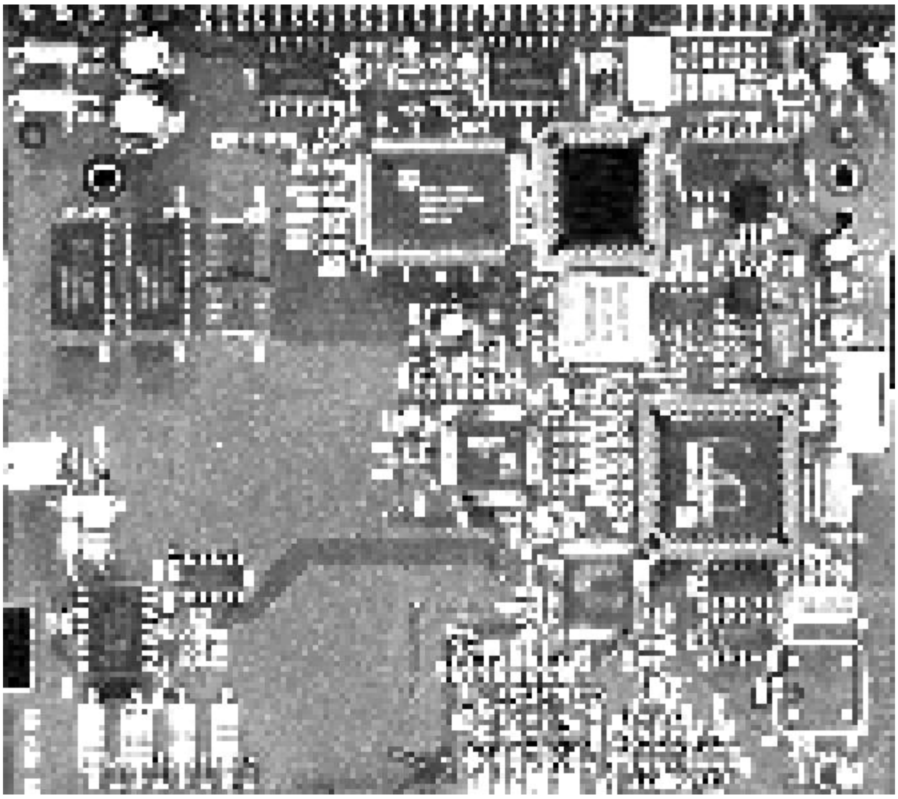


Figure 8: Zoomed shrunk.tif by factor 4 to the original resolution

Exercise 2 – Histogram equalization

a. IPhistogram is implemented by counting the numbers of each grey level. Only one input parameter is required - imageName. After read the image I iterated the image and record the numbers of each grey level. Then store the number of them into an array. Finally give the histogram picture of it. Listing4 shows how the function is implemented.

```
%This file is used to implementat IPhistogram

function res = IPhistogram(imName)

im = imread(imName);
originalSize = size(im);

rawHistogram = zeros(1,256, 'uint32');


for i= 1:originalSize(1)
    for j = 1:originalSize(2)
        va = im(i, j);
        count = rawHistogram(1, va + 1);
        count = count + 1.0;
        rawHistogram(1, va + 1) = count;
    end
end

bar( 0:255, rawHistogram );
res = rawHistogram;

end
```

Listing 4: IPhistogram

b. To implement IPhisteq, we need to use the probability of occurrence of intensity level r_k:

$$P_r(r_k) = \frac{n_k}{MN} \tag{1}$$

where MN is the total number of pixels in the image and n_k is the number of pixels that have intensity r_k.

The discrete form of the transformation which is what we want in this function is:

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

By combining the equation(1) and (2), we can have our implementation of this funciton in Listing 5. Here we also give the histogram of processed image after histogram equalization which is helpful to get a better insight of the processing.

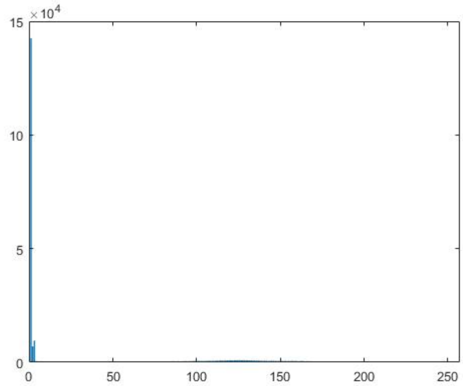
```
%This file is used to implement IPhisteq



```

```
function res = IPhisteq(imName)

im = imread(imName);
originalSize = size(im);
MN = originalSize(1) * originalSize(2);
histo = IPhistogram(imName);
L = 255; % L
prk = double(histo) / MN;

toHE = imread(imName);


for i = 1:originalSize(1)
    for j = 1:originalSize(2)
        originalValue = toHE(i,j);
        HEvalue = round((L - 1) * sum(prk(1:originalValue + 1)));
        toHE(i,j) = HEvalue;
    end
end

imwrite(toHE, 'hemoon.tif');

HEhisto = IPhistogram('hemoon.tif');

res = toHE;

end
```

Listing 5: IPhisteq

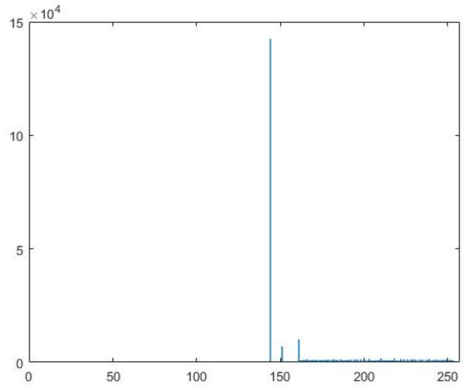
c. Applying the IPhisteq on Figure9(a), we could get the processed in Figure9(c). Also the histogram of Figure9(a) is shown in Figure9(b) and histogram of Figure9(c) is shown in Figure9(d). By examining the Figure9(b), we found that there is biased toward the lower end of the intensity scale. This type of histogram tells us that the image is a candidate for histogram equalization, which will spread the histogram over the full range of intensities, thus increasing visible detail. The more balanced result histogram in Figure9(d) confirms this. The visible detail is much improved than the original.

(a) original moon



(b) histogram of original moon

(c) Result of histogram equalization



(d) histogram of image(c)

Figure 9: Histogram Equalization.

Exercise 3 – Spatial filtering

a. Here we assign the mask is a 3*3 matrix and also I use 0 paddings to the original image. Which meansI will add 1 pixel width, original image height padding to the left and right edge of the original image. Also1 pixel height and original image height +2 width padding to the top and bottom edge. This could avoidout of range errors in during the calculation.

The spatial filtering of an image of size $M \times N$ with a kernel of size $m \times n$ is given by following equation:

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x+s, y+t) \tag{3}$$

where x and y are varied so that the center (origin) of the kernel visits every pixel in f once. For a fixed value of (x, y),

The implementation of `IPfilter` here we use function signature as `IPfilter(imName, mask)`. The `imName` refers to the image name to be filtered, and the `mask` refers to the maks used to filter where it is a 3 * 3 mask here.

The implementation code is shown in Listing6.

```
%This is the implementation of UPFilter

function res = IPfilter(imName, mask)

maskSize = size(mask);
if maskSize(1) ~= 3 || maskSize(2) ~=3
    error('use a 3 * 3 mask please!')
    res = 0
end

originalImage = imread(imName);

%Using the original size to generete paddings
originalSize = size( originalImage);
leftPadding = zeros(originalSize(1), 1);
rightPadding = zeros(originalSize(1), 1);
topPadding = zeros(1,originalSize(2)+2);
bottomPadding = zeros(1,originalSize(2)+2);

%Adding paddings
paddedImage = cat(2, leftPadding, originalImage);
paddedImage = cat(2, paddedImage, rightPadding);
paddedImage = cat(1, topPadding, paddedImage);
paddedImage = cat(1, paddedImage, bottomPadding);


toFilter = imread(imName);

for i = 2: originalSize(1) + 1
    for j = 2:originalSize(2) + 1
        gFinal = 0;
        %adding the pixels neibors to the pixel with weights
        for a = 1:3
            for b = 1:3
                gFinal = gFinal + paddedImage(i + (a - 2), j + (b - 2)) * mask(a, b);
            end
        end
        toFilter(i-1, j-1) = gFinal;
    end
end

```

```
imwrite(toFilter, 'filtered.tif');%write to local
res= toFilter;

end
```

Listing 6: IPfilter

The code

b. The Gaussian kernels of the form is:

$$w(s,t) = G(s,t) = Ke^{-\frac{s^2+t^2}{2\delta^2}} \tag{4}$$

where the s,t are the coordinator based on the center pixel.

So in this function IPlaplacian(imName, K, delta) we just need three input parameters : imName indiates the image to be processed, K and delta are all the constant variables for the Gaussian distribution equation. This function is just an extension of `IPfilter`. Just in there we use the two other input parameters to generate a mask. Listing shows the implementation of IPlaplacian(imName, K, delta).

```
%This file is used to implement IPlaplacian

function res = IPlaplacian(imName, K, delta)


mask = zeros(3);

for i = -1:1
    for j = -1:1
        mask(i+2, j+2) = K * power(exp(1), -((power(i,2) + power(j,
            2)) / (2 * power(delta,2))));
    end
end

disp(mask / sum(mask(:)));
FinalMask = mask / sum(mask(:));

res = IPfilter(imName, FinalMask);

end
```

Listing 7: IPLaplician

c. By using the IPlaplacian we got the Figure10. It's blurred by a Gaussian kernel. The figure shows that after using the laplacian filter, the effect on image got blurred. By using the laplacian, we can avoid poor approximations to the blurring characteristics of lenses. Also we can avoid that box filters favor blurring along perpendicular directions. Because the The kernels of choice in laplacian are circularly symmetric. We could see that this image is blurred well, each of the neighbor pixels gives the different weights but the nearer to the center, the bigger the weight is but the pixels with same distance to the center give the same weight. So the blur is effected more significant by the nearby pixels to avoid the unbalanced bad blur.

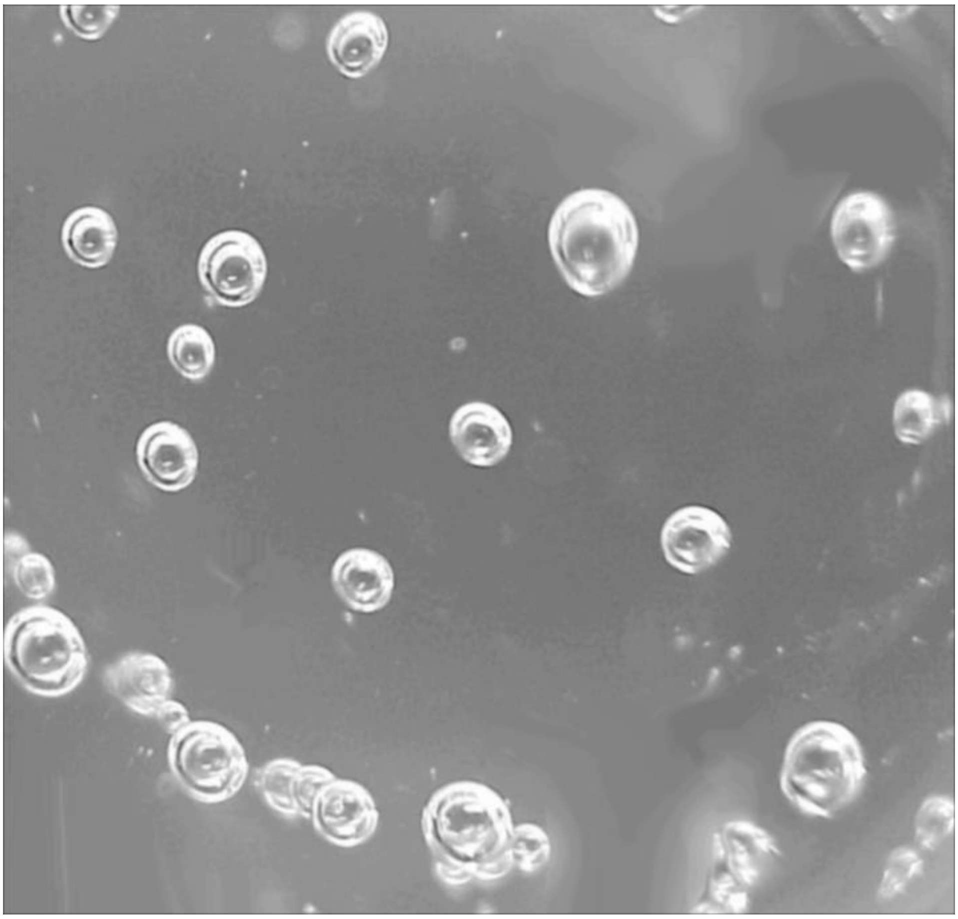


Figure 10: Applying IPlaplacian on bubble.tif

Appendix A Individual Contributions

We did the the questions individually.

Zhang Yuanqing took care of the Exercise1, Exercise3a and Exercise3c. He made the program design, program implementation, answering questions posed and writing the report of these 3 questions by himself.

Loran Ooosterhaven also did the Exercise2 and Exercise3b all by himself. The program design, program implementation, answering questions posed and writing the report of his part is totally done by himself.

Appendix B Scaled image after upsampling