Practical Work - Compression, Analysis and Visualization of Multimedia Content

EIT Digital Master in Data Science, 2º year – Multimedia and Web Science for Big Data

Yolanda de la Hoz Simón - yolanda93h@gmail.com
Ignacio Uyá Lasarte - i.uya.lasarte@gmail.com

Introduction

Compression is an essential operation that allow us an easy storage, transmission and manipulation of huge quantities of data.

In this assignment, it is implemented and discussed two essential steps in the compression process; the quantization and transformation. At the end of each chapter it is also discussed the loss of information with different tests applied in different images and also different parameters.

Quantization algorithm

Quantization is a process that allow us to represent a digital signal with lower levels or lower bits per pixel for compression purposes. In this process it is performed an optimal adaptation of the original signal to the levels defined in a codebook or set of clusters in order to introduce less error or loss of information in the signal.

For this assignment, it is proposed an uniform scalar quantization. This algorithm assigns each value of the signal to the middle point of the nearest neighbour computed in a set of clusters or partitions of the original signal.

```
function quantized_matrix = quantize_matrix(matrix, new_bitsize)
    quantized_matrix = zeros(size(matrix,1),size(matrix,2),'uint8');
    for row = 1:size(matrix,1)
        for col = 1:size(matrix,2)
            quantized_matrix(row,col) = quantize(matrix(row,col), new_bitsize);
        end
    end
end

function quantization = quantize(pixel, new_bitsize)
    dif_levels = 2^(8-new_bitsize);
    quantization=floor(double(pixel)/dif_levels);
end
```

Figure 1. Uniform Scalar Quantization - Code implementation in Matlab

In the Figure 1 it is shown the implementation of this algorithm. This algorithm receives as arguments the input image and the number of bits in which to store it.
We are assuming that the original image contains 256 levels (or 8 bits) in grayscale.
In the implementation of this algorithm we are trying to make things as efficient as possible and make some assumptions such as: the representative of a cluster is

always the middle value and when given a number of bits, it will use all the possible levels (for example, given 4 bits, it will use 16 levels even if it would be possible)

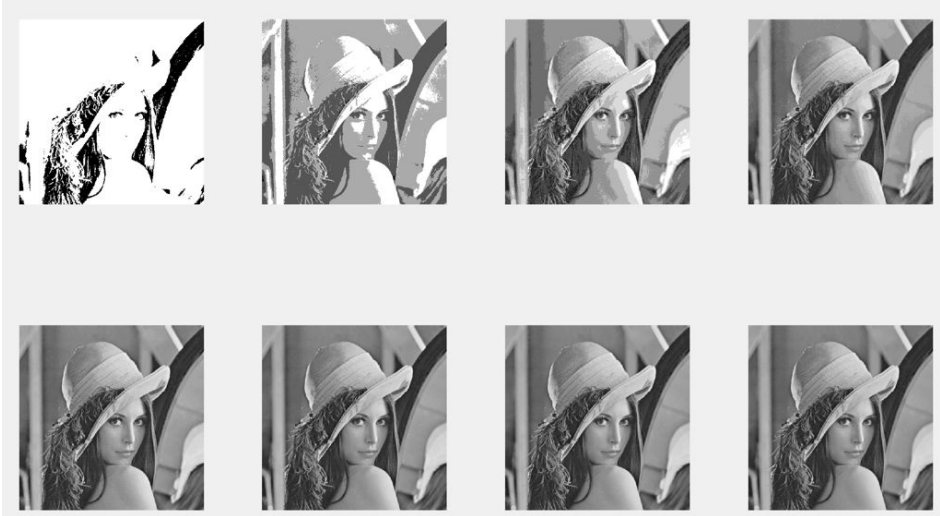

Figure 2 - Output images after applying quantization algorithm with bits from 1 to 8

This images are actually stored in the designated number of bits, so for bits=1, there are only 0's and 1's (2 levels). The image looks ok thanks to the function "imshow" since it is possible to indicate the number of levels the image has. For example, in the case of 1-bit quantization, we can use imshow(image, [0,1]) and it will look black and white instead of just black (because we are using 255 by default).

We need another function for the actual de-quatization of the image:

```
% dequantize_matrix:
% Uniformly projects a matrix with a number of levels (2^old_bitsize)
% into a new matrix with a number of levels (2^new_bitsize)
function dequantized_matrix = dequantize_matrix(matrix, old_bitsize, new_bitsize)
   avg_point = 2^(new_bitsize-old_bitsize-1);
   dequantized_matrix = zeros(size(matrix,1),size(matrix,2),'uint8');
   for row = 1:size(matrix,1)
      for col = 1:size(matrix,2)
         dequantized_matrix(row,col) = dequantize(matrix(row,col), avg_point);
      end
   end
end

function dequantization = dequantize(pixel, avg_point)
   dequantization=  avg_point*(pixel + 1);
end
```

Figure 3. Uniform Scalar De-quantization - Code implementation in Matlab

The results of these de-quantization into 256 levels looks very similar to the imshow implementation but we are using the middle value of each partition as a

representative and imshow is probably using the corner values (0 and 255 in the case of 1-bit to 8-bit instead of 64 and 192)

Figure 4 - Output images after applying De-Quantization algorithm with bits from 1 to 8

We can also plot the different input/output functions using quantize(0:255) and dequantize(quantize(0:255)) in order to easily see the representant of the different clusters:

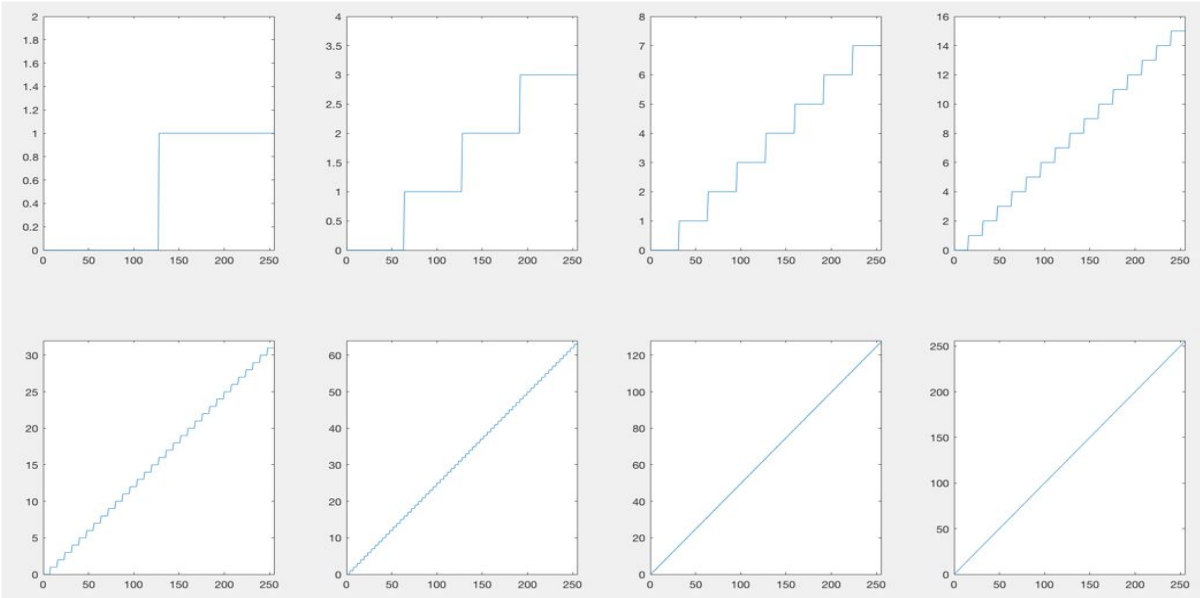

Figure 5 -Plot of input/output function with "quantize(0:255)" from 1 to 8 bits

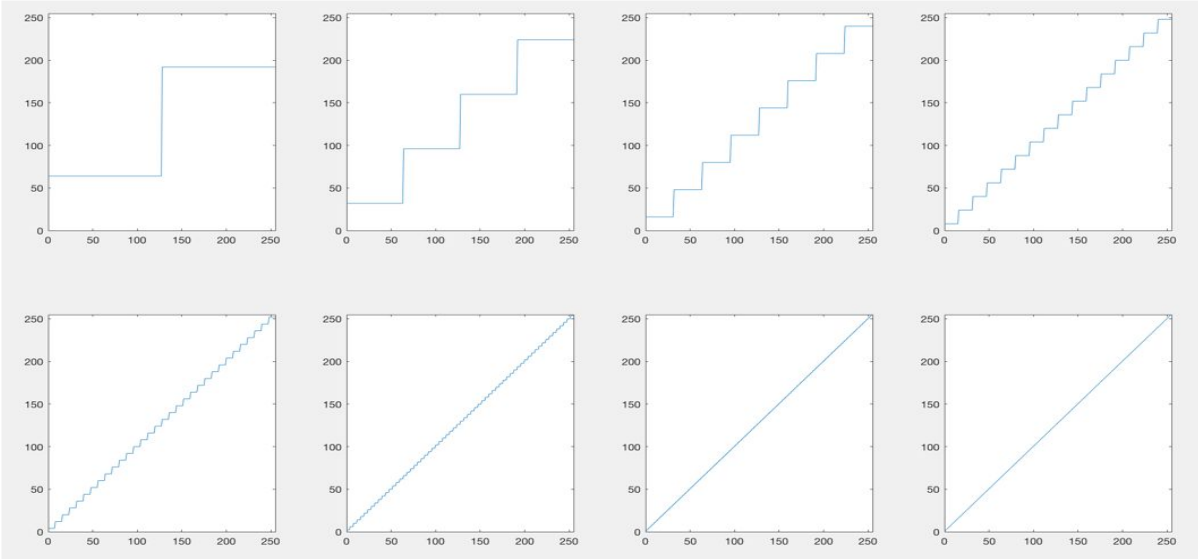

Figure 6 -Plot of input/output function with "dequantize(quantize(0:255))" from 1 to 8 bits

Another interesting metric we can calculate is the Mean Square Error for each number of bits used (taking into account that levels = 2^bits or inversely bits=log2(levels). In this case we can calculate the MSE for each image after being quantized/dequantized to a given number of bits.

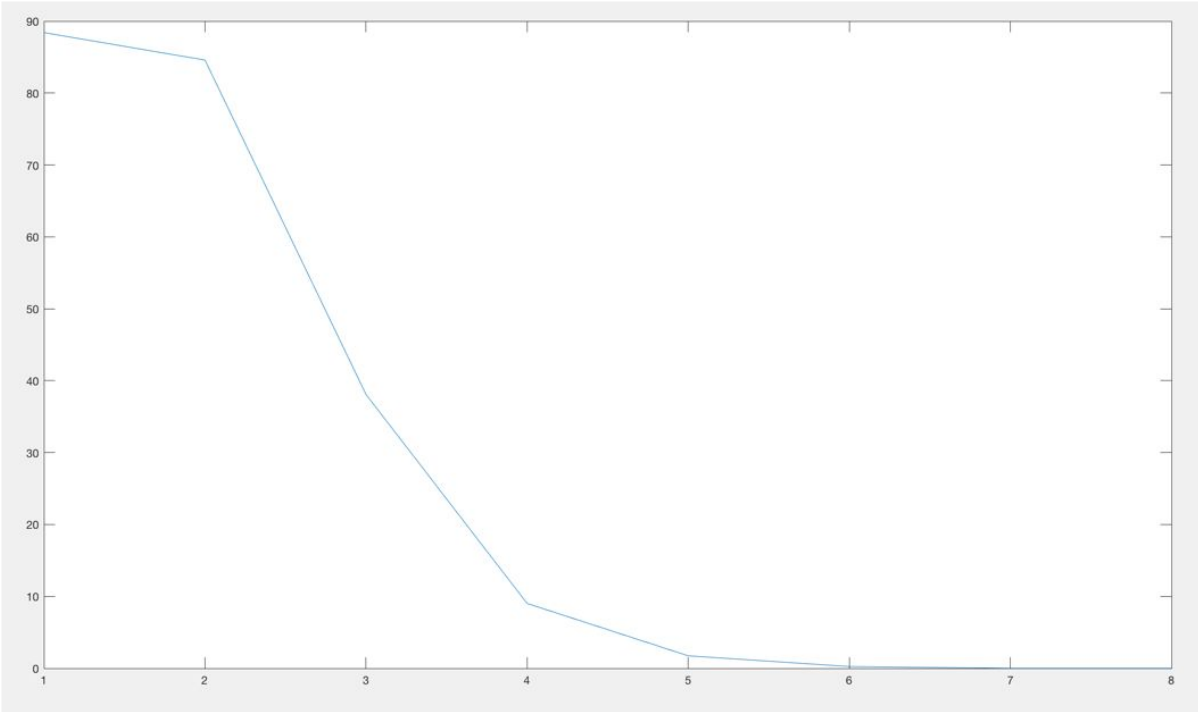

Figure 7 - Plot of the MSE (y) depending on the number of bits (x) used to quantize the image "lena_gray_512"

We can see how the MSE decreases as the number of bits (and levels) increases, up to 8 bits (or 256 levels) where the image is exactly the same and the MSE is 0. Still, when using 6 bits, we can see how the MSE is almost 0, meaning that we have not lose --almost-- any information in the image. Still it is important to take into notice

that MSE as a measure of distortion does not directly translate to perceived distortion, as the human eye weights every kind of distortion differently, and thus an image with a greater MSE could be perceived a less distorted than another with lower MSE if judged by a human. MSE is a purely arithmetic metric and does not take into account the internal geometry of the image.

Haar Wavelet algorithm

Haar Transform:

```
% haar_transform:
% Given an input greyscale image of 256 levels, apply the haar transform to it.
% Will apply the transform first to the rows and then to the columns
function haar_matrix = haar_transform(original_matrix)
        temp_matrix = haar_pass(original_matrix);
        temp_matrix = transpose(temp_matrix);
        temp_matrix = haar_pass(temp_matrix);
        haar_matrix = transpose(temp_matrix);
end

function haar_matrix = haar_pass(matrix)
        haar_matrix = zeros(size(matrix,1),size(matrix,2),'double');
        half_col = size(matrix,2)/2;
        for row = 1:size(matrix,1)
        i = 1;
        for col = 1:2:size(matrix,2)
        haar_matrix(row,i) = (double(matrix(row,col)) + double(matrix(row,col+1)))/2.0;
        haar_matrix(row,i+half_col) = (double(matrix(row,col)) -
double(matrix(row,col+1)))/2.0;
        i = i + 1;
        end
        end
end
```

Reverse Haar Transform:

```
% haar_reverse:
% Given an input haar_transform image of 256 levels, undo the transform.
function haar_matrix = haar_reverse(original_matrix)
   temp_matrix = transpose(original_matrix);
   temp_matrix = revert_haar(temp_matrix);
   temp_matrix = transpose(temp_matrix);
   haar_matrix = revert_haar(temp_matrix);
end

function haar_matrix = revert_haar(matrix)
    haar_matrix = zeros(size(matrix,1),size(matrix,2),'double');
    half_col = size(matrix,2)/2;
```

```
    for row = 1:size(matrix,1)
        i = 1;
        for col = 1:half_col
            haar_matrix(row,i) = double(matrix(row,col)) + matrix(row,col + half_col);
            haar_matrix(row,i+1) = double(matrix(row,col)) - matrix(row,col + half_col);
            i = i + 2;
        end
    end
end
```

The results of using this transformation and its reverse can be seen in the following images:

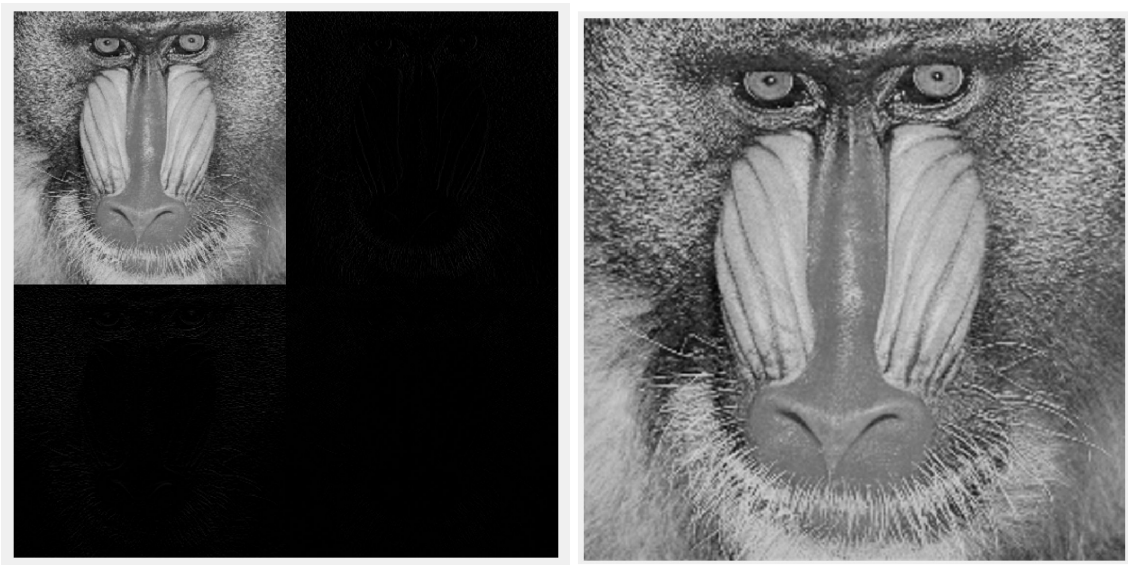

Figure 8 - Baboon after using the Haar transform and the Haar reverse transform.

As we can see, after the Haar transform we obtain an image composed of 4 quadrants, in the first one the baboon can be clearly seen since the only filters to be applied have been in order to average the pixels (2 low-pass filters) On the other hand, the quadrants 2 and 4 have received low-pass and high-pass filtering (LH in one and HL in the other). Lastly, the 3rd quadrant has received 2 high-pass filters and shows no recognizable features while quadrants 2 and 4 show some contours.

We have also implemented the Haar filter with multiple depth (code in the Annex at the end of the report) which is interesting because the successive passes of the transform are only performed in the LL (1st quadrant) part of the image.

In the following figure we show an example of a triple Haar transform as well as its reverse. Please note that it is very difficult to identify any features in the LH/HL part of the images due to their small size and it may be necessary to increase the screen brightness to the maximum in order to see anything.

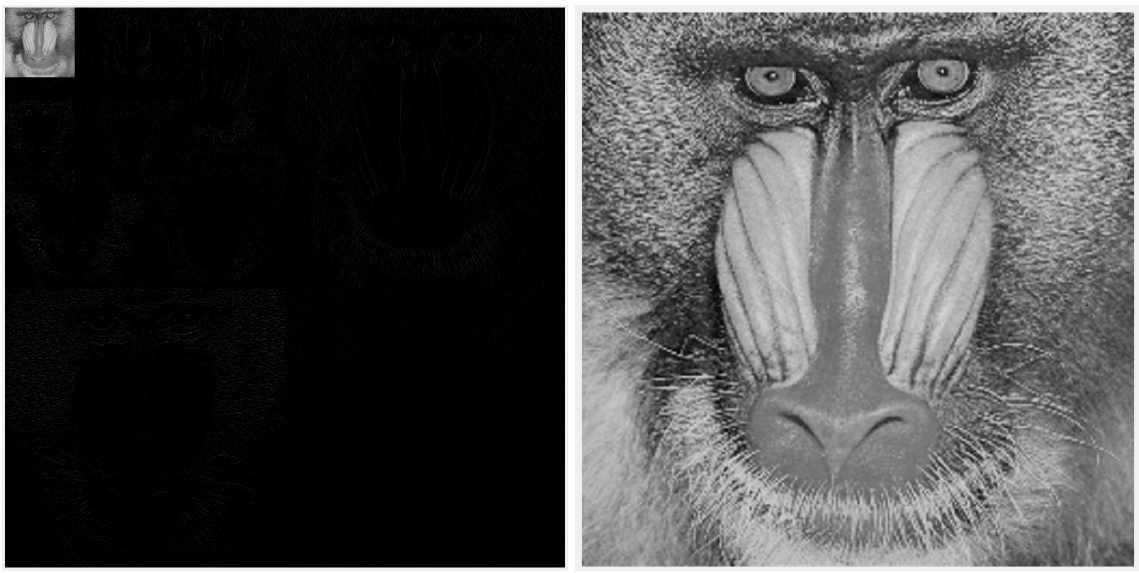

Figure 9 - Baboon after using the triple Haar transform and the triple Haar reverse transform.

As we can see in both cases, there is some loss of information due to the compression of the image. The benefit of this transform comes from the HL/LH/HH part of the images, which usually require much lesser space (bits) to encode and transmit/store since the number of different values (or levels) in those quadrants should be considerably lower than those the LL quadrant (255 values/levels).

Laplacian pyramid algorithm

The laplacian pyramid algorithm is a range transformation algorithm used in the compression process. This algorithm uses a representation of the image by a set of the different frequency-band images. For example, if we have 3 layers of images the first image will represent low frequency band (smooth detail), the second image will represent middle frequency band (some detail), and the last image will represent high frequency band (finest detail), as it can be seen in the figures 8 and 9.

For the implementation of the laplacian algorithm, first it is build the gaussian pyramid and then it is computed the laplacian pyramid by subtracting every two sequential layers. The results of both algorithms with a pyramid of 5 layers can be shown in figure 8 and 9.

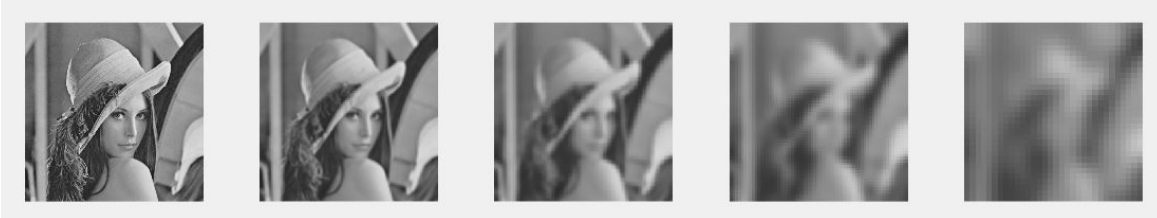

Figure 8. Gaussian Pyramid

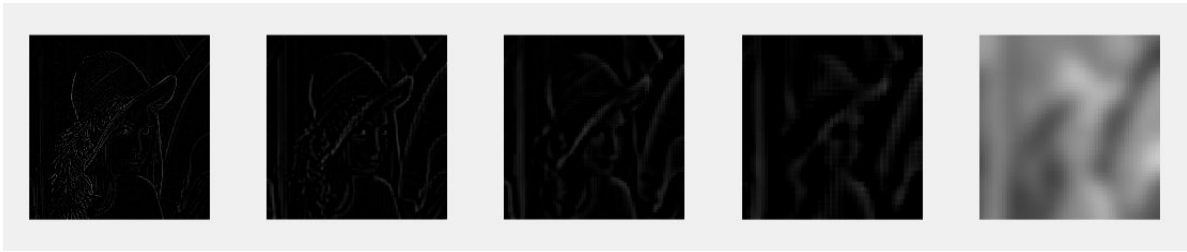

Figure 9. Laplacian Pyramid - Analysis

The gaussian pyramid function receives as parameters the image *input_img* and the number of layers *N* and returns the 5 decomposition layers *decomposition_layers* . In the implementation of this algorithm first it is stored the original image as first layer and for the rest of the layers it is convolved the image with a gaussian filter and downsampled the image subsequently. The implementation of this function can be seen in *Figure 10*.

```
function [ decomposition_layers ] = gaussianPyramid( input_img, N )
%gaussianPyramid Build the gaussianPyramid
decomposition_layers=cell(1,N);
decomposition_layers{1}=input_img; % first layer is the original image

for k = 1:N-1
gauss_img = gaussianFilter(decomposition_layers{k}, 3);
aux = downsample(gauss_img,2);
decomposition_layers{k+1}= downsample(aux',2)';
end

end
```

Figure 10. Gaussian Pyramid - Code implementation in Matlab

On the other hand, the laplacian pyramid function receives as parameters the image *input_img* and the number of layers *N* and returns the 4 laplacian layers *laplacian_layers.* For the implementation of this algorithm first it is computed the decomposition_layers of the gaussian pyramid, as it is described above, and then it is subtracted and upsampled every two sequential layers to build the different laplacian layers. Finally, the last layer is build with the gaussian convolution of the N-1 layer.

```
function [ laplacian_layers ] = pyramidLaplacian( input_img, N )
%pyramideLaplacian laplacian pyramid
% N decomposition layers

laplacian_layers=cell(1,N);
% Convolve the filtered image with the gaussian filter
decomposition_layers = gaussianPyramid( input_img, N );

for k = 1:N-1

img_upsampled = upsample(decomposition_layers{k+1});
img_upsampled = upsample(img_upsampled')';
```

```
% Construct N Laplacian descomposition layer
laplacian_layers{k}= uint8(decomposition_layers{k}) - uint8(img_upsampled);
end
laplacian_layers{N} = gaussianFilter(laplacian_layers{N-1}, 3);

end
```

Figure 11. Analysis Laplacian Pyramid - Code implementation in Matlab

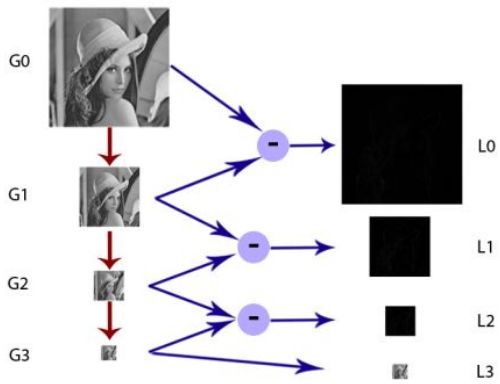

Figure 12. Laplacian Pyramid - Construction of the N Laplacian descomposition layers

In the figure 12 it is shown the process used to build the Laplacian Pyramid or the pyramid of band-pass filtered image. As it can be seen, each layer of this pyramid is made from subtract two consecutive layer from Gaussian Pyramid together. In order to subtract, the lower later need to be upsampled to be the same size to the upper layer

Example 2D signal synthesis

For the reconstruction or synthesis of the original image it is implemented the function *laplacianReconstruction* receiving as parameters the laplacian layers and the number of layers. This function reverts the operation computing the upsampled image and adding two sequential layers iteratively until obtain the original image. The result of this process and the implementation can be seen in the figure 12 and 13.

Figure 13. Synthesis Laplacian pyramid

```
function [ original_img ] = laplacianReconstruction( laplacian_layers, layers )
%laplacianReconstruction

img_upsampled = upsample(laplacian_layers{layers});
img_upsampled = upsample(img_upsampled')';
for k = layers-1:-1:2

original_img = uint8(laplacian_layers{k-1}) + uint8(img_upsampled);
img_upsampled = upsample(original_img);
img_upsampled = upsample(img_upsampled')';
end

end
```

Figure 14. Synthesis Laplacian pyramid - Code implementation in Matlab

Annex I. Code used to generate some images

In this annex we would like to gather the code used to make some of the images in this document. Since it is not an “official” part of the document we did not see fit to add the code to the main document, but we believe it is interesting to show the scripts used to generate some of the images nonetheless .

Quantization

Input/Output function with quantize:

```
for j = 1:8
   output=zeros(256,1);
   for i = 0:255
      output(i+1,1) = quantize(i,j);
   end
   subplot(2,4,j,'replace')
   plot(0:255,output(:,1));
   axis([0,255,0,2^(j)])
end
```

Input/Output function with dequantize(quantize):

```
for j = 1:8
   output=zeros(256,1);
   delta = 2^(8-j);
   for i = 0:255
      output(i+1,1) = dequantize(quantize(i,j),delta);
   end
   subplot(2,4,j,'replace')
   plot(0:255,output(:,1));
   axis([0,255,0,255])
end
```

MSE chart:

```
results = zeros(8,1);
for i = 1:8
   temp = dequantize_matrix(quantize_matrix(lena_gray_512,i),i,8);
   D = abs(lena_gray_512-temp).^2;
   MSE = sum(D(:))/numel(lena_gray_512);
   results(i,1) = MSE;
end
plot(1:8,results);
```

Harr transform

“Composed” Transform:

```
function haar_matrix = haar_transform_multilevel(original_matrix, n_transforms)
    haar_matrix = original_matrix;
    for i = 1:n_transforms
        rows = 1:size(haar_matrix,1)/i;
        columns = 1:size(haar_matrix,1)/i;
        haar_matrix(rows,columns) = haar_transform(haar_matrix(rows,columns));
    end
end
```

“Composed” Reverse Transform:

```
function haar_matrix = haar_reverse_multilevel(original_matrix, n_transforms)
    haar_matrix = original_matrix;
    for i = n_transforms-1:-1:0
        rows = 1:(size(haar_matrix,1)/2^i);
        columns = 1:(size(haar_matrix,1)/2^i);
        haar_matrix(rows,columns) = haar_reverse(haar_matrix(rows,columns));
    end
end
```

Laplacian pyramid

Gaussian Filter:

```
function [ output_img ] = gaussianFilter( input_img, sigma )
%gaussianFilter Build a 2D Gaussian Filter
if nargin==1, sigma=3; end
x = -5:5; % sigma = 1, support = ?2:2 (N=5)
h = exp(-(x.^2)/(2*sigma.^2)); % Gauss expression 1D
h = h/sum(h); % Normalize; sum=1
G = h' * h; % 2D mask

output_img = imfilter(input_img,G,'same'); % convolve image

end
```

Script to plot the results:

```
img2 = imread('lena_gray_512.tif');

figure(1);
layers=5;
gaussian_layers= gaussianPyramid(img2,layers);

for k = 1:layers
subplot(1,layers,k),imshow(gaussian_layers{k});
end

layers=5;
laplacian_layers= pyramidLaplacian(img2,layers);

figure(2);
for k = 1:layers
subplot(1,layers,k),imshow(laplacian_layers{k});
end

figure(3);
%synthesis
original_img = laplacianReconstruction(laplacian_layers, layers);

imshow(original_img);
```