Practical Work 2 - 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

Codec **3**

Motion Estimation **6**

Block Matching Algorithm 6

Motion Compensation 9

Distortion measurement 11

Stereoscopy **12**

Annex I. Code used to generate some images **14**

Codec 14

Motion Estimation 17

Stereoscopy 19

Source code **23**

Codec

The codec is composed of a Haar Transform and a uniform scalar quantizer. Later we will use Huffman coding as entropy coding to compare it with the minimum entropy possible.
The initial step is to load the image “lena_gray_512”

Figure 1. Codec - Initial version of Lena_gray_512

After loading the image we will use the Haar transformation twice: The first time over the whole image, and the second time only over the LL (low frequency) part of it.

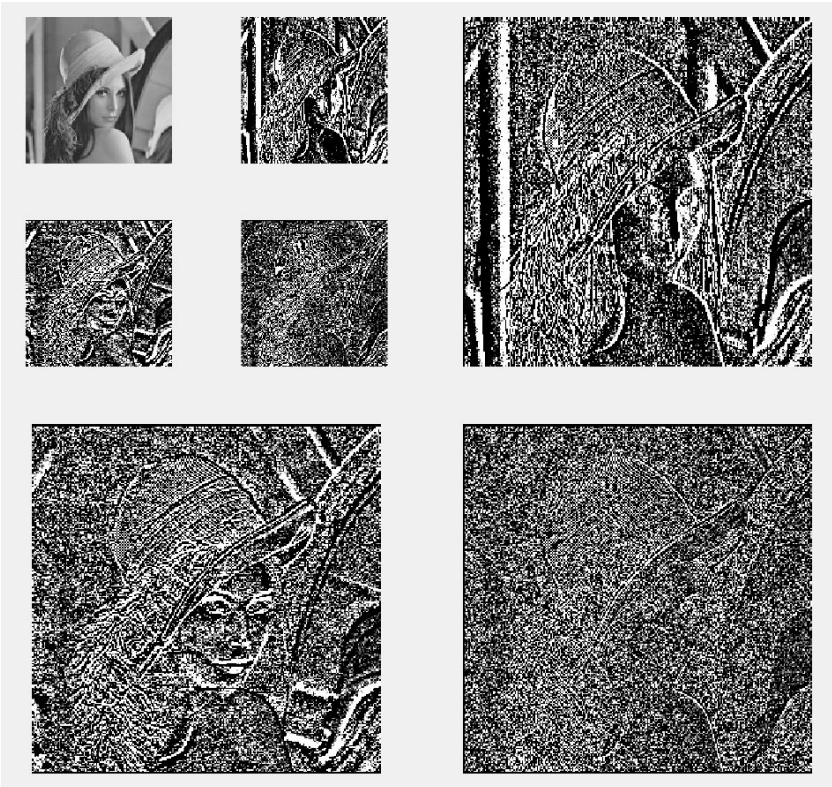

Figure 2. Codec - Lena_gray_512 after using Haar Transform twice

It is important to note that the previous image is not the actual image since all the subbands other than the first one only contain values very close to 0, so the command “IMSHOW” is exaggerating the intensity of the colors to provide a better visualization experience.

The next step is to quantize the image using the Uniform Scalar Quantizer developed for the previous assignment. Each subband must have a different quantization step, but in order to maintain the best quality we are going to explore the histogram of each subband.

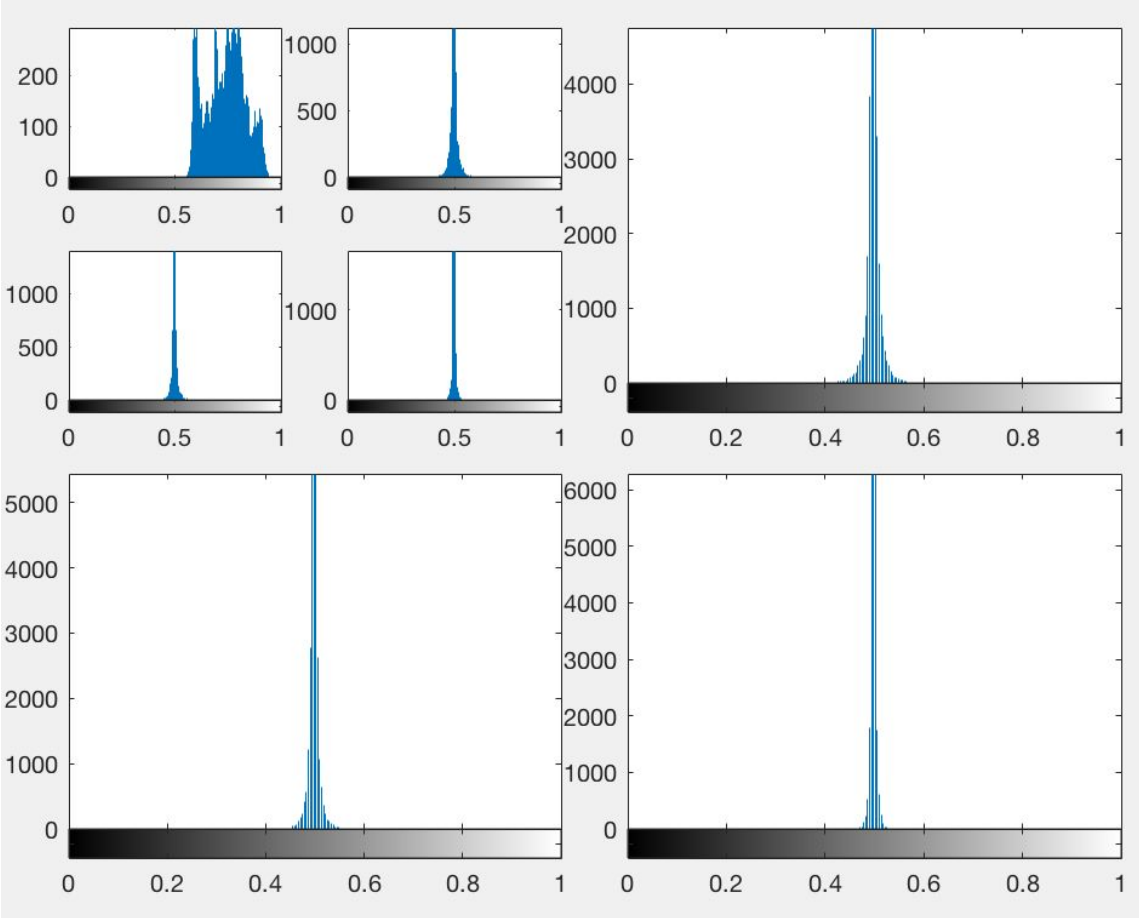

Figure 2. Codec - Histogram of Lena_gray_512 after using Haar Transform twice.

Since the histogram tool didn’t work with negative numbers, we have slided all the values up, so in reality, the 0.5 of the histogram corresponds to a 0 in the image, 0 corresponds to -255 and 1 to 255.
In any case, the general structure is maintained and we can see that the subbands closer to the low frequency subband contain more different values than the outer ones and that the histogram of the low frequency subband is very similar to that of the original image.

We have decided to use 7 bits for the quantization of the HL2 and LH2 subbands (the number 2 denotes that they are the result of the second pass of the Haar Transform) while using 6 for the HH2 subband. For the HL1 and LH1 we will use 5 bits and for the HH1 only 4. In any case, we would get much better results using a non-uniform scalar quantizer that used smaller quantization steps around the origin.

Once the image has been quantized, we can calculate the entropy and the huffman entropy for each subband as well as for the whole image:

Subband	Entropy (bps)	CR(Entrop)	Huffman(bps)	CR (Huffman)
LL2	11.2518	0.7	11.2827	0.7
HL2	2.7155	2.9	2.7401	2.9
LH2	1.934	4.1	2.0665	3.9
HH2	0.705	11.3	1.2525	6.4
HL1	0.6078	13.2	1.1995	6.7
LH1	0.3524	22.7	1.0904	7.3
HH1	0.0049	1641.8	1.0004	8
Whole image	1.7207	4.6	2.1243	3.8

Figure 3. Codec - Table of entropies

Since the bits cannot be divided, all the values have to be rounded up to the closest superior integer, so the difference between the entropy and the Huffman entropy can be higher than it looks. For example the entropy of the image is 1.7207 that transforms into 2 bits, while the Huffman Entropy is 2.1243, that transforms into 3 bits, making Huffman a 50% more expensive in this case.

In any case, we have achieved a 3 bits per pixel compression down from the original 8 bits per pixel and in the case of some subbands we can get a compression rate of 8. It all comes at a cost of using more bps in the LL2 subband than the original image, but it is offset by the compression ratios in the other subbands. One strange outlier is the HH1 subband, which contains the smallest bps of them all with almost 1.0 bps in the case of huffman and 0.0049 bps for the entropy function. This means that we have almost no different symbols on this subband, meaning that the quantizer has erased a lot of information from this subband due to the big size of the quantization step and the small distribution of the values (all very close to 0)

After calculating the entropy, all that is left is to perform the synthesis of the image and compare the results with the original image. In order to do so, first we must dequantize the image and then apply the reverse Haar Transform.

Figure 4. Codec - Synthesis of Lena_gray_512

We can see some pixelation on the contours of the image, for example on the nose or the edge of the hat due to the lost of high frequencies during the quantization step (high frequencies contain the information about the contours of the objects or when there is an abrupt change in color/intensity)
We can compare the original image and the synthesis by using the Peak Signal Noise Ratio, which requires to calculate the MSE(minimum square error) between the original and the synthesis.
The matlab function “measerr” provides multiple error measurements such as PSNR, MSE or MAX_ERR, so we can get the PSNR directly or by using MSE in the formula:

“PSNR = 10 * log10((255^2)/MSE)”

In both cases the result is the same: **25.8014.** It is important to note that the PSNR uses a logarithmic scale, so an increase of 1 in the PSNR means that the signal is 10 times stronger than before.

The code for this part can be found in the section “Annex - Compression”

Motion Estimation

In Video Compression, motion is a key source of information. The motion is produced due to the movement of objects in the 3D scene and it is estimated capturing the resulting spatio-temporal variations of pixel intensities in successive images of a sequence.

In this chapter, it is explained the design and implementation of a motion estimation algorithm with the aim of use this technique to recover the information by analyzing the image content.

Block Matching Algorithm

The block matching algorithm is used in image sequence coding and in most of the video coding standard to date, including H.264.

The algorithm implements a way of locating matching macroblocks in a sequence of digital video frames for the purposes of motion estimation. Then, this estimated motion is used to discover temporal redundancy in the video sequence, increasing the effectiveness of inter-frame video compression by defining the contents of a macroblock by reference to the contents of a known macroblock which is minimally different.

For the implementation of this algorithm, first, the image is subdivided in macroblocks as it is done in the Motion_Estimation function (*figure 5*) and then it is applied the block matching algorithm to compute each motion vector. In any case, for all the pixels in the block, is computed a single motion vector. The block matching algorithm is performed by computing the similarity between the blocks x,y and x-i, x-j. The displacement d(i,j) of each block is performed considering a search parameter p.

Finally, it is computed the motion vector for each macroblock using the minimum cost function as a result of calculate the Mean Square Error between each displacement within the search parameter p.

```
function motion_vectors = Motion_Estimation(reference_frame, new_frame, block_size, p)
%Motion_Estimation algorithm  Computes motion vectors using exhaustive search method
%
% Input
%   reference_frame  : The frame{t-1}
%   new_frame        : The frame{t}
%   p                : Search parameter
%   block_size       : Size of the macroblock
%
% Ouput
%   motion_vectors  : The motion vectors for each macroblock
%

[rows, cols] = size(reference_frame);
motion_vectors = zeros(3,rows*cols/block_size^2);

vect_count = 1;
for posx = 1:block_size:rows-block_size
   for posy = 1:block_size:cols-block_size
        [predicted_posx,predicted_posy,min_MSE]=Block_Matching( reference_frame, new_frame, p,
posx, posy, block_size  );
        motion_vectors(1:2,vect_count) = [predicted_posx,predicted_posy];
        motion_vectors(3,vect_count) = min_MSE;
        vect_count = vect_count+1;
   end
end

end
```

Figure 5. Motion Estimation - Code implementation in Matlab

The block matching implements an exhaustive search and it receives the reference and new frames and the position x,y of the macroblock within the frame with the search parameter p. In the figure 6, it is illustrated an example of the exhaustive search comparing the block of the previous frame and the block of the current frame after applying the pixels displacement in the current image.

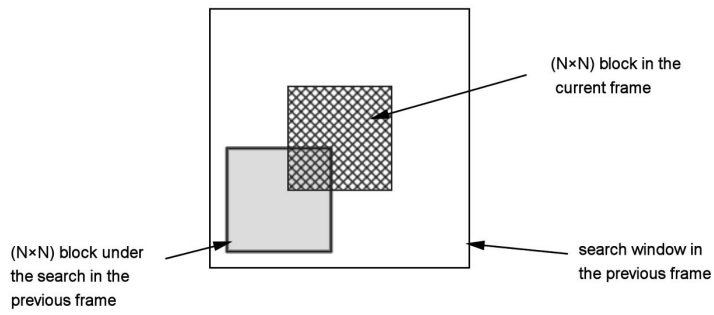

Figure 6. Block Matching algorithm

Once that it is obtained both macroblocks it is computed the mean square error between the 2 macroblocks. This mean square error it is compared with all the rest, after this operation for each displacement defined by p (search parameter). Finally, it is selected the best

matching displacement which gives as result the corresponding the motion vector at that location.

```
function [ x_predicted, y_predicted, min_MSE ] = Block_Matching( reference_frame, new_frame, p, x, y, block_size  )
% Block_Matching Algorithm. Block Matching between 2 subsequent frames to detect motion estimation
%
% Input
%   reference_frame :  The frame{t-1}
%   new_frame       :  The frame{t}
%   p               :  Search parameter
%   x,y             :  Position withing the frame
%   block_size      :  Size of the macroblock
%
% Ouput
%   motion_vect     :  Motion vector for each macroblock
%   min_MSE         :  The minimum MSE (Mean Square Error)
%

reference_block = reference_frame(x:x+(block_size-1),y:y+(block_size-1));
MSE = ones(2*p + 1, 2*p +1) * 7000;

% Search the correspondence of the macroblock of a reference frame
% using an exhaustive method. This method searches for the minimum
% cost function at each possible location with respect to the search parameter.

for m=-p:p
   for n=-p:p
      new_xpos = x + m;
      new_ypos = y + n;
      if(new_xpos+block_size>size(reference_frame,1) || new_ypos+block_size>size(reference_frame,2)
|| new_xpos<1 || new_ypos<1)
         continue;
      else
         new_block
=new_frame(new_xpos:new_xpos+(block_size-1),new_ypos:new_ypos+(block_size-1));
         MSE(m+p+1,n+p+1)=double(mean2(((reference_block-new_block).^2)));
      end
   end
end

% Create the motion vectors with the minimum cost function.
min = 70000;
for i = 1:size(MSE,1)
   for j = 1:size(MSE,2)
      if (MSE(i,j) < min)
         min = MSE(i,j);dx = j; dy = i;
      end
   end
end

min_MSE = min;
x_predicted = dx-p-1;
y_predicted = dy-p-1;

end
```

Figure 7. Block Matching - Code implementation in Matlab

Motion Compensation

Motion compensation is used to estimate the next video frame using the reference frame and the motion vectors computed with motion estimation techniques, as the motion technique described previously.

For this purpose, it is applied the displacement between blocks in the reference frame using the motion vector of each macroblock.

In the figure 8 it is implemented the motion compensation algorithm using the motion vectors computed previously. In this algorithm it is looped all the subsequents macroblocks and it is interchanged the current macroblock for the macroblock displaced with its corresponding the motion vector.

```
function  estimated_image = Motion_Compensation( reference_frame, motion_vect, block_size)
%Motion_Compensation algorithm. Predicts the next image using the motion
%vectors
%
%  Input
%   reference_frame  : The frame{t-1}
%   motion_vect      : The motion vectors for each macroblock
%   block_size       : Size of the macroblock
%
%  Output
%   estimated_image  : Result of using the motion vectors to predict a frame in a video stream
%

[row, col] = size(reference_frame);
vect_count = 1;
image_compensation = reference_frame;

for i = 1:block_size:row-block_size
   for j = 1:block_size:col-block_size
      dy = motion_vect(1,vect_count);
      dx = motion_vect(2,vect_count);
      y_block = i + dy;
      x_block = j + dx;
      if(y_block>=1&&x_block>=1&&dy~=0&&dx~=0)
         image_compensation(i:i+block_size-1,j:j+block_size-1) =
reference_frame(y_block:y_block+block_size-1, x_block:x_block+block_size-1);
      end
      vect_count = vect_count + 1;
   end
end

estimated_image = image_compensation;
```

```
end
```

Figure 8. Motion Compensation - Code implementation in Matlab

In the figures 9, 10 it is showed the result after applying the motion compensation between the frames (40,42) and (1,2). As it can be seen in *figure 9*, if the contained motion between frames is very big there is a huge loss of information and If there is no almost motion between the image remains the same as it can be seen in *figure 10*.

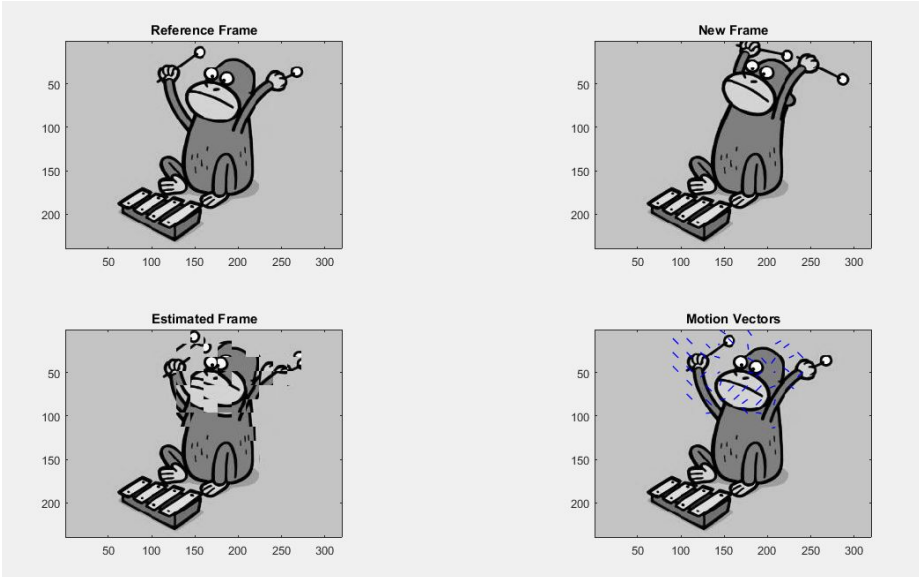

Figure 9 - Images 40,42; reference frame, new frame, estimated frame and motion vectors

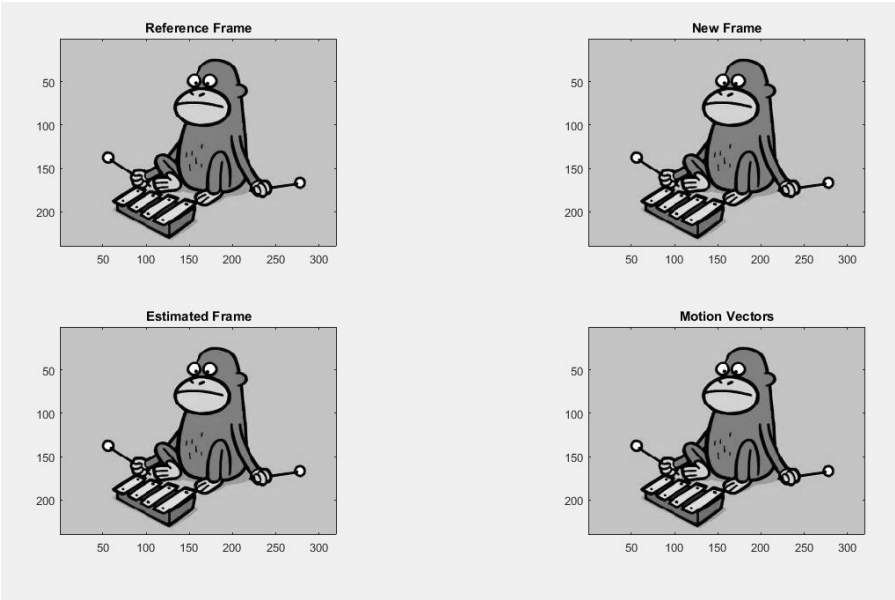

Figure 10 - Output images of the reference frame, new frame, estimated frame and motion vectors

Distortion measurement

The Peak Signal-to-Noise Ration (PSNR) is a term used for the ratio between the maximum possible power of a signal and the power of corrupting noise that affects the fidelity of its representation.

In this chapter it is measured the distortion between the estimated frame and the predicted frame with the PSNR. The *Figure 11* shows the implementation of PSNR and computed as with the same formula as it is defined in the Codec chapter.

```
function PSNR = image_PSNR(new_image, estimated_image)

[row, col] = size(new_image);
sum_err = 0;
for i = 1:row
   for j = 1:col
      sum_err = sum_err + double(mean2(abs(double(new_image(i,j)) -
double(estimated_image(i,j))).^2));
   end
end

PSNR = 10*log10(double(255^2/sum_err));
```

Figure 11 - Peak Signal-to-Noise Ratio function code

In the following table (*Figure 12*), it is shown the different PSNR and MSE between subsequent frames and frames taken randomly.

For measure the distortion between frames it is interesting also use different parameters of the block matching algorithm. For this purpose, the figure 12 also compares the PSNR and MSR using different values of p (search parameter) and block size.

As it is expected, the MSE decrease if it is incremented the search parameter or decremented the block size. Nevertheless, as we increase the search parameter p or decrease the block size the computational cost of the function increases exponentially. For this reason, there are other algorithms that make use of different structures as trees or others to represent and compute the matching between frames.

Frame number	***PSNR***	***MSE***	***p***	***block_size***
40-42	**13.7382**	**2.7496e+03**	**6**	**17**
40-42	**14.8220**	**2.1423e+03**	**20**	**17**
40-42	**14.8958**	**2.1062e+03**	**6**	**2**
1-2	58.6805	0.0881	6	17

1-20	26.7581	137.1739	6	17

Figure 12 - Table with frame numbers and PSNR

Stereoscopy

Stereoscopy is a technique used to record and display 3D (three dimensional) images to provide the illusion of depth in an image. For this exercise we will use a color image provided specially for this exercise as it is shown in *Figure 13*.

Figure 13. Stereoscopy - Original image

We need to split the image into 2, resulting in 2 images of 550x551, each of them with 3 layers (red, green and blue). Then we can generate 2 images with opposite colors (red and cyan)

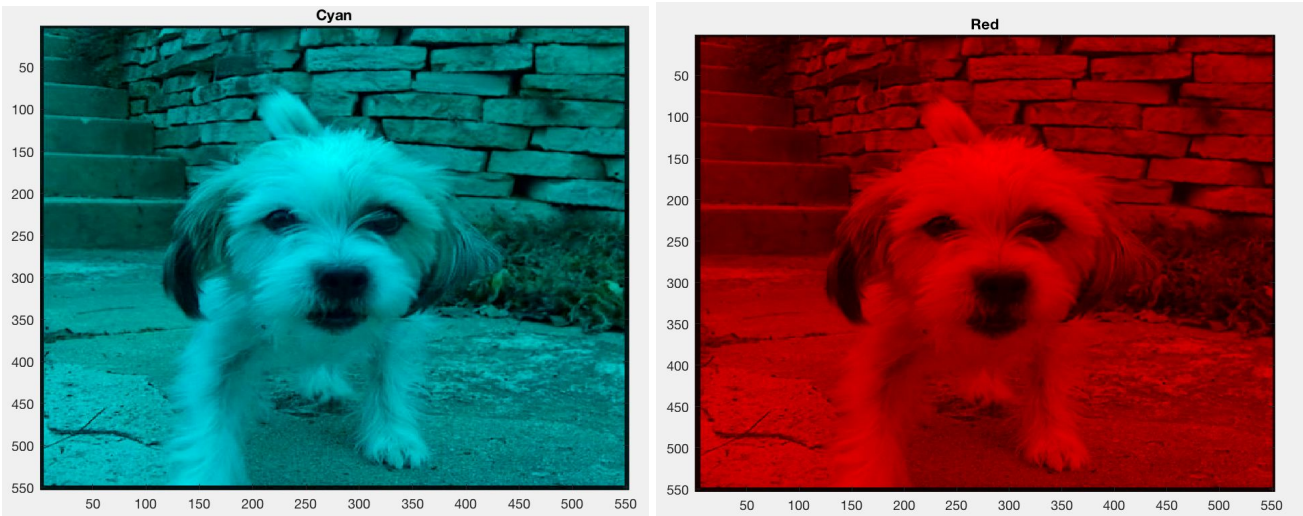

Figure 14. Stereoscopy - Image decomposition into R and GB

After than we can proceed to apply the Haar transform with 2 decomposition layers and the quantizer with R=4 for the 1st layer and R=3 for the 2nd.

One problem we encounter with the transformation step is that the size of the images is not divisible by for, which is a prerequisite for using the 2 layer haar transform. In order to solve this problem, we have used padding with 0's to obtain an acceptable size (552x552)

After the padding we just need to apply the Haar transform with 2 decomposition layers and then apply the quantizer.

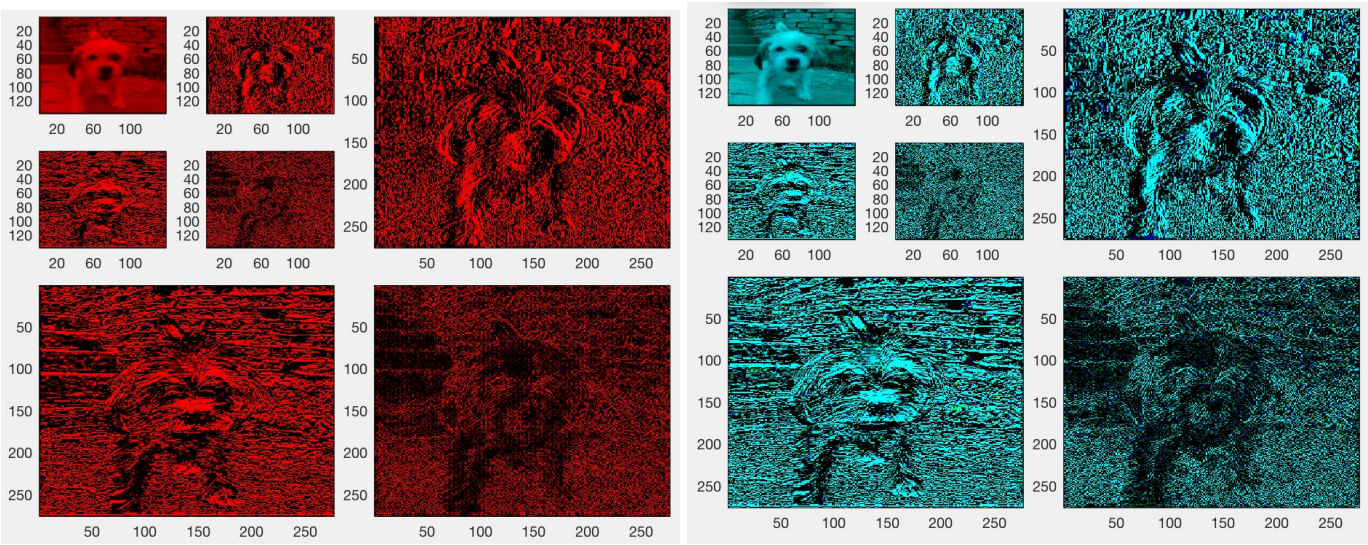

Figure 15. Stereoscopy - Results of applying the Haar transform to R and GB

After this is done, we can synthesize both the images and produce the anaglyph by overlaying the red and cyan layers.

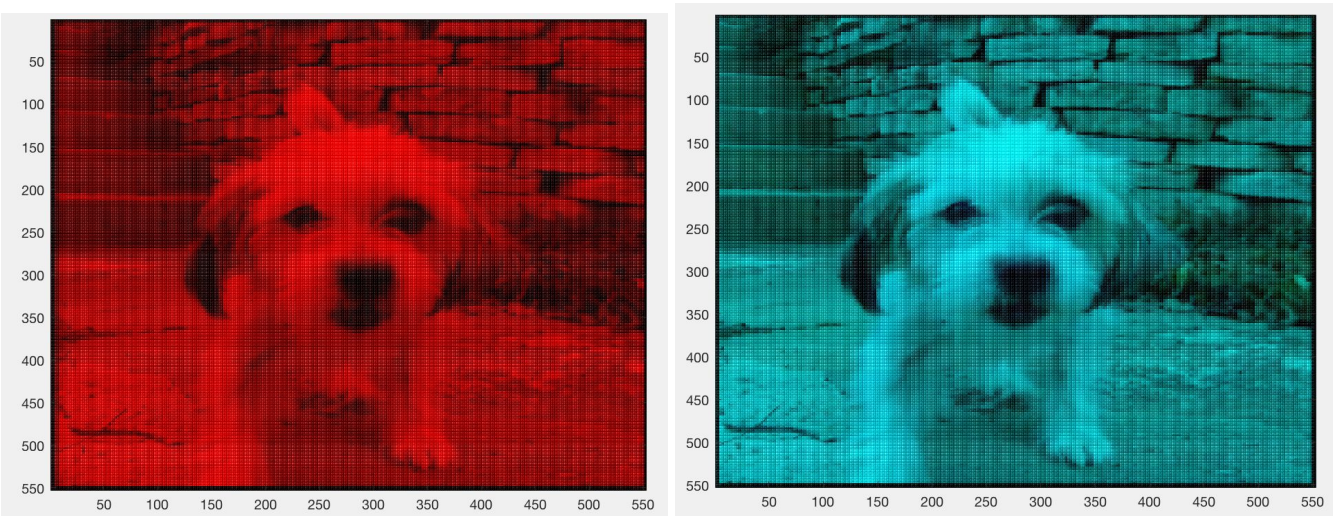

Figure 16. Stereoscopy - Synthesis of R and GB

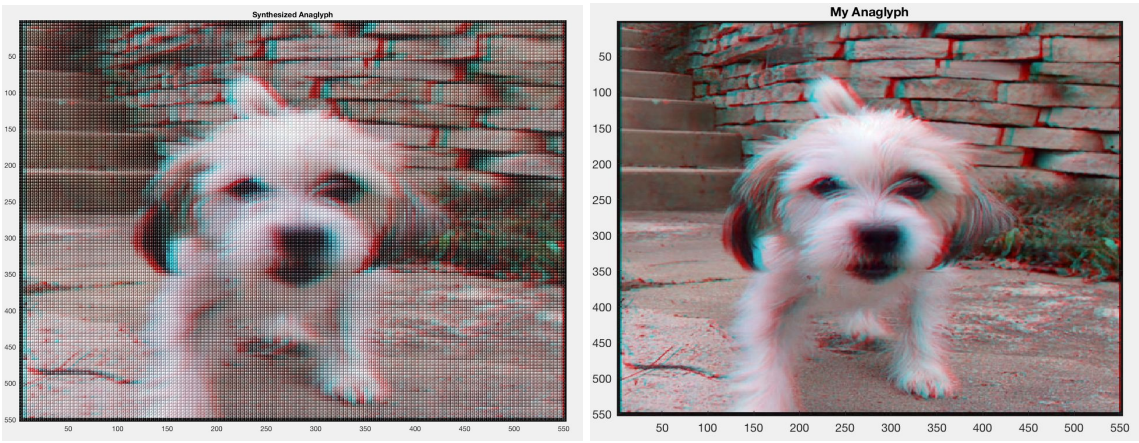

Figure 17. Stereoscopy - Synthesized anaglyph vs original anaglyph.

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 .

Codec

Most of the code from the codec is already in the first assignment, so all of the new code is in this annex since it is used mostly to get images and values (such as PSNR or entropies) The code itself is very long due to generating different variables for each subband in order to transform and display them.

```
% Part 3. Implementation of a codec
% 1. Load image/lena_gray_512.tif
% Use the double haar transform.
lena_haar = haar_transform_multilevel(lena_gray_512,2);
% Let's see the histogram for each sub-band
% First we divide the matrix into submatrices
% We have 3x3 sub-bands
sub11 = lena_haar(1:128,1:128);
sub12 = lena_haar(1:128,129:256);
sub13 = lena_haar(129:256,1:128);
sub14 = lena_haar(129:256,129:256);
sub21 = lena_haar(1:256,257:512);
sub22 = lena_haar(257:512,1:256);
sub23 = lena_haar(257:512,257:512);
figure
subplot(4,4,1)
imshow(sub11,[0,255])
subplot(4,4,2)
imshow(sub12)
```

```
subplot(4,4,5)
imshow(sub13)
subplot(4,4,6)
imshow(sub14)
% Now for the bigger subbands we are going to use bigger subplots
subplot(4,4,[3:4 7:8])
imshow(sub21)
subplot(4,4,[9:10 13:14])
imshow(sub22)
subplot(4,4,[11:12 15:16])
imshow(sub23)

% We are going to use the same subplot distribution to show the histogram
% of each sub image
% Since imhist only uses values from 0 to 1,
% we need to move everything 256 values up (to make the negative numbers)
% possitive and then divide by 512 to obtain values from 0 to 1.
% In this scale, the old 0 is at 0.5, -255 at 0 and 255 at 1.
figure
subplot(4,4,1)
imhist((sub11+255)./512)
subplot(4,4,2)
imhist((sub12+255)./512)
subplot(4,4,5)
imhist((sub13+255)./512)
subplot(4,4,6)
imhist((sub14+255)./512)
% Now for the bigger subbands we are going to use bigger subplots
subplot(4,4,[3:4 7:8])
imhist((sub21+255)./512)
subplot(4,4,[9:10 13:14])
imhist((sub22+255)./512)
subplot(4,4,[11:12 15:16])
imhist((sub23+255)./512)

% After looking at the distribution of data, we should use a non-uniform
% scalar quantizer and with a smaller quantization step around 0, but since
% we only have a uniform scalar quantizer, we will use that.
% No Quantization for the ll part of the image (sub 11)

% Need to count all the appearances of each number
entropies = zeros(7,8);
q_sub11 = sub11;
q_sub12 = quantize_matrix(sub12,7);
q_sub13 = quantize_matrix(sub13,7);
q_sub14 = quantize_matrix(sub14,6);
q_sub21 = quantize_matrix(sub21,5);
q_sub22 = quantize_matrix(sub22,5);
q_sub23 = quantize_matrix(sub23,4);
```

```
e_sub11 = shannonEntropy(q_sub11);
h_sub11 = numel(huffmanCode(q_sub11))/numel(q_sub11);

e_sub12 = shannonEntropy(q_sub12);
h_sub12 = numel(huffmanCode(q_sub12))/numel(q_sub12);

e_sub13 = shannonEntropy(q_sub13);
h_sub13 = numel(huffmanCode(q_sub13))/numel(q_sub13);

e_sub14 = shannonEntropy(q_sub14);
h_sub14 = numel(huffmanCode(q_sub14))/numel(q_sub14);

e_sub21 = shannonEntropy(q_sub21);
h_sub21 = numel(huffmanCode(q_sub21))/numel(q_sub21);

e_sub22 = shannonEntropy(q_sub22);
h_sub22 = numel(huffmanCode(q_sub22))/numel(q_sub22);

e_sub23 = shannonEntropy(q_sub23);
h_sub23 = numel(huffmanCode(q_sub23))/numel(q_sub23);

% We need to "stitch" all the pieces of q_lena together
q_lena = zeros(size(lena_gray_512,1),size(lena_gray_512,2));
q_lena(1:128,1:128) = q_sub11;
q_lena(1:128,129:256) = q_sub12;
q_lena(129:256,1:128) = q_sub13;
q_lena(129:256,129:256) = q_sub14;
q_lena(1:256,257:512) = q_sub21;
q_lena(257:512,1:256) = q_sub22;
q_lena(257:512,257:512) = q_sub23;
e_lena = shannonEntropy(q_lena);
h_lena = numel(huffmanCode(q_lena))/numel(q_lena); %q_lena in huffman code

entropies = [e_sub11, e_sub12, e_sub13, e_sub14, e_sub21, e_sub22, e_sub23, e_lena;
        h_sub11, h_sub12, h_sub13, h_sub14, h_sub21, h_sub22, h_sub23, h_lena]';

entropies
compression_rates = 8./entropies; %original image has 8 bpp
compression_rates

%%%%%%%% SYNTHESIS %%%%%%%%%%%%%
dq_lena = zeros(size(lena_gray_512,1),size(lena_gray_512,2));
dq_lena(1:128,1:128) = q_lena(1:128,1:128);
dq_lena(1:128,129:256) = dequantize_matrix(q_lena(1:128,129:256),7,8);
dq_lena(129:256,1:128) = dequantize_matrix(q_lena(129:256,1:128),7,8);
dq_lena(129:256,129:256) = dequantize_matrix(q_lena(129:256,129:256),6,8);
dq_lena(1:256,257:512) = dequantize_matrix(q_lena(1:256,257:512),5,8);
dq_lena(257:512,1:256) = dequantize_matrix(q_lena(257:512,1:256),5,8);
dq_lena(257:512,257:512) = dequantize_matrix(q_lena(257:512,257:512),4,8);
```

```
synth_lena = haar_reverse_multilevel(dq_lena,2);
figure
imshow(synth_lena,[0,255])

%%%%%%%% PSNR calculation %%%%%%%%
[PSNR,MSE,MAXERR,L2RAT] = measerr(lena_gray_512,synth_lena);
% measerr already provides the PSNR but we can calculate it again using the
% formula: 10 * log10((255^2)/MSE)
PSNR_2 = 10 * log10((255^2)/MSE);
% Which gives the exact same result
[PSNR, PSNR_2]

% END OF FILE

% AUXILIARY FUNCTIONS
function code = huffmanCode(matrix)
        a = unique(matrix);
        count = numel(matrix);
        out = [a,histc(matrix(:),a)/count];
        p = out(:,2);
        dict = huffmandict(a,p);
        code = huffmanenco(matrix(:),dict);
end

function bitsPerSymbol = shannonEntropy(matrix)
        a = unique(matrix);
        count = numel(matrix);
        out = [a,histc(matrix(:),a)/count];
        p = out(:,2);
        H = sum(-(p(p>0).*(log2(p(p>0))))); % Shannon entropy formula
        bitsPerSymbol = H;
end
```

Motion Estimation

Auxiliar script file

```
% Read the video file
video=VideoReader('../Test-images/Xylophone.mp4');
n=video.NumberofFrames;

% Divide the video in frames
for x=1:n
frame=read(video,x);
imwrite(frame,sprintf('../Test-images/Frames/image%d.jpg',x));
end

% Compute the motion vectors between 2 frames
reference_frame = imread('../Test-images/Frames/image40.jpg');
```

```
new_frame = imread('../Test-images/Frames/image41.jpg');

% Transform in gray scale for simplification
reference_frame = rgb2gray(reference_frame);
new_frame = rgb2gray(new_frame);

% Plot the reference and new frames
subplot(2,2,1),subimage(reference_frame),title('Reference Frame'); hold on;
subplot(2,2,2),subimage(new_frame),title('New Frame');

% p is the search parameter, block_size is the size of the macroblock
p=7;block_size=16;

% Motion estimation using an exhaustive method
[motion_vect] = Motion_Estimation(reference_frame,new_frame,block_size,p);

% Motion competation using the motion vectors
estimated_frame= Motion_Compensation(reference_frame, motion_vect, block_size);
subplot(2,2,3),subimage(estimated_frame),title('Estimated Frame');
draw_Optical_Flow( reference_frame, motion_vect, block_size );
hold off;

% Peak Signal-to-Noise Ratio (PSNR) to measure the distortion of the frame which is
estimated and the predicted one
PSNR = image_PSNR(reference_frame, estimated_frame);
fprintf('Peak-Signal-To-Noise Ratio: %d \n',PSNR);
```

Draw optical flow function

```
function draw_Optical_Flow( reference_frame, motion_vect, block_size )
%UNTITLED Draw the motion vectors in the reference frame

[row, col] = size(reference_frame);
mbCount = 1;
subplot(2,2,4),subimage(reference_frame),title('Motion Vectors');
hold on;
for i = 1:block_size:row-block_size
   for j = 1:block_size:col-block_size
      dy = motion_vect(1,mbCount);
      dx = motion_vect(2,mbCount);
      y_block = i + dy;
      x_block = j + dx;
      if((y_block>=1&&x_block>=1&&dy~=0&&dx~=0) &&  (motion_vect(3,mbCount)>0.4)
&& (x_block<=size(reference_frame,1)) && (y_block<=size(reference_frame,2)))
         quiver(j,i,dx, dy,'color', 'b', 'linewidth', 1);
      end
      mbCount = mbCount + 1;
   end
end
hold off;
```

```
end
```

Stereoscopy

Again, most of the code has been used in TD3 (Codec) and the code here is simply to display some images (and a very small part of the code to do the padding and split the image into r + gb)

```
[img1, colormap] = imread('../Test-images/stereoscopy-images/image3.jpg');
figure; image(img1), title('Original Image'); % Original image

%  Split them into 2 separate images
leftImage = img1(:,1:(end/2),:);

rightImage = img1(:,end/2+1:end,:);
figure; image(leftImage), title('Left Image');
figure; image(rightImage), title('Right Image');

leftImage = paddingZeros(leftImage,4); %We need a dimension divisible by 4
rightImage = paddingZeros(rightImage,4);


% Encode each eye's image using filters of different (usually chromatically opposite)
colors, red and blue
r = zeros(size(leftImage));
gb = zeros(size(rightImage));
r(:,:,1) = double(leftImage(:,:,1));
gb(:,:,2:3) = double(rightImage(:,:,2:3));
anaglyph = uint8(r+gb);
figure,image(uint8(r)), title('Red')
figure,image(uint8(gb)), title('Cyan')
figure,image(anaglyph), title('My Anaglyph')

% Create stereo anaglyph with Matlab function to check the result
J = stereoAnaglyph(leftImage,rightImage);
figure, image(uint8(J)), title('Anaglyph using Matlab function')

% Use TD3 to code them using 2 decomposition layers
leftImage_haar = zeros(size(r,1),size(r,2),size(r,3));
rightImage_haar =zeros(size(gb,1),size(gb,2),size(gb,3));
rows_l=size(leftImage,1);
rows_r=size(rightImage,1);
cols_l=size(leftImage,2);
cols_r=size(rightImage,2);
sub11_l = zeros(rows_l/4,cols_l/4,3); sub11_r = zeros(rows_r/4,cols_r/4,3);
sub12_l = zeros(rows_l/4,cols_l/4,3); sub12_r = zeros(rows_r/4,cols_r/4,3);
sub13_l = zeros(rows_l/4,cols_l/4,3); sub13_r = zeros(rows_r/4,cols_r/4,3);
```

```
sub14_l = zeros(rows_l/4,cols_l/4,3); sub14_r = zeros(rows_r/4,cols_r/4,3);
sub21_l = zeros(rows_l/2,cols_l/2,3); sub21_r = zeros(rows_r/2,cols_r/2,3);
sub22_l = zeros(rows_l/2,cols_l/2,3); sub22_r = zeros(rows_r/2,cols_r/2,3);
sub23_l = zeros(rows_l/2,cols_l/2,3); sub23_r = zeros(rows_r/2,cols_r/2,3);

for i=1:3
  % temp = leftImage(:,:,i);
        temp = r(:,:,i);

        leftImage_haar(:,:,i) = haar_transform_multilevel(temp,2);

        sub11_l(:,:,i) = leftImage_haar(1:rows_l/4, 1:cols_l/4,i);
        sub12_l(:,:,i) = leftImage_haar(1:rows_l/4, cols_l/4+1:cols_l/2,i);
        sub13_l(:,:,i) = leftImage_haar(rows_l/4+1:rows_l/2, 1:cols_l/4,i);
        sub14_l(:,:,i) = leftImage_haar(rows_l/4+1:rows_l/2, cols_l/4+1:cols_l/2,i);
        sub21_l(:,:,i) = leftImage_haar(1:rows_l/2, cols_l/2+1:cols_l,i);
        sub22_l(:,:,i) = leftImage_haar(rows_l/2+1:rows_l, 1:cols_l/2,i);
        sub23_l(:,:,i) = leftImage_haar(rows_l/2+1:rows_l, cols_l/2+1:cols_l,i);

        temp2 = gb(:,:,i);
        rightImage_haar(:,:,i) = haar_transform_multilevel(temp2,2);
        sub11_r(:,:,i) = rightImage_haar(1:rows_l/4, 1:cols_l/4,i);
        sub12_r(:,:,i) = rightImage_haar(1:rows_l/4, cols_l/4+1:cols_l/2,i);
        sub13_r(:,:,i) = rightImage_haar(rows_l/4+1:rows_l/2, 1:cols_l/4,i);
        sub14_r(:,:,i) = rightImage_haar(rows_l/4+1:rows_l/2, cols_l/4+1:cols_l/2,i);
        sub21_r(:,:,i) = rightImage_haar(1:rows_l/2, cols_l/2+1:cols_l,i);
        sub22_r(:,:,i) = rightImage_haar(rows_l/2+1:rows_l, 1:cols_l/2,i);
        sub23_r(:,:,i) = rightImage_haar(rows_l/2+1:rows_l, cols_l/2+1:cols_l,i);
end
figure;image(uint8(leftImage_haar));title('Left Image after haar filter');
figure;image(uint8(rightImage_haar));title('Right Image after haar filter');

% With more detail
figure
subplot(4,4,1)
image(sub11_l./255)
subplot(4,4,2)
image(sub12_l)
subplot(4,4,5)
image(sub13_l)
subplot(4,4,6)
image(sub14_l)
% Now for the bigger subbands we are going to use bigger subplots
subplot(4,4,[3:4 7:8])
image(sub21_l)
subplot(4,4,[9:10 13:14])
image(sub22_l)
subplot(4,4,[11:12 15:16])
image(sub23_l)
```

```
% With more detail
figure
subplot(4,4,1)
image(sub11_r./255)
subplot(4,4,2)
image(sub12_r)
subplot(4,4,5)
image(sub13_r)
subplot(4,4,6)
image(sub14_r)
% Now for the bigger subbands we are going to use bigger subplots
subplot(4,4,[3:4 7:8])
image(sub21_r)
subplot(4,4,[9:10 13:14])
image(sub22_r)
subplot(4,4,[11:12 15:16])
image(sub23_r)

leftImage_dq = zeros(size(leftImage,1),size(leftImage,2),size(leftImage,3));
leftImage_synth = zeros(size(leftImage,1),size(leftImage,2),size(leftImage,3));

rightImage_dq =zeros(size(rightImage,1),size(rightImage,2),size(rightImage,3));
rightImage_synth = zeros(size(rightImage,1),size(rightImage,2),size(rightImage,3));

for i=1:3
        q_sub11_l(:,:,i) = sub11_l(:,:,i);
        q_sub12_l(:,:,i) = quantize_matrix(sub12_l(:,:,i),4);
        q_sub13_l(:,:,i) = quantize_matrix(sub13_l(:,:,i),4);
        q_sub14_l(:,:,i) = quantize_matrix(sub14_l(:,:,i),4);
        q_sub21_l(:,:,i) = quantize_matrix(sub21_l(:,:,i),3);
        q_sub22_l(:,:,i) = quantize_matrix(sub22_l(:,:,i),3);
        q_sub23_l(:,:,i) = quantize_matrix(sub23_l(:,:,i),3);

        q_sub11_r(:,:,i) = sub11_r(:,:,i);
        q_sub12_r(:,:,i) = quantize_matrix(sub12_r(:,:,i),4);
        q_sub13_r(:,:,i) = quantize_matrix(sub13_r(:,:,i),4);
        q_sub14_r(:,:,i) = quantize_matrix(sub14_r(:,:,i),4);
        q_sub21_r(:,:,i) = quantize_matrix(sub21_r(:,:,i),3);
        q_sub22_r(:,:,i) = quantize_matrix(sub22_r(:,:,i),3);
        q_sub23_r(:,:,i) = quantize_matrix(sub23_r(:,:,i),3);

        %Synthesis
        leftImage_dq(1:rows_l/4, 1:cols_l/4,i)= q_sub11_l(:,:,i);
        leftImage_dq(1:rows_l/4, cols_l/4+1:cols_l/2,i) =
dequantize_matrix(q_sub12_l(:,:,i),4,8);
        leftImage_dq(rows_l/4+1:rows_l/2, 1:cols_l/4,i) =
dequantize_matrix(q_sub13_l(:,:,i),4,8);
        leftImage_dq(rows_l/4+1:rows_l/2, cols_l/4+1:cols_l/2,i) =
dequantize_matrix(q_sub14_l(:,:,i),4,8);
        leftImage_dq(1:rows_l/2, cols_l/2+1:cols_l,i) =
```

```
dequantize_matrix(q_sub21_l(:,:,i),3,8);
        leftImage_dq(rows_l/2+1:rows_l, 1:cols_l/2,i) =
dequantize_matrix(q_sub22_l(:,:,i),3,8);
        leftImage_dq(rows_l/2+1:rows_l, cols_l/2+1:cols_l,i) =
dequantize_matrix(q_sub23_l(:,:,i),4,8);
        leftImage_synth(:,:,i) = haar_reverse_multilevel(leftImage_dq(:,:,i),2);

        rightImage_dq(1:rows_r/4, 1:cols_r/4,i)= q_sub11_r(:,:,i);
        rightImage_dq(1:rows_r/4, cols_r/4+1:cols_r/2,i) =
dequantize_matrix(q_sub12_r(:,:,i),4,8);
        rightImage_dq(rows_r/4+1:rows_r/2, 1:cols_r/4,i) =
dequantize_matrix(q_sub13_r(:,:,i),4,8);
        rightImage_dq(rows_r/4+1:rows_r/2, cols_r/4+1:cols_r/2,i) =
dequantize_matrix(q_sub14_r(:,:,i),4,8);
        rightImage_dq(1:rows_r/2, cols_r/2+1:cols_r,i) =
dequantize_matrix(q_sub21_r(:,:,i),3,8);
        rightImage_dq(rows_r/2+1:rows_r, 1:cols_r/2,i) =
dequantize_matrix(q_sub22_r(:,:,i),3,8);
        rightImage_dq(rows_r/2+1:rows_r, cols_r/2+1:cols_r,i) =
dequantize_matrix(q_sub23_r(:,:,i),4,8);
        rightImage_synth(:,:,i) = haar_reverse_multilevel(rightImage_dq(:,:,i),2);
end
figure; image(uint8(rightImage_synth));
figure; image(uint8(leftImage_synth));

%Anaglyph of the synthesis
anaglyph_synth = uint8(leftImage_synth+rightImage_synth);
figure,image(anaglyph_synth), title('Synthesized Anaglyph')

% Encode each eye's image using filters of different (usually chromatically opposite)
colors, red and blue
rightImage_synth  = imtranslate(rightImage_synth,[10, 0]);

subplot(3,3,3),subimage(rightImage_synth), title('Anaglyph Synth');
r = zeros(size(rightImage_synth));
gb = zeros(size(leftImage_synth));
r(:,:,1) = double(rightImage_synth(:,:,1));
gb(:,:,2:3) = double(leftImage_synth(:,:,2:3));
anaglyph = uint8(r+gb);
figure,image(anaglyph), title('Anaglyph Synth');

% Create stereo anaglyph with Matlab function to check the result
J = stereoAnaglyph(rightImage_synth,leftImage_synth);
figure, image(uint8(J)), title('Anaglyph using Matlab function')
```

Auxiliar function paddedMatrix

```
function paddedMatrix = paddingZeros(matrix, divisor)
        %paddedMatrix = zeros(size(matrix,1), size(matrix,2), size(matrix,3));
        rows = size(matrix,1);
        cols = size(matrix,2);
        paddedMatrix = wextend('ar','sym',matrix, round(rows/divisor)*divisor - rows, 'd');
        paddedMatrix = wextend('ac','sym',paddedMatrix, round(cols/divisor)*divisor - cols,
'l');
end
```

Source code

All the code corresponding to this assignment can be downloaded and tested from the following GitHub repository: https://github.com/yolanda93/compression_algorithms