INFORMATION ENGINEERING

PROBLEM 2: COMPRESSION

GROUP V

SARAWAGI Saumya, SARKAR Shourya, SCARANO Miriam, SHARMA Akash,
SHERIFF Sameer, SONG Seonghyun, WANG Zican

PROBLEM STATEMENT

The objectives of this exercise can be divided into three broad headers:

- Assess the performance of the **Huffman Coding**, a lossless compression scheme, to better understand how it works
- Determine the best **Quantization Factor** which allows enhanced visibility of the ISAE logo
- Compute the **Compression Ratio** of the JPEG-L scheme, to quantify its efficacy.

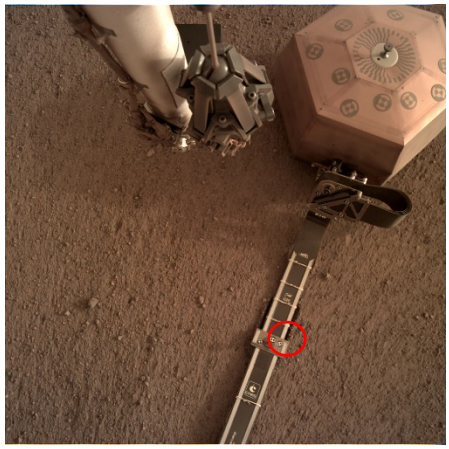

Figure 1: Original Image with ISAE Logo (encircled)

PROJECT METHODOLOGY

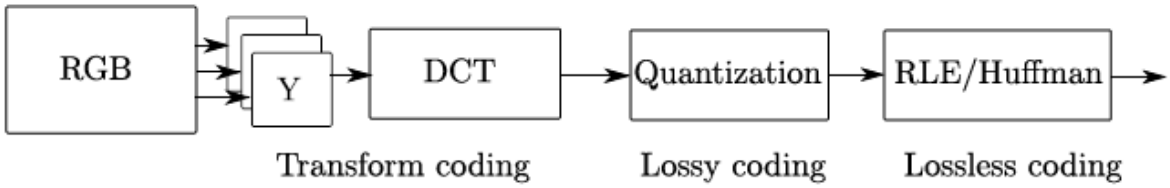

Figure 2: Flowchart of the JPEG-L Scheme

The JPEG-L Scheme, represented diagrammatically above in Figure 2, consists of three operations. At the onset, the image (encoded in Red-Green-Blue data) is converted to YCbCr, followed by a Discrete Cosine Transform (DCT), a 2-dimensional discrete spatial Fourier Transform – this operation is known as **Transform Coding** – as a result of which we obtain a matrix, where the components at high (AC) and low (DC) frequencies are separated. The DC component is the first element of each 8x8 matrix block, with a significantly higher magnitude, while the remaining elements are the AC components; this separation is imperative as they undergo different compression processing.

Next, the data undergoes a **Lossy Coding** operation. The DCT Matrix is quantized, i.e., its values are converted to integers; this can be thought of as the first compression. Since the conversion of decimal values to integers leads to data loss, this process causes loss of information. After quantization, the DC components undergo two processes – *Differential Pulse Modulation Coding (DPMC)*, which computes the DC vector whose components are the differences between two consecutive values, and *Category Amplitude Mapping (CAM)*, which

computes the category and the number of bits for each value. Meanwhile, the AC components undergo three processes – *Zigzag Reading*, to benefit from consecutive equal values in order to contain a larger number of zeroes, *Run Length Encoding (RLE)*, to optimize the reading of the AC vector according to the number of consecutive zeroes separating each element, and *Category Amplitude Mapping*, similar to the DC components.

```
for i_plane in range(0,3):

    # Select a plane  Y, Cb, or Cr
    image_plane = image_trunc[:,:,i_plane]

    for i_row in range(0,n_row,8):
        for j_col in range(0,n_col,8):

            #  Compute the block number
            block_number =( np.round(i_row/8*(n_row/8)+ j_col/8 )).astype(np.int32)
            # Select of block of 8 by 8 pixels
            image_plane_block = image_plane[i_row:i_row+8, j_col:j_col+8]
            # DCT transform
            image_dct = fc.DCT(image_plane_block)
            # Quantization
            image_dct_quant = np.round(image_dct/Q_matrix ).astype(np.int32)
            # Zigzag reading
            image_dct_quant_zgzg =  fc.zigzag(image_dct_quant)
            # Separate DC and AC components
            image_DC[block_number,i_plane] = image_dct_quant_zgzg[0]
            image_AC[block_number*63 + 1:(block_number+1)*63,i_plane] = image_dct_quant_zgzg[1:63]
```

Figure 3: Transform Coding

Finally, the data is further compressed using the Huffman Algorithm and/or Run Length Encoding, depending on its nature, AC or DC. This operation is known as **Lossless Coding**.

IMPLEMENTATION OF THE HUFFMAN CODE

The Huffman algorithm is coded differently, depending on the nature of the components – AC or DC. The following flowcharts illustrate the procedure that has been adopted.

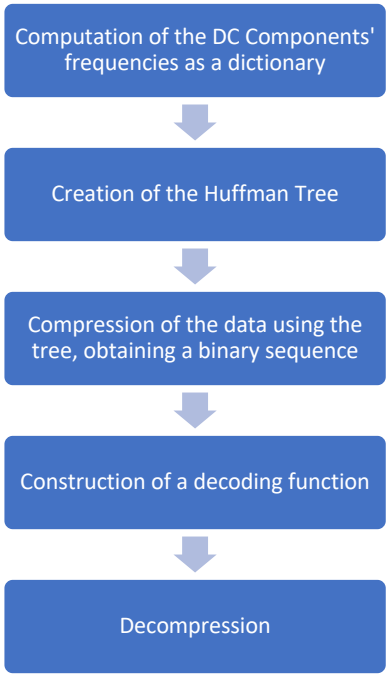

Figure 4: Huffman Methodology for DC Components

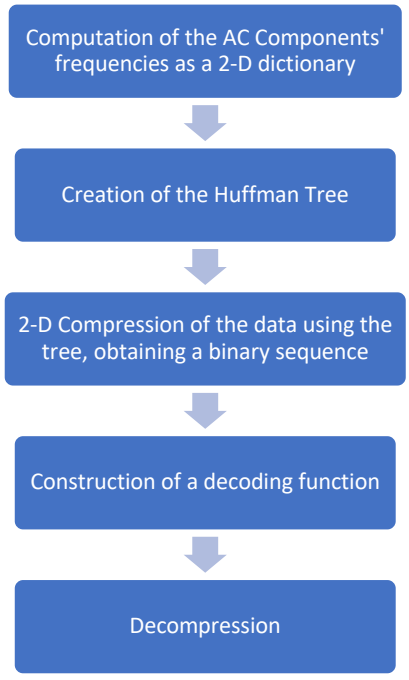

Figure 5:Huffman Methodology for AC Components

Using the functions present in Huffman_functions.py provided to us, the DC and AC components were compressed following the order described in Figure 4 and Figure 5 above.

```
# ------------------------Students work on DC components ------------------------------------
# Compress with Huffman

alphabet_DC = list()
for i in range(17):
    alphabet_DC.append(i)

pixels_probability_DC = hj.dict_freq_numbers(list_image_cat_DC, alphabet_DC)
pixels_probability_DC_dict = pixels_probability_DC[0]
huffman_tree_DC = hj.build_huffman_tree(pixels_probability_DC_dict)
huffman_code_generate_DC = hj.generate_code(huffman_tree_DC)
huffman_code_compression_DC = hj.compress(list_image_cat_DC, huffman_code_generate_DC) ###

# Decompress with Huffman
huffman_decoding_dict_DC = hj.build_decoding_dict(huffman_code_generate_DC)

# decompressed_cat_DC should be the output of your Huffman decompressor
decompressed_cat_DC = hj.decompress(huffman_code_compression_DC, huffman_decoding_dict_DC)
#decompressed_cat_DC = list_image_cat_DC
```

Figure 6: Huffman compression – DC Component

```
# ------------------------Students work on AC components ------------------------------------
# Compress with Huffman

alphabet_AC = list()

for i in list_image_rl_AC:
    for j in i:
        if j in alphabet_AC:
            continue
        alphabet_AC.append(j)

pixels_probability_AC = hj.dict_freq_numbers_2(list_image_rl_AC, alphabet_AC)
pixels_probability_AC_dict = pixels_probability_AC[0]
huffman_tree_AC = hj.build_huffman_tree(pixels_probability_AC_dict)
huffman_code_generate_AC = hj.generate_code_2(huffman_tree_AC)
huffman_code_compression_AC = hj.compress_2(list_image_rl_AC, huffman_code_generate_AC)


# Decompress with Huffman
huffman_decoding_dict_AC = hj.build_decoding_dict(huffman_code_generate_AC)

# Rdecompressed_cat_AC should be the output of your Huffman decompressor
decompressed_cat_AC = hj.decompress(huffman_code_compression_AC, huffman_decoding_dict_AC)
```

Figure 7: Huffman compression – AC Component

CHOOSING THE OPTIMAL QUANTIZATION FACTOR

The Quantization Factor is responsible for the values in the Compression Matrix (Q), which allows us to compress the DCT Matrix (T) as given by the following formula:

$$Q_{uv} = \text{Round}\left(\frac{T}{Q}\right)$$

In order to improve the clarity of the ISAE logo the quantization factor was selected by trial and error. The variation was started from the given value of 50, where the ISAE logo was not clear. Upon *decreasing* the value, the quality of the picture degrades; upon *increasing* the value,

the quality begins to improve until the value of 100, beyond which various artifacts/corrupted pixels appear on the image reducing the quality.

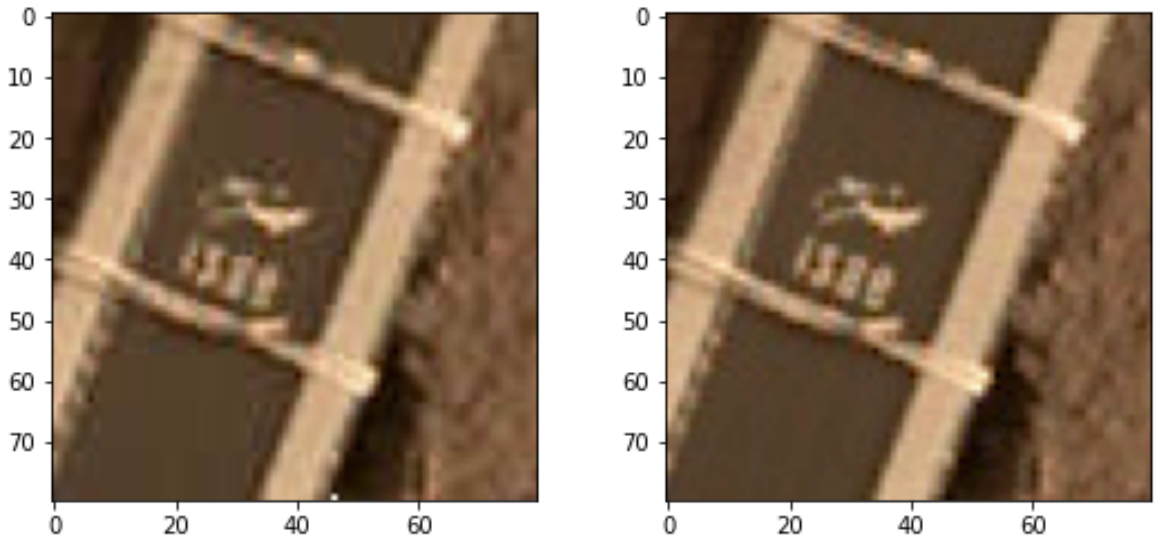

Figure 8: Variation in Quantization Factor
Q=50 (left) Q=97 (right)

Nevertheless, between 90 and 100, no visible differences are observed in the image. Hence, a quantization factor of 97 was determined to be optimal.

CALCULATION OF THE COMPRESSION RATIO

The original image after truncation is reduced to 80 x 80 x 3 pixels (RGB). Each pixel is an 8-bit symbol which gives the total number of bits.

The compression ratio is calculated by total number of bits in uncompressed image divided by the total number of bits in the compressed image.

$$CR = \frac{Total\ bits\ in\ original\ image}{Total\ bits\ in\ compressed\ image}$$

```
total_bits += len(huffman_code_compression_DC)+len(huffman_code_compression_AC)
compression_ratio = (80*80*8*3)/total_bits
```

Figure 9: Compression ratio code

This gives a compression ratio of 5.226

SUMMARY

The process of Lossy compression and Lossless compression (Huffman compression) was implemented in this paper provided a satisfactory compression ratio (CR) of 5.226 at a quantization factor (Q) of 97. The value of Q was decided based on the clarity and visibility of the output image. The results of this paper justify the use of the JPEG-Lossy compression algorithm and the Huffman lossless compression algorithm.

APPENDIX: COMPLETE PYTHON CODE

```
from matplotlib import pyplot
import numpy as np
from PIL import Image as im
import jpeg_functions as fc
import huffman_functions as hj

# Quantization matrix
Q =97; # quality  factor of JPEG in %

Q_matrix = fc.quantization_matrix(Q) # Quantization matrix

# Open the image (obtain an RGB image)
image_origin = im.open("ISAE_Logo_SEIS_clr.PNG")
image_origin_eval = np.array(image_origin)

# Transform into YCrBR
image_ycbcr = image_origin.convert('YCbCr')
image_ycbcr_eval= np.array(image_ycbcr)

#  Troncate the image (to make simulations faster)
image_trunc = image_ycbcr_eval[644:724,624:704]
image_trunc_on_LMS = im.fromarray(image_trunc)
image_trunc_on_LMS.save('Image_on_LMS.png')
#pyplot.imshow(image_trunc)

# Initializations
n_row = np.size(image_trunc,0) # Number of rows
n_col = np.size(image_trunc,1)  # Number of columns
N_blocks =  (np.round(n_row/8 * n_col/8 )).astype(np.int32) # Number of (8x8) blocks
image_plane_rec = np.zeros((n_row,n_col,3),dtype=np.uint8)
image_DC = np.zeros((N_blocks.astype(np.int32),3), dtype = np.int32)
image_DC_rec = np.zeros((N_blocks.astype(np.int32),3), dtype = np.int32)
image_AC = np.zeros((63*N_blocks,3),dtype = np.int32 )
image_AC_rec = np.zeros((63*N_blocks,3),dtype = np.int32 )
bits_per_pixel=np.zeros( 3 )



# ------------------------------------------------------------
#                         Compression
#  ------------------------------------------------------------
total_bits_compressed_image = 0;
total_entropy = 0;

for i_plane in range(0,3):

    # Select a plane  Y, Cb, or Cr
    image_plane = image_trunc[:,:,i_plane]

    for i_row in range(0,n_row,8):
        for j_col in range(0,n_col,8):

            #  Compute the block number
            block_number =( np.round(i_row/8*(n_row/8)+ j_col/8 )).astype(np.int32)
            # Select of block of 8 by 8 pixels
            image_plane_block = image_plane[i_row:i_row+8, j_col:j_col+8]
            # DCT transform
            image_dct = fc.DCT(image_plane_block)
            # Quantization
            image_dct_quant = np.round(image_dct/Q_matrix ).astype(np.int32)
            # Zigzag reading
            image_dct_quant_zgzg =  fc.zigzag(image_dct_quant)
            # Separate DC and AC components
            image_DC[block_number,i_plane] = image_dct_quant_zgzg[0]
            image_AC[block_number*63 + 1:(block_number+1)*63,i_plane] = image_dct_quant_zgzg[1:63]

    # ---- DC components processing
    # DPCM coding over DC components
    image_DC_DPCM = image_DC[1:N_blocks,i_plane] - image_DC[0:N_blocks-1,i_plane]
    image_DC_0 = image_DC[0,i_plane] # first DC constant (not compressed)
```

```python
#  Map the resulting values in categoeries and amplitudes
image_DC_DPCM_cat = fc.DC_category_vect(image_DC_DPCM)
image_DC_DPCM_amp = fc.DC_amplitude_vect(image_DC_DPCM)
list_image_cat_DC = np.ndarray.tolist(image_DC_DPCM_cat)  # create a list of DC components


# --------------------------Students work on DC components ----------------------------------
# Compress with Huffman

alphabet_DC = list()
for i in range(17):
    alphabet_DC.append(i)

pixels_probability_DC = hj.dict_freq_numbers(list_image_cat_DC, alphabet_DC)
pixels_probability_DC_dict = pixels_probability_DC[0]
huffman_tree_DC = hj.build_huffman_tree(pixels_probability_DC_dict)
huffman_code_generate_DC = hj.generate_code(huffman_tree_DC)
huffman_code_compression_DC = hj.compress(list_image_cat_DC, huffman_code_generate_DC) ###

# Decompress with Huffman
huffman_decoding_dict_DC = hj.build_decoding_dict(huffman_code_generate_DC)

# decompressed_cat_DC should be the output of your Huffman decompressor
decompressed_cat_DC = hj.decompress(huffman_code_compression_DC, huffman_decoding_dict_DC) ###
#decompressed_cat_DC = list_image_cat_DC

# ------------------------------------------------------------------------------------------

# ---- AC components processing
# RLE coding over AC components

AC_coeff = image_AC[:,i_plane]
[AC_coeff_rl, AC_coeff_amp]= fc.RLE(AC_coeff)
list_image_rl_AC = np.ndarray.tolist(AC_coeff_rl)

# --------------------------Students work on AC components ----------------------------------
# Compress with Huffman

alphabet_AC = list()

for i in list_image_rl_AC:
    for j in i:
        if j in alphabet_AC:
            continue
        alphabet_AC.append(j)

pixels_probability_AC = hj.dict_freq_numbers_2(list_image_rl_AC, alphabet_AC)
pixels_probability_AC_dict = pixels_probability_AC[0]
huffman_tree_AC = hj.build_huffman_tree(pixels_probability_AC_dict)
huffman_code_generate_AC = hj.generate_code_2(huffman_tree_AC)
huffman_code_compression_AC = hj.compress_2(list_image_rl_AC, huffman_code_generate_AC)


# Decompress with Huffman
huffman_decoding_dict_AC = hj.build_decoding_dict(huffman_code_generate_AC)

# Rdecompressed_cat_AC should be the output of your Huffman decompressor
decompressed_cat_AC = hj.decompress(huffman_code_compression_AC, huffman_decoding_dict_AC)


# --------------------------------Students work on the nb_bit/pixel ----------------------

total_bits_compressed_image += len(huffman_code_compression_DC)+len(huffman_code_compression_AC)
compression_ratio = (80*80*8*3)/total_bits_compressed_image
```

```
# ------------------------------------------------------------
#                       Decompression
#  ------------------------------------------------------------
    # ---- DC components processing
    # Map the  categories and amplitudes DC components back into values
    decompressed_cat_DC = np.array(decompressed_cat_DC)
    image_DC_DPCM_rec = (fc.cat_ampl_to_DC_vect(decompressed_cat_DC,image_DC_DPCM_amp))

    # DPCM decoding of DC components
    image_DC_rec[0,i_plane]=image_DC_0
    for i_block in range(1,N_blocks):
        image_DC_rec[i_block,i_plane]= image_DC_rec[i_block-1,i_plane] + image_DC_DPCM_rec[i_block-1]

    # ---- AC components processing
    # Map AC components back into their original values
    decompressed_cat_AC = np.array(decompressed_cat_AC)
    image_AC_rec[:,i_plane] =  fc.RLE_inv(decompressed_cat_AC,AC_coeff_amp)

    for i_row in range(0,n_row,8):
        for j_col in range(0,n_col,8):

            block_number = (np.round(i_row/8*(n_row/8)+ j_col/8 )).astype(np.int32)

            # Combining AC and DC components
            image_dct_quant_zgzg_rec = np.zeros(64, dtype= np.int32)
            image_dct_quant_zgzg_rec[1:63]= image_AC_rec[block_number*63 + 1:(block_number+1)*63,i_plane]
            image_dct_quant_zgzg_rec[0] = image_DC_rec[block_number,i_plane]

            # Inverse zigzag reading
            image_dct_quant_rec = fc.zigzag_inv(image_dct_quant_zgzg_rec)

            # De-Quantization
            image_dct_rec = image_dct_quant_rec*Q_matrix

            # Inverse DCT
            image_rec =  fc.DCT_inv(image_dct_rec)
            image_plane_rec[i_row:i_row+8, j_col:j_col+8,i_plane ] = image_rec.astype(np.uint8)


# Recovering the image from the array of YCbCr
image_ycbcr_rec = im.fromarray(image_plane_rec,'YCbCr')

# Convert back to RGB
image_rec =  image_ycbcr_rec.convert('RGB')

# Plot the image
pyplot.imshow(image_rec)

image_rec.save('Image_with_my_code.png')
```