Task 5

Compression, closely oriented on the JPEG 2000 Standard

Christian Riess (christian.riess@fau.de)

1 Summary

JPEG2000 is an algorithm for image compression. It was standartized by the *Joint Photographic Experts Group* in the year 2000. Although not widely adopted in practice, this algorithm outperforms the older JPEG image compression on photographic pictures. At the core of JPEG2000 lies a 2-D wavelet transform, but you will notice that there is a significant amount of work around the wavelet transform, to make the image representation really compact. Strictly speaking, the JPEG2000 standard defines a family of compression formats. *Lossless compression* uses the wavelet *CDF 5/3*. *Lossy compression* uses the wavelet *CDF 9/7* with additional quantization and omission of insignificant coefficients. In this exercise, we implement a simplified version of the lossy compression scheme.

Here is an overview on the JPEG2000 encoding steps, with comments on variations in our task:

1. Shifting of the input values to the domain of $[-127; 128]$.
2. Conversion of RGB colors to another color space **(omitted)**
3. Image tiling for more efficient processing **(omitted)**
4. Multiresolution analysis with wavelet CDF 9/7 **(slightly varied: we use *bior4.4*)**
5. Level-adaptive quantization of the coefficients
6. Subdivision into small code blocks
7. Bit-level encoding of code blocks, and further bit-level processing **(omitted)**
8. Arithmetic coding **(varied: we use Huffman coding)**
9. Partitioning of coded data into packets **(omitted)**

Decoding of a JPEG2000 image inverts this processing chain.

2 Encoding and Decoding

Please use your methods `mra2D` and `mrs2D` from the previous exercise as a basis for this implementation. Load an image and make sure that image height and image width are powers of two. Throughout this worksheet, we assume that we only deal with 8 bit images (i.e., intensities range between 0 and 255).

1. First, shift the input values to a domain of $[-127; 128]$.

2. Implement the function `compress` for encoding the input image. As CDF 9/7 is not part of Matlab's wavelet toolbox, we use here `bior4.4`. Use the Matlab function `wfilters` to obtain the filter coefficients. Perform a 2-D multi-scale wavelet decomposition with these filters.

3. Implement quantization of the obtained wavelet coefficients in the function `quantize`. To achieve this, define first a matrix with the same size as the wavelet coefficients. Each entry of this matrix shall contain the quantization level Δ_b for the respective wavelet coefficient.

   Let the maximum decomposition level be L, and an intermediate level l, $0 < l \leq L$. Furthermore, let τ denote a user-chosen quantization strength that is passed as an argument to `quantize`. Then, Δ_b is defined at

   - Approximation (at level L): $\Delta_b = 2^{-L} \cdot \tau$
   - DetailH, DetailV (at every level l): $\Delta_b = 2^{-(l-1)} \cdot \tau$
   - DetailD (at every level l): $\Delta_b = 2^{-(l-2)} \cdot \tau$

   The quantization levels are applied to the coefficients by using the equation

   $$w' = \text{signum}(w) \cdot \left\lfloor \frac{|w|}{\Delta_b} \right\rfloor$$

4. Implement the function `encode` for encoding the quantized coefficients. First, decompose the coefficients into blocks of 8×8 entries. The goal is to have a stream of 8×8 blocks representing the whole image. We will add processing for each 8×8 block in the submethod *encode_block* (see description below). The result of *encode_block* is a list of code numbers in the range of 0 and 64 (inclusively). Concatenate these numbers to a list and perform Huffman coding to reduce the bitlength of the list. For Huffman encoding, you may find the methods `huffmandict` and `huffmanenco` useful. In a real compression algorithm, the Huffman dictionary would have to be stored together with the encoded stream. We omit this step, and just return the encoded stream and the dictionary to the calling function.

5. Implement the function `encode_block` for encoding a block of 8×8 quantized coefficients. Decompose the coefficients into bit planes. First, it is necessary to find out how many bit planes are required. This depends on the maximum value of the coefficients. Each bit plane is subject to run length encoding. Thus, instead of storing the values of a bit plane directly, store the number of subsequent zeros and ones. For example, "11100010" becomes "3311". If the bit plane starts with a zero, the resulting sequence shall also start with a zero. For example, "00011101" becomes "03311". Note that matlab's matrix trickery allows run length encoding without an explicit for-loop. It may be worth to google for or think about such a solution.

   The fully encoded block shall first contain one number indicating the number of bit planes, and then for each bit plane the number of values in the sequence, then the sequence itself. Return this list of values to the calling function.

   There is one important detail to take care of, prior to run length encoding: negative coefficients must be represented. A good strategy is to append the sign to the binary representation of a number. For example, "3" (binary "110") becomes "1100", "-3" becomes "1101".

6. Implement the function `decode_block`. It shall invert the function `encode_block`, i.e., undo the run length encoding for each bit plane, and reconstruction of the bit plane. This is a great opportunity to test your implementation of the block encoder and decoder: feed artificial input to `encode_block`, and check whether a subsequent call to `decode_block` leads to the original input. Please also test our method with negative coefficients and zero coefficients.

7. Implement the function `dequantize`. First, construct the matrix of quantization levels. Then, multiply the quantization levels to the coefficients.

8. Implement the function `decompress`. It calls `decompress_block` and `dequantize`. Then, perform multiresolution synthesis using `bior4.4`.

3 Experiments with the Code

1. Make sure that what the code that you have implemented works. Compare the input image with the output image by visualizing the differences between them, and by computing the root mean squared error.

2. Think about the stream encoding as a whole: how does the subdivision into 8×8 blocks interact with the Huffman encoding?

3. In `encode_block`, we appended the sign to the binary representation of a number. Why is this a better strategy than, e.g., adding an offset to each number to end up with all-positive values?

4. Evaluate different compression parameters τ: how do compression rate and deviations from the original image change? (Approximately) compare your compression method to JPEG 1992 images and the lossless PNG format. How well are we performing?

5. Finally, if you are curious, here is a link to a draft for the first ISO standard for JPEG2000: `http://nmdos.zesoi.fer.hr/projekt/2006-2007/jpg2000/Literatura/JPEG2000_PosljednjiDraftNijeKonacnoISOIEC 15444-12000.pdf`.