Lab 12

Basic Image Processing Algorithms
Fall 2022

Introduction to image compression

Compression is the reduction of the number of bits used for the representation of an image, while

- being able to reconstruct the original data (**lossless compression**) OR
- maintaining an acceptable quality of the reconstructed data (**lossy compression**)

Compression ratio: original size / compressed size

The images are compressible, because

- they contain spatial (or temporal) redundancy,
- they have a structure which can be described in a more compact way,
- they contain parts which are perceptually irrelevant.

Information Theory – some definitions

Source: an information generating process which emits random sequence of symbols from a finite alphabet. Alphabet can be

- in natural written languages: letters + punctuation + space + numbers
- in 8-bit grayscale images: different shades of gray (2^8 symbols)

Discrete Memoryless Source (DMS): the generated successive symbols are independent and identically distributed at each time

Information Theory – some definitions

Self Information: Property of the symbol: “How much information is provided by the emission of a certain symbol?” The occurrence of a less probable event provides more information.

$$\text{info}(s_i) = -\log(p_i)$$

where the S source emits the s_i signal with p_i probability.

Entropy: Property of the source: “What is the average information per symbol?” The entropy is maximal if the probability of the symbols are equal; and minimal if one symbol has a probability of 1 while the others are 0.

$$H(S) = \sum_i p_i \,\text{info}(s_i) = -\sum_i p_i \log(p_i)$$

Information Theory – some definitions

First order codes: encodes each symbol independently, every symbol has a codeword (e.g. on a binary image we define codes for black and white pixels.)

□□□□■■□■■■□□□□□□□■■□□ → 000011011100000001100

Block codes: group the symbols of the source into N length blocks and generate a codeword for each block (e.g. on a binary image groups of 4 are coded)

□□□□■■□■■■□□□□□□□■■□□ → 01202

Other codes exist too...

Shannon’s Source Coding Theorem

Let S be a source with alphabet size n and entropy $H(S)$ and let consider coding N source symbols into one binary code word (block coding). Then for every $\delta > 0$ it is possible by choosing N large enough, to construct a code with average bits per symbol l_{avg} that satisfies the following inequality:

$$H(S) \leq l_{\text{avg}} < H(S) + \delta$$

<u>This means:</u>

- Entropy is the lower bound of the code efficiency, we cannot beat it
- but we can come arbitrarily close to it by increasing N

Increasing N results larger dictionary and a delay in decoding. In general it is not straightforward to calculate entropy (the formula is only for DMS).

Lossless Compression

Lossless Compression

Reversible process: the original data can be exactly reproduced from the compressed data.

Only limited compression ratio can be achieved with lossless compression, determined by the entropy of the source data.

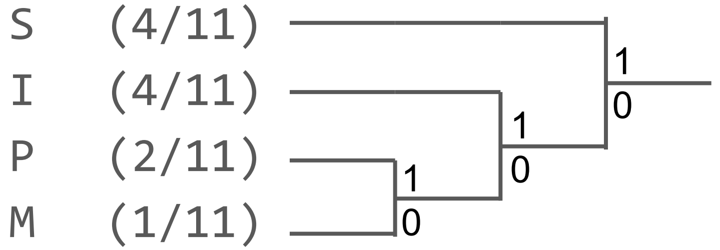
There is a tradeoff between

Efficiency	(compression ratio)
Complexity	(required memory, computational power etc.)
Coding delay	(how long does it take to code the signal)

Lossless Compression – examples

Huffman coding: variable length, prefix code; the more common symbols have shorter codeword. To create the code, we sort the symbols according to their probability, and repeatedly, combine the two least probable symbols to a composite symbol until only one composite symbol remains.

Example: Let the signal be `[MISSISSIPPI]`, the dictionary is [`MISP`]. In this, the probabilities of the symbols and hence the tree is the following:

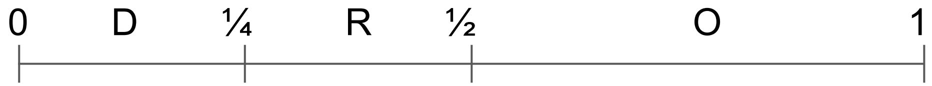


original	M	I	S	S	I	S	S	I	P	P	I
encoded	000	01	1	1	01	1	1	01	001	001	01

Lossless Compression – examples

Arithmetic coding: A unique **tag** is generated from the sequence of symbols, and then, this tag is coded into a binary code. The tag is from the [0, 1) interval (infinitely many tags are possible), tags are produced using recursive partitioning.

Example: Let the signal be `[DOOR]`, the dictionary is [`DRO`]. The frequencies of the symbols are mapped to the [0,1) interval:



To encode a signal, the appropriate intervals will be recursively used. This will produce a final interval (tag), which can be coded into binary form.

Lossless Compression – examples

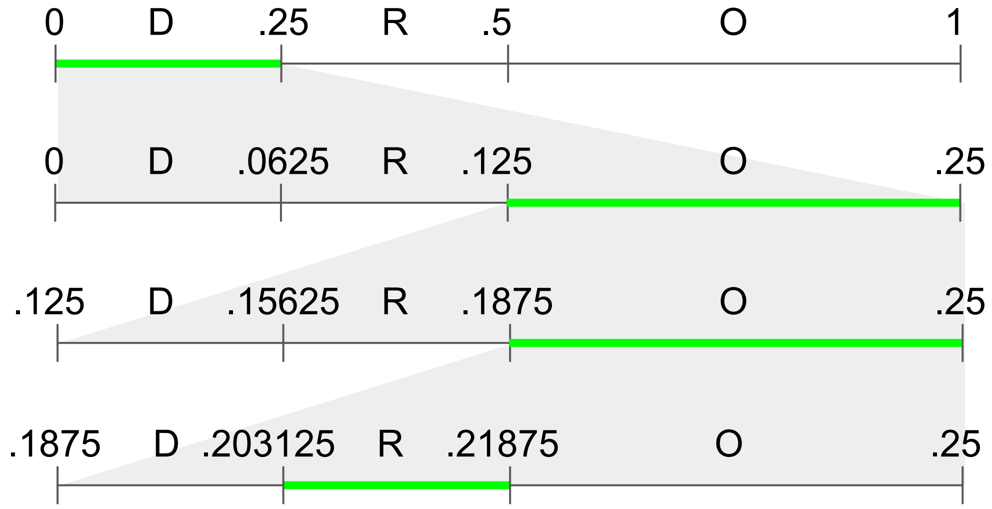
The process of encoding of DOOR is the following:

The green tag encodes “D”:

The green tag encodes “DO”:

The green tag encodes “DOO”:

The green tag encodes “DOOR”:



The limits of the final tag are $0.203125_{10} = 0.001101_2$
and $0.21875_{10} = 0.00111_2$

Lossless Compression – examples

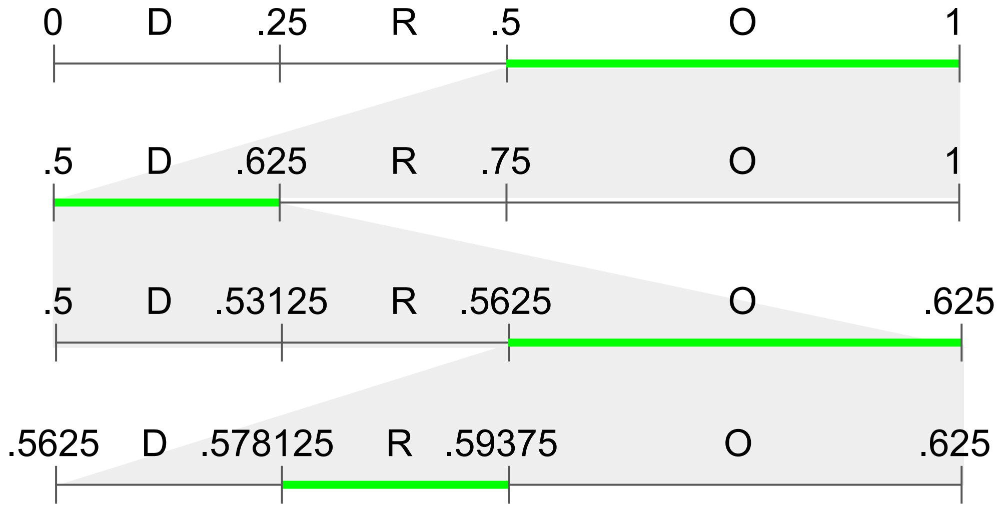
The process of encoding of ODOR is the following:

The green tag encodes “O”:

The green tag encodes “OD”:

The green tag encodes “ODO”:

The green tag encodes “ODOR”:



The limits of the final tag are $0.578125_{10} = 0.100101_2$
and $0.59375_{10} = 0.10011_2$

Lossless Compression – examples

Dictionary coding: In many applications there are frequently repeated patterns emitted by the source. It can be efficient to create a list (a dictionary) of the most frequent patterns, so they can be encoded by their address in the dictionary.

Example: Let the signal be the following:

```
[Never gonna give you up
 Never gonna let you down
 Never gonna run around and desert you]
```

With the following dictionary:

```
0 → Never gonna
1 → ou
2 → nd
```

the original signal can be encoded as

```
[0 give y1 up
 0 let y1 down
 0 run ar12 a2 desert y1]
```

Credit: https://www.youtube.com/watch?v=dQw4w9WgXcQ

Now please

download the ‘Lab 12’ code package

from the

moodle system

Exercise 1

Implement the function `lossless_compress` in which:

- The input is a binary, non-compressed image (in logical format).
- Compress the image with the method described on the next slide.
- Return the compressed image as a cell array.

Test the implementation using `test1.m`

Implement the function `lossless_decompress` in which:

- The input is the compressed image (cell) created with the previous function.
- Decompress the image which was compressed using the aforementioned method
- Return the decompressed image as a logical array.

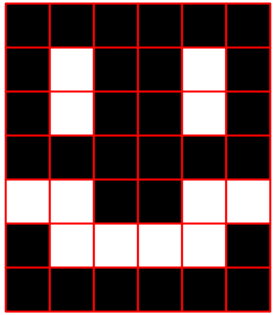
Test the implementation using `test2.m`

Run `test_lossless.m` and examine the results.

Exercise 1 – Compression method

The compressed image (which is a cell) has the same number of rows as the original image. Every row is a vector. In every row, the first element of the vector is a logical value, the color of the first pixel in that row. Then the lengths of the constant color blocks are listed.

Example:



This is the color of the first block

These are the lengths of the blocks

```
compressed{1} = [0 6]
compressed{2} = [0 1 1 2 1 1]
compressed{3} = [0 1 1 2 1 1]
compressed{4} = [0 6]
compressed{5} = [1 2 2 2]
compressed{6} = [0 1 4 1]
compressed{7} = [0 6]
```

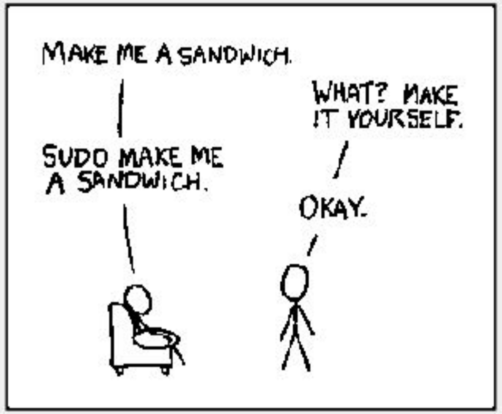
Exercise 1 – A good way to implement compression

```
Create a H×1 cell (H is the height of the image) which will be the compressed_image.
For each row (y = 1:H)
    Create a variable symbol and set its value to the first pixel's value in this row.
    Create a variable counter and set its value to 1.
    Create a variable row_desc and set its value to [symbol] (so we put the color
    of the first block into the row description vector). Then
    For each pixel in this row, starting from the second one (x = 2:W) (W is the width of the image)
        If symbol has the same color as the current pixel:
            increment counter: counter = counter + 1
        Else
            Invert symbol
            Append the counter's value to the row description: row_desc = [row_desc, counter]
            Reset the counter: counter = 1
    After this loop, append the last counter value to the row description too.
    Save the row_desc into the compressed_image at position {y}.
```

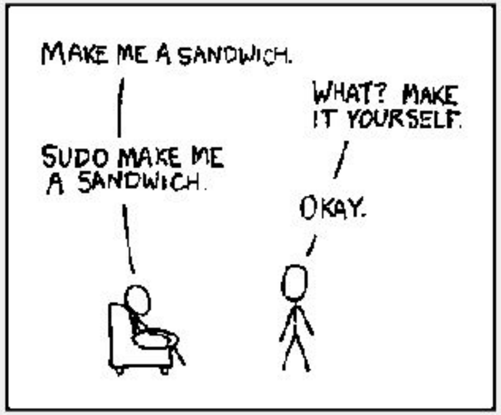
Exercise 1 – A good way to implement <u>decompression</u>

```
Create an empty (I mean [] ) matrix which will be the decompressed_image.
For each row in the compressed image (y = 1:H)
    Create a variable symbol and set its value to the first value in this row (that's the first block's color)
    Create a variable row_pixels and set its value to [] (so it contains nothing). Then
    For each number in this row, starting from the second one (x = 2:N) (so only for the block sizes)
        Create a variable block which is a row vector containing the pixel intensity stored in symbol
        and the length of this block should come from the compressed image. Use repmat():
        block = repmat(symbol, 1, compressed_image{y}(x))
        Append this block to the row of pixels: row_pixels = [row_pixels, block]
        Invert symbol
    Save the row_pixels into the decompressed_image at position (y, :).
```

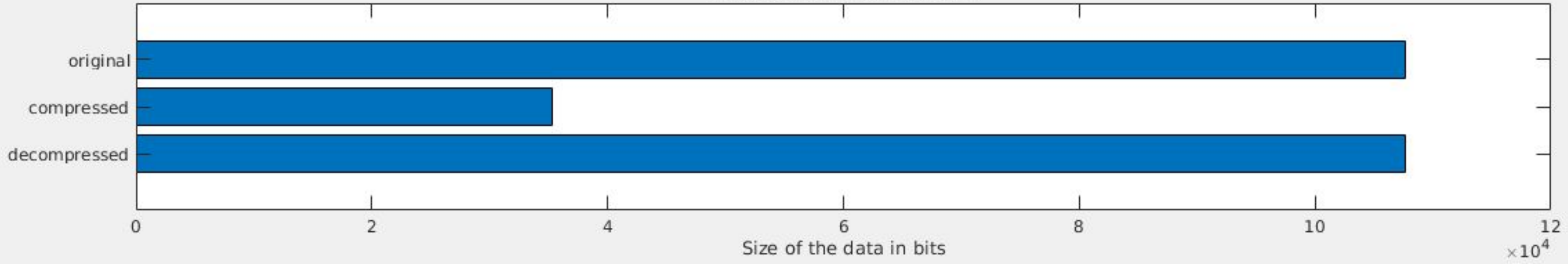
Original image
Size: 107640 bits



Decompressed image
Size: 107640 bits



Statistics
Compression ratio: 3.0521



Lossy Compression

Lossy Compression – examples

Scalar quantization: The values of the signal are quantized using a uniform or non-uniform quantizer.

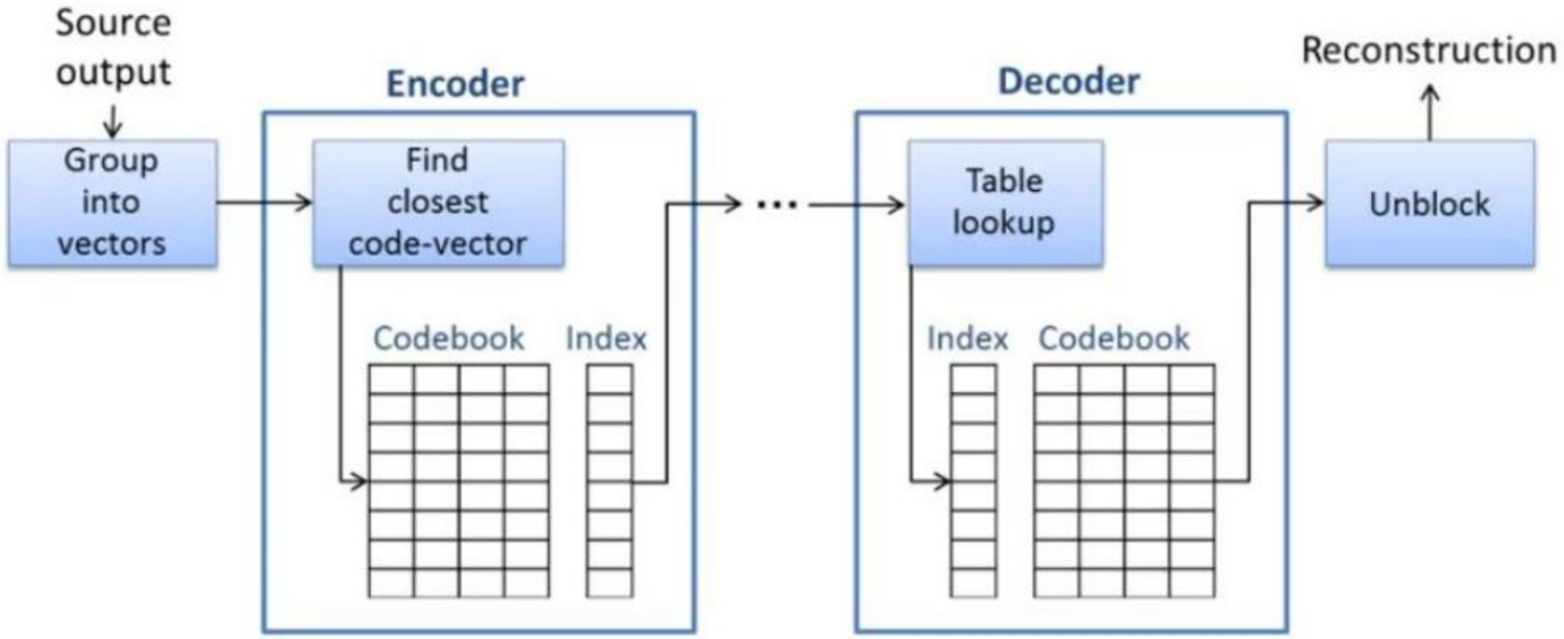
Example: The input signal is the following: `[1 2 3 4 5 4 5 4 5 5 6 7 8 9]`.

```
    UNIFORM quantizer: A:{1,2,3}   B:{4,5,6} C:{7,8,9}
NON-UNIFORM quantizer: A:{1,2,3,4} B:{5}     C:{6,7,8,9}
```

```
Result of the uniform method:       [A A A B B B B B B B B C C C]
Result of the non-uniform method:   [A A A A B A B A B B C C C C]
```

Lossy Compression – examples

Vector quantization: The input image is divided into small blocks, which are coded using a look-up-table.



Exercise 2

Implement the function `lossy_compress` in which:

- The input is a grayscale, non-compressed image (in uint8 format).
- Compress the image with the method described on the next slide.
- Return the compressed image as a struct.

Implement the function `lossy_decompress` in which:

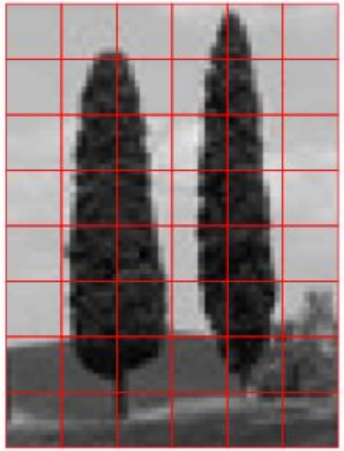
- The input is the compressed image (struct) created with the previous function.
- Decompress the image which was compressed using the aforementioned method
- Return the decompressed image as an uint8 array.

Run `test_lossy.m` and examine the results.

Exercise 2 – Compression method

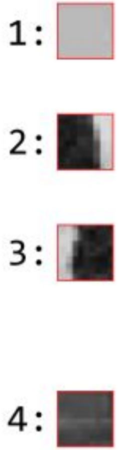
The compressed image representation is a dictionary and a list of indices. The input image is decomposed into $B \times B$ size blocks (if the size of the image is not divisible by B then discard the last rows and columns). Then k significant blocks are selected and put into a dictionary ($DICT$). Meanwhile, every block of the image is encoded as the index of the block in $DICT$ which is closest to the actual image block (LUT).

Non-compressed image



Compressed representation

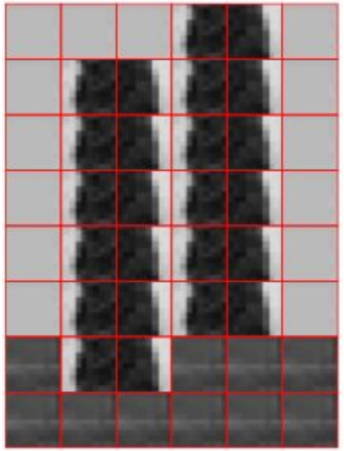
DICT



LUT

1	1	1	3	2	1
1	3	2	3	2	1
1	3	2	3	2	1
1	3	2	3	2	1
1	3	2	3	2	1
1	3	2	3	2	1
4	3	2	4	4	4
4	4	4	4	4	4

Decompressed image



Exercise 2 – A good way to implement compression

```
Compute the cut size: cut_size = floor(size(input_image)/block_size) * block_size
Convert the input image to double: input_image = double(input_image)
Create an empty block array: LIST = []
For each row-block (r = 1:block_size:cut_size(1))
    For each column-block (c = 1:block_size:cut_size(2))
        Crop the block form the image:
        crop = input_image(r:r+block_size-1, c:c+block_size-1);
        Squeeze the cropped area into a row vector and append it to the LIST. The LIST should be a
        matrix, each row is a flattened block: LIST = [LIST; crop(:)']
```

Do a k-means clustering on the `LIST` matrix, the result of the `kmeans` function will be the `LUT` and the `DICT` (in this order). You should return the `compressed` struct, where

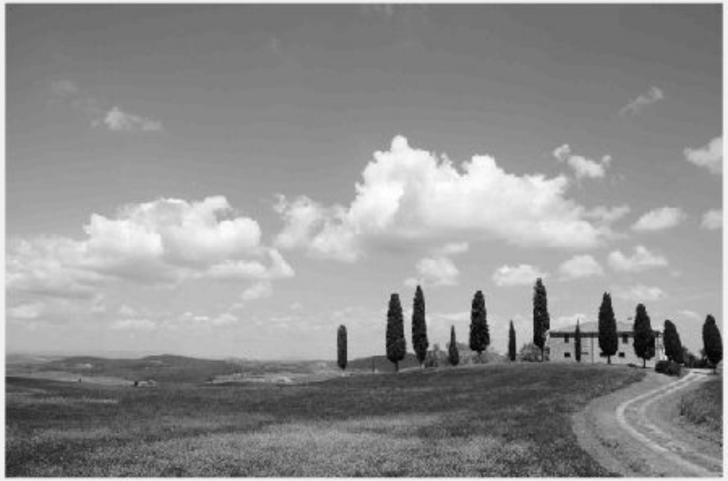
```
compressed.DICT = DICT
compressed.LUT = LUT
compressed.cut_size = cut_size
```

Exercise 2 – A good way to implement <u>decompression</u>

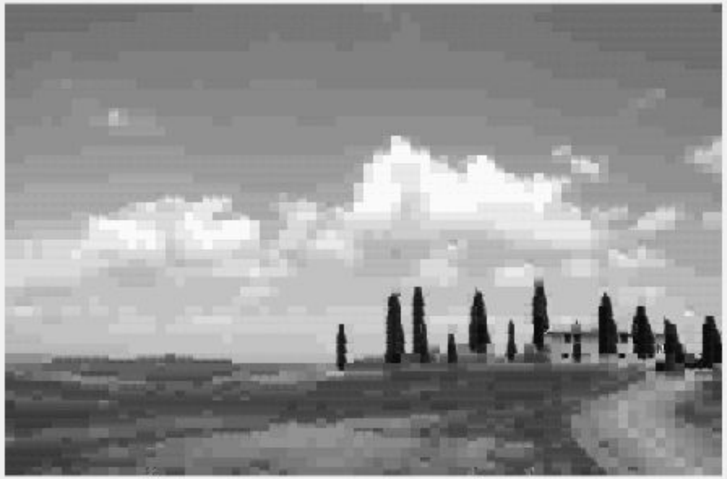
```
Create a variable LIST in which you restore the block list using DICT and LUT: LIST = DICT(LUT, :)
Initialize the decompressed image as uint8(zeros(cut_size))
Compute the block_size from the LIST: block_size = sqrt(numel(LIST(1,:)))
Create a counter and set it to 1: k = 1
For each row-block (r = 1:block_size:cut_size(1))
    For each column-block (c = 1:block_size:cut_size(2))
        Restore the block from the appropriate row vector. Reshape it to block_size×bock_size, and save
        it into the decompressed image to the r, c block:
        part = reshape(LIST(k,:), block_size, block_size)
        decompressed(r:r+block_size-1, c:c+block_size-1) = part
        Increment the k counter.

Finally, return the decompressed image as a uint8 image.
```

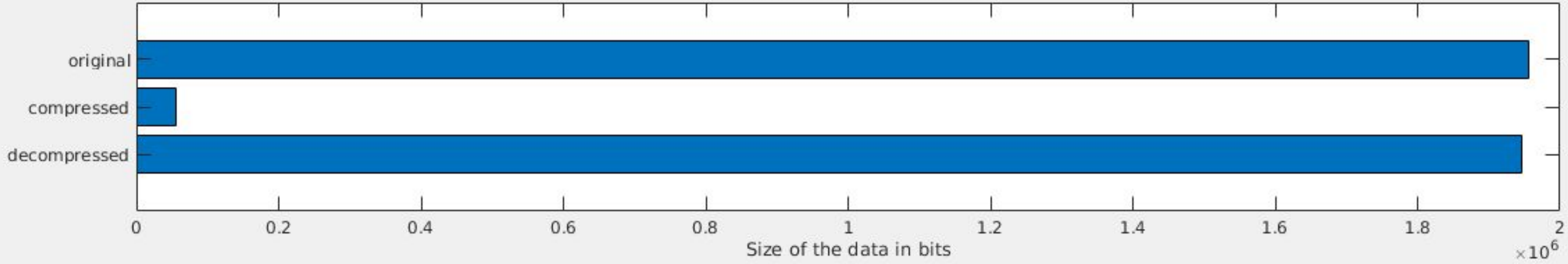
Original image
Size: 1955328 bits



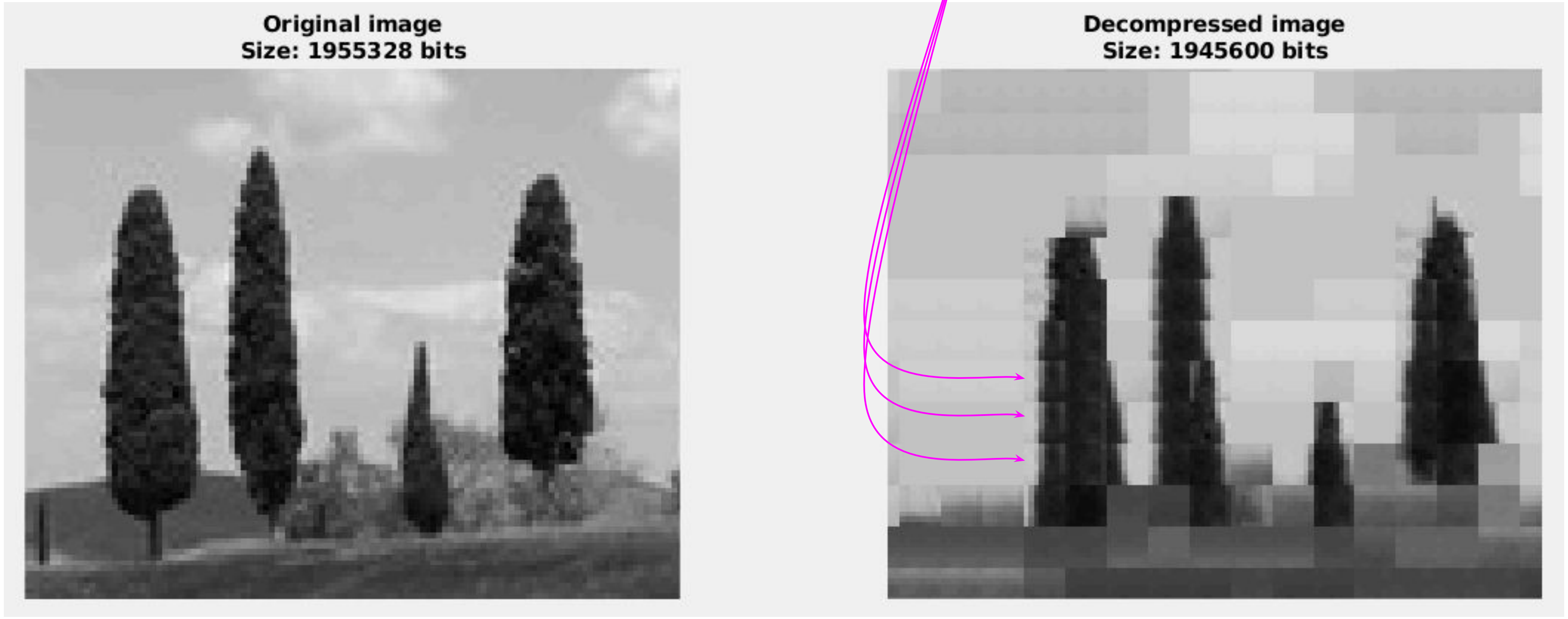
Decompressed image
Size: 1945600 bits



Statistics
Compression ratio: 34.7329



Magnification of the results. Please note that the trees on the right side are built up from different tree-like blocks



THE END