Assignment 4

Basic Image Processing
Fall 2019

Overview

The scope of this assignment is image compression. You will see (and code) various examples of lossless and lossy image compression methods.

First part: **A lossless compression method**

Second part: **Dictionary-based lossy compression method**

Third part: **A naive, lossy, JPEG-like implementation**

By JPEG-like we mean that this is not the standardized jpeg compression algorithm; the result is not a valid JPEG file (the appropriate byte-sequence with the necessary header is missing), but the main steps of the transform coding algorithm are the same, and therefore you can understand the key ideas of the actual JPED compression method.

Theory

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 signal s_i with probability p_i

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.)

□□□□■■□■■■□□□□□□■■□□ → 00001101110000001100

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:

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

This means:

- 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.

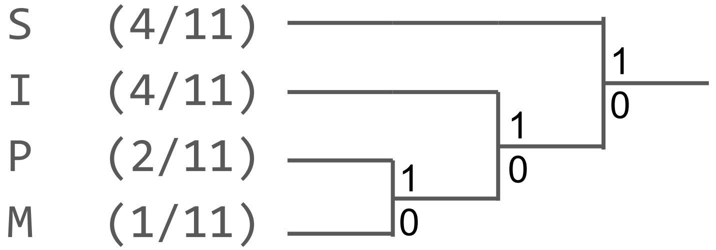
There is a tradeoff between

● 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:

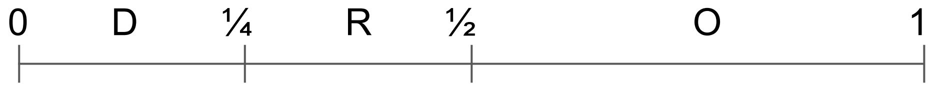


original	M	I	S	S	I	S	S	I	P	P	I
encoded	000	01	1	1	01	1	1	01	001	001	01

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

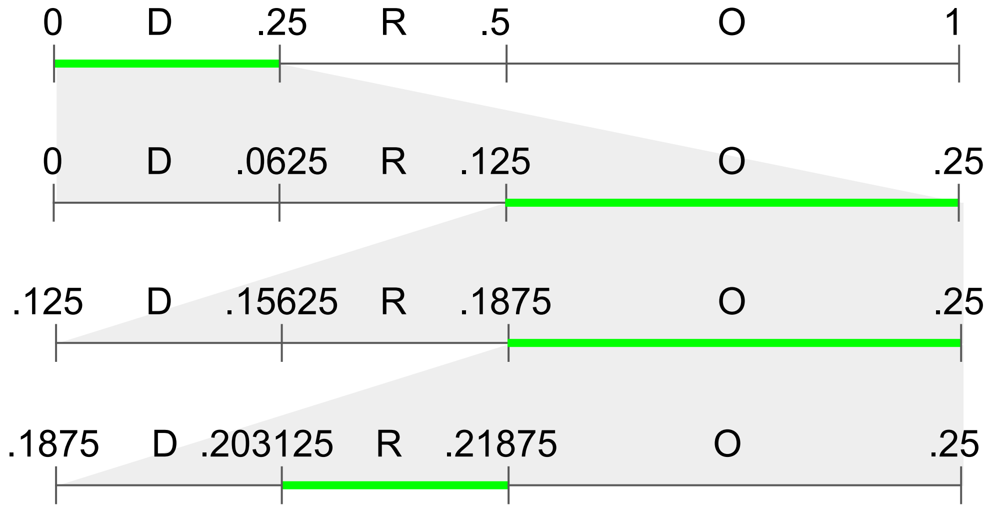
The process of encoding of DOOR is the following:

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

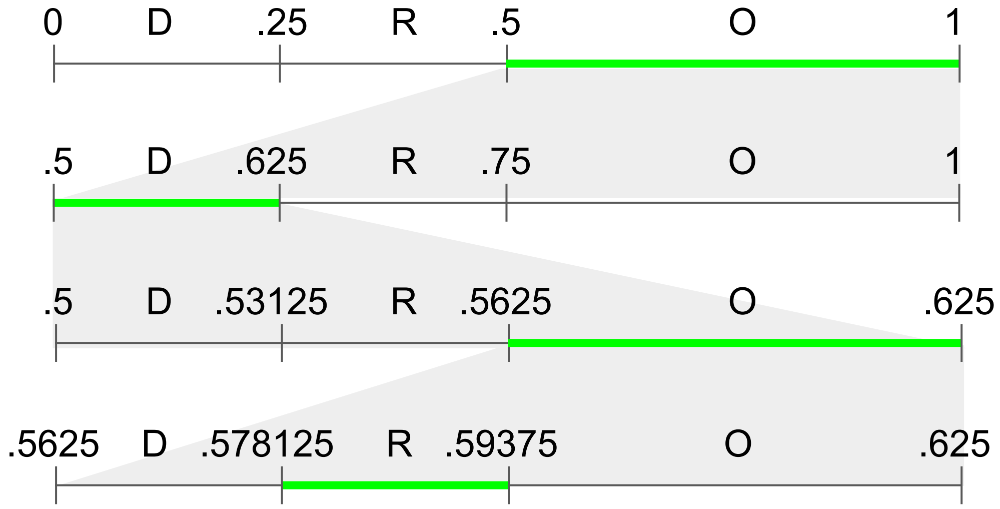
The process of encoding of ODOR is the following:

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

Let’s code!

Please

download the ‘Assignment 4’ code package

from the

submission system

The maximum score of this assignment is
7 points

The points will be given in 0.25 point units.
(Meaning that you can get 0, 0.25, 0.5, 0.75, 1, 1.25 etc. points).

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 `lossless_test_compress.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.

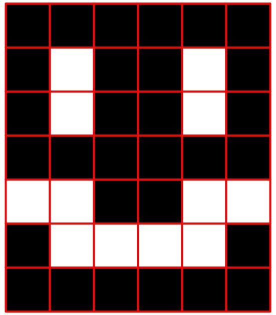
Test the implementation using `lossless_test_decompress.m`

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]
```

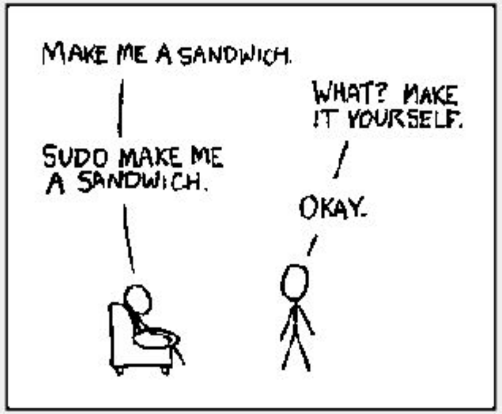
Exercise 1 – A good way to implement compression

```
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}.
```

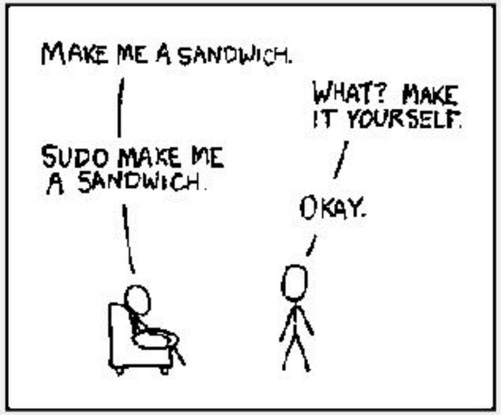
Exercise 1 – A good way to implement <u>decompression</u>

```
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, :).
```

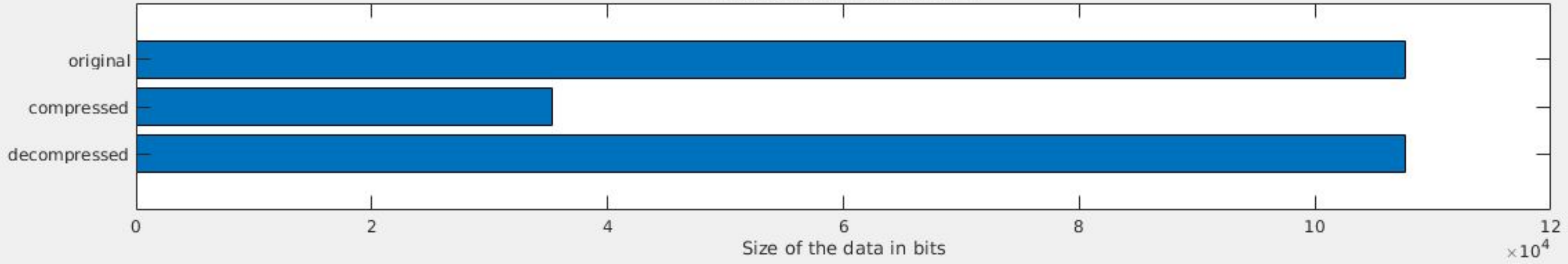
Original image
Size: 107640 bits



Decompressed image
Size: 107640 bits



Statistics
Compression ratio: 3.0521



Lossy Compression

VECTOR QUANTIZATION

Lossy Compression – examples

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

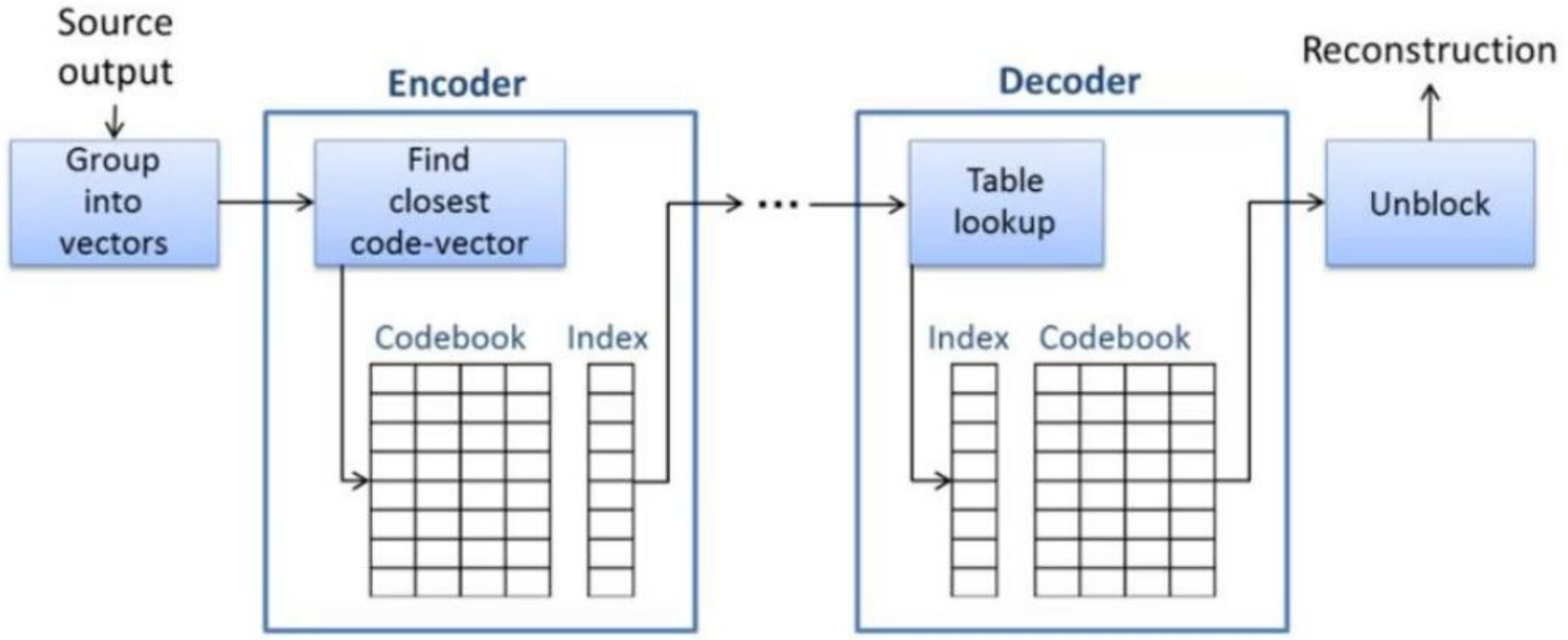
<u>*Example:*</u> The input signal is the following: `[1 2 3 4 5 4 5 4 5 5 6 7 8 9]`.

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:

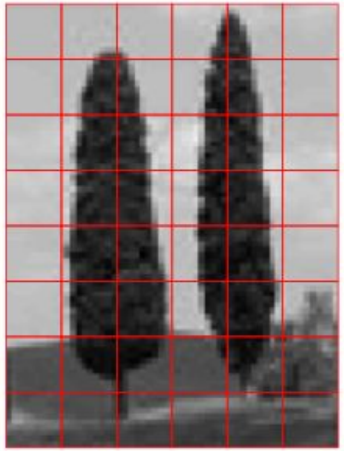
- The input is the compressed image (struct) created with the previous function.
- Decompress the image which was compressed using the aforementioned method
- Return the decompressed image as an uint8 array.

Run `test_lossy.m` and examine the results.

Exercise 2 – Compression method

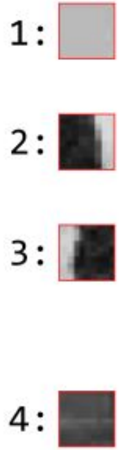
The compressed image representation is a dictionary and a list of indices. The input image is decomposed into $B \times B$ size blocks (if the size of the image is not divisible by B then discard the extra rows and columns). Then k significant blocks are selected and put into a dictionary ($DICT$). Meanwhile, 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

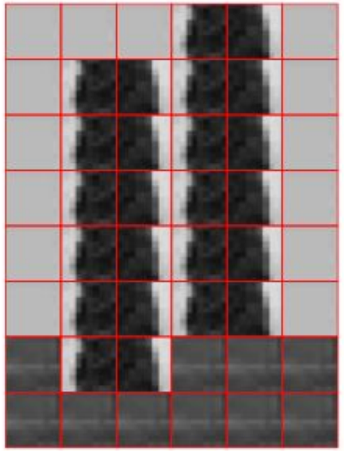
DICT



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 built-in `kmeans()` function will be the `LUT` and the `DICT` (in this order). You should return the `compressed` struct, where

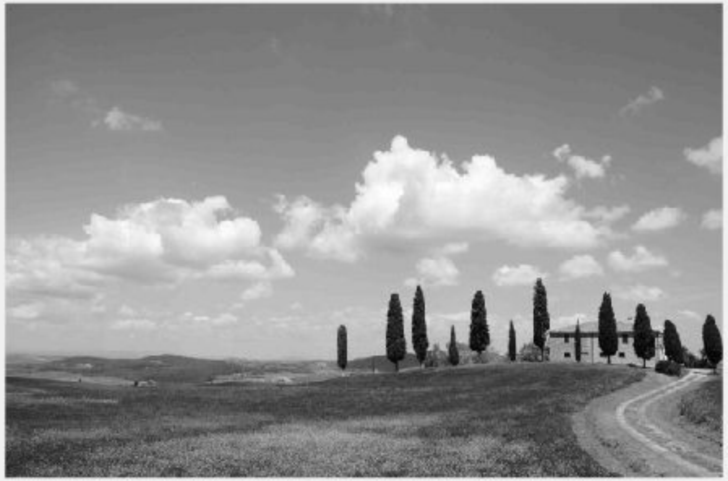
```
compressed.DICT = DICT
compressed.LUT = LUT
compressed.cut_size = cut_size
```

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

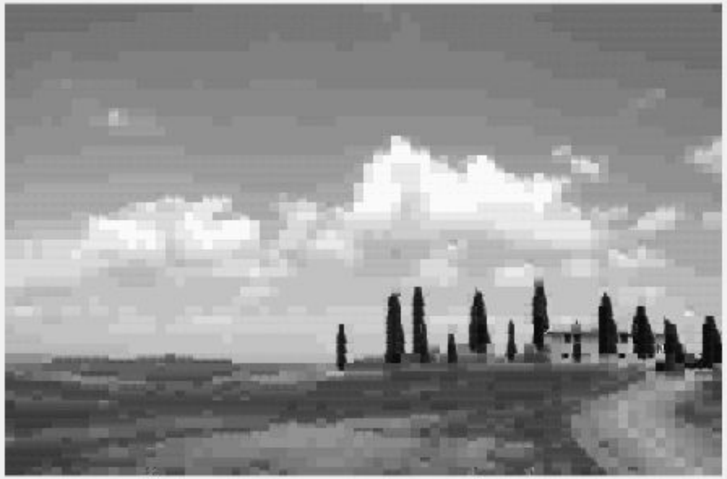
```
Create a variable LIST in which you restore the block list using DICT and LUT: LIST = DICT(LUT, :)
Initialize the decompressed image as uint8(zeros(cut_size))
Compute the block_size from the LIST:
Create a counter and set it to 1: k = 1
For each row-block (r = 1:block_size:cut_size(1))
    For each 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.

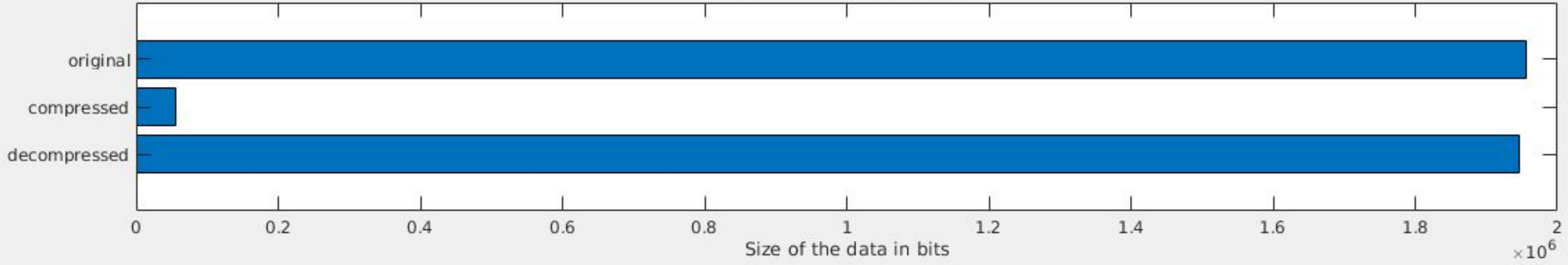
Original image
Size: 1955328 bits



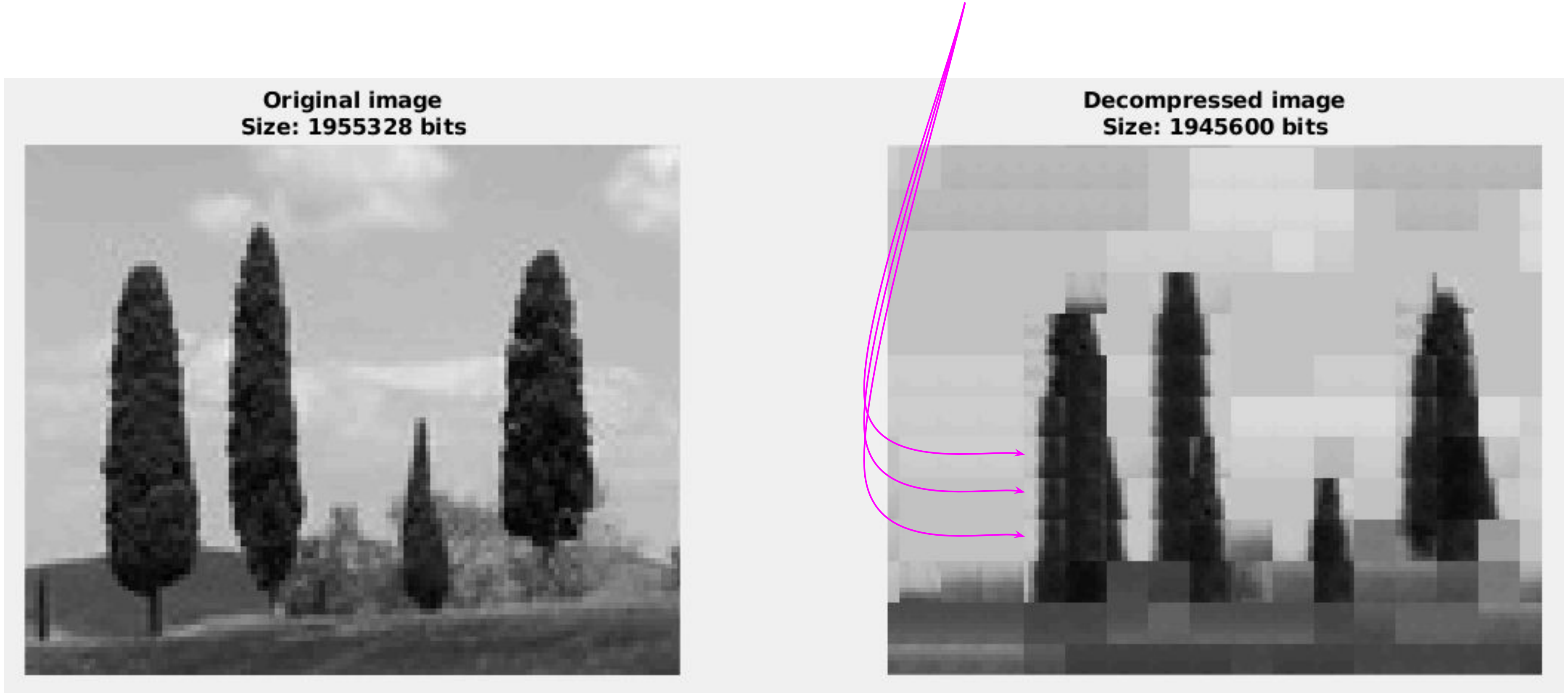
Decompressed image
Size: 1945600 bits



Statistics
Compression ratio: 34.7329



Magnification of the results. Please note that the trees on the right side are built up from n×n blocks of the same tree part.



Lossy Compression

TRANSFORM CODING

Lossy Compression – examples

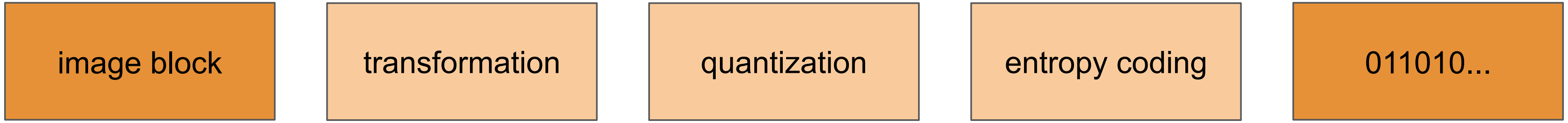
Transform coding:

- Very popular approach, it is part of most of the current image and video coding standards (JPEG, H.26X, MPEGX).
- The basic idea is to decorrelate the data with a suitable transformation, so that the transformation coefficients will describe the image perfectly.
- The transformation:
  - has to decorrelate the data,
  - has to compact the energy of the image,
  - has to have image independent basis,
  - has to have a fast implementation.
- We do coarse or fine quantization of the transformation coefficients based on their significance (their variance, or their contribution to the total energy of the image).

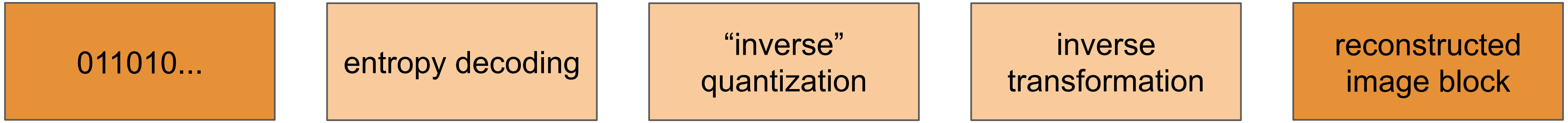
Lossy Compression – examples

Transform coding - continued:

encoding:



decoding:



Lossy Compression – examples

Joint Photographic Experts Group

- International standard since 1991.
- Capable of compressing continuous-tone still images (grayscale and color images) with ratio 10-50.

The JPEG algorithm:

- Uses Discrete Cosine Transform on 8x8 blocks:
    - the blocks’ gray level is shifted by -128 to the range [-128, 127];
    - the first coefficient is called DC, the rest: AC coefficients.
- The DC coefficients of the blocks are quantized, then coded differentially.
- The AC coefficients are first quantized, vectorized by zig-zag scan and then entropy coded.
- The quantizer is uniform and uses quantization tables with different step sizes for the different frequencies.

JPEG in practice

Open the script `test_jpeg` and please check the following parts:

- The input image is loaded, converted to **`double`**.
- We will work on small blocks with size of 8x8, for this reason the image is cropped to have the side-lengths as an integer multiple of 8.
- Space allocated to the output / decompressed image (**`decoded_img`**).
- With a huge **`for`**-loop the blocks are processed (both encoded and decoded) from left-to-right, from top-to-bottom - ***details on the next slide***.
- After the decompressed image is fully built, compression ratio is calculated and the images are visualized.

This script should be run **only when you are ready with all of the necessary functions** (5 × 2), right now the aim is at the understanding of the process as a whole.

JPEG in practice – still `script test_jpeg`

A image block (a 8×8 part) goes through 5 steps during its encoding/decoding. For every *compression* function we have the appropriate *decompression* one.

`jpeg_RGB_to_YCbCr`: converts the color domain to luminosity and chrominance values
`jpeg_YCbCr_to_RGB`: convert the luminosity and chrominance values back to an RGB array

`jpeg_DCT`: applies Discrete Cosine Transform on the supplied channel
`jpeg_IDCT`: applies Inverse Discrete Cosine Transform

`jpeg_quantizer`: quantizes the DCT-components with quantizer-tables
`jpeg_dequantizer`: reconstructs the DCT-components (*of course, with quantization error*)

`jpeg_zigzag`: reorders the quantized elements to have long runs of zero-valued coeffs
`jpeg_izigzag`: reorders the vector to have the quantized values as a normal 2D array

`jpeg_rle_encoder`: a simple run-length encoder to make compact representation to the 0s
`jpeg_rle_decoder`: reconstructs the long runs of zeros

1) the Holy Grail in this topic: https://www.w3.org/Graphics/JPEG/itu-t81.pdf , images are also taken from there
2) another useful overview (but we create slightly different realization): http://www.dmi.unict.it/~battiato/EI_MOBILE0708/JPEG%20(Bruna).pdf

Exercise 3

Implement the function `jpeg_RGB_to_YCbCr` in which:

- Please decompose your color (3D) image array to 2D matrices (`R`, `G` and `B`), in accordance with the color channels.
- Create the luminosity array (`Y`) and the two chrominance arrays (`Cb`, `Cr`) with respect to these formulas:

```
Y  =   0.2990 R + 0.5870 G + 0.1140 B
Cb = - 0.1687 R - 0.3313 G + 0.5000 B + 128
Cr =   0.5000 R - 0.4187 G - 0.0813 B + 128
```

- Shift the new values with -128 (the reason: the range of the DCT-components will be narrower this way).
- Sample down your chrominance arrays; take every 4th element (`1:4:end`) (With this the 8×8 blocks are reduced to 2×2 ones.)

further reading: https://www.w3.org/Graphics/JPEG/jfif3.pdf

Exercise 3

Implement the function `jpeg_YCbCr_to_RGB` in which:

- Please apply upsampling on your chrominance arrays (use `imresize`, with `nearest` option).
- Shift the range of the layers back with +128.
- Create the color (3D) array in which the layers are the following:

```
R = Y                            + 1.40200 (Cr-128)
G = Y - 0.34414 (Cb-128) - 0.71414 (Cr-128)
B = Y + 1.77200 (Cb-128)
```

Run `jpeg_test_colorspace_conversion.m` and examine the results.

Exercise 4

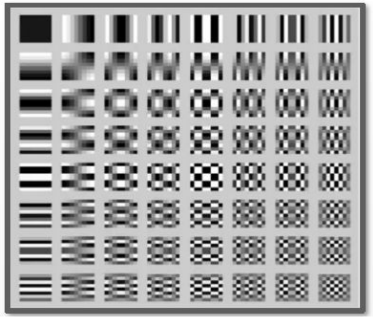
Implement the function `jpeg_DCT` in which you have to compute the Discrete Cosine Transform of the input array. The formula is on the next slide.

Discrete Cosine Transform (DTC):

- Represents finite sequence of data points with the sum of different cosine functions.
- It uses real numbers only, it can be considered as a frequency-spectrum description.
- Further properties: it is a linear transformation, it decorrelates the data, it compacts the energy of the image, it has image-independent basis, it has fast implementation, etc.
- There are different versions with minor differences (DCT-I to DCT-IV).

The figure on the right shows the basic DTC matrices for an 8×8 case:

- In the first row we have a cosine function with increasing horizontal frequency.
- In the first column we have a cosine function with increasing vertical frequency.
- The cells of table show the mixed horizontal and vertical frequencies.



Exercise 4

Computing the DCT

In your implementation you should realize the following formula (this is the DCT):

$$G_{u,v} = \sqrt{\frac{2}{N_1}}\sqrt{\frac{2}{N_2}}\alpha(u)\alpha(v)\sum_{x=1}^{N_1}\sum_{y=1}^{N_2} g_{x,y} \cos\left[\frac{(2(x-1)+1)(u-1)\pi}{2N_1}\right] \cos\left[\frac{(2(y-1)+1)(v-1)\pi}{2N_2}\right]$$

Where

$N_1 \quad N_2$	height and width of the image
$u = 1, \ldots, N_1$	vertical spatial frequency
$v = 1, \ldots, N_2$	horizontal spatial frequency
$x = 1, \ldots, N_1$	vertical pixel coordinate (row index)
$y = 1, \ldots, N_2$	horizontal pixel coordinate (column index)
$\alpha(u) = \begin{cases} \frac{1}{\sqrt{2}} & \text{if} \quad u = 1 \\ 1 & \text{otherwise} \end{cases}$	a normalization factor
$g_{x,y}$	pixel intensity value at (x, y)
$G_{u,v}$	the DCT coefficient at (u, v)

IMPORTANT NOTICE

You have to do the DCT for each (color) layer of the image separately!

Matlab starts indexing from 1; and the input-output array sizes should match.

Exercise 4

You also have to implement the function `jpeg_IDCT`

The formula for the inverse DCT is the following:

$$f_{x,y} = \sqrt{\frac{2}{N_1}}\sqrt{\frac{2}{N_2}}\sum_{u=1}^{N_1}\sum_{v=1}^{N_2}\alpha(u)\alpha(v)F_{u,v}\cos\left[\frac{(2(x-1)+1)(u-1)\pi}{2N_1}\right]\cos\left[\frac{(2(y-1)+1)(v-1)\pi}{2N_2}\right]$$

Where

$N_1, N_2, u, v, x, y, \alpha(u)$ as shown in the previous slide

$F_{u,v}$ the DCT coefficient at (u, v)

$f_{x,y}$ pixel intensity value at (x, y)

IMPORTANT NOTICE

You have to do the IDCT for each (color) layer of the image separately!

Matlab starts indexing from 1; and the input-output array sizes should match.

Run `jpeg_test_DCT_IDCT.m` and examine the results.

see: https://en.wikipedia.org/wiki/JPEG#Decoding

Exercise 5

Implement the function `jpeg_quantizer` according to the followings:

- The goal: to quantize the coefficients of the DCT transformation.
- Input parameters:
  - `dct_coeffs`: the coefficients of DCT, this array should be quantized;
  - `lumi_chromi`: there are different quantization tables for the luminance and chrominance arrays. This argument defines which one should be used (1: luminance quant., 2: chrominance quant.);
  - `QF`: quality-factor, influences the values of the quantizer tables.
- The output is the quantized array, please calculate this way:
  - `quantized_values` $=$ `round`(`dct_coeffs` / (`QF` $\times$ $QuantizationValues$))
- Please note:
  - you should dynamically downsample your *quantization table* depending on the size of the chrominance arrays, the downsampled table gives you the $QuantizationValues$. `(1:ratio:end)`
  - the division should be realized *elementwise*.
- Necessary luminance and chrominance quantization tables: exactly the same, as in the case of the **`jpeg_dequantizer`** - ***please see next slide***.

Exercise 5

Implement the function `jpeg_dequantizer`:

- The goal: from the quantized values get back the original DCT coefficients.
- Input parameters:
  - `quantized_values`: from these values should we calculate the coefficients of the DCT;
  - `lumi_chromi`: which quantization table to use (1: luminance quant., 2: chrominance quant.);
  - `QF`: quality-factor, influences the values of the quantizer tables.
- The output array contains the DCT coefficients, please calculate this way:
  - $\texttt{dct\_coeffs} = (\texttt{quantized\_values} \times (\texttt{QF} \times QuantizationValues))$
- Please note:
  - you should dynamically downsample your *quantization table* depending on the size of the chrominance arrays, the downsampled table gives you the $QuantizationValues$. `(1:ratio:end)`
  - the multiplication should be realized *elementwise*.

```
lumi_table = ...
[  16 11 10 16 24 40 51 61; ...
   12 12 14 19 26 58 60 55; ...
   14 13 16 24 40 57 69 56; ...
   14 17 22 29 51 87 80 62; ...
   18 22 37 56 68 109 103 77; ...
   24 35 55 64 81 104 113 92; ...
   49 64 78 87 103 121 120 101; ...
   72 92 95 98 112 100 103 99];
chromi_table = ...
[  17 18 24 47 99 99 99 99; ...
   18 21 26 66 99 99 99 99; ...
   24 26 56 99 99 99 99 99; ...
   47 66 99 99 99 99 99 99; ...
   99 99 99 99 99 99 99 99; ...
   99 99 99 99 99 99 99 99; ...
   99 99 99 99 99 99 99 99; ...
   99 99 99 99 99 99 99 99];
```

Run `jpeg_test_quantizer_dequantizer.m` and examine the results.

source of the values: http://www.dmi.unict.it/~battiato/EI_MOBILE0708/JPEG%20%28Bruna%29.pdf

Exercise 6

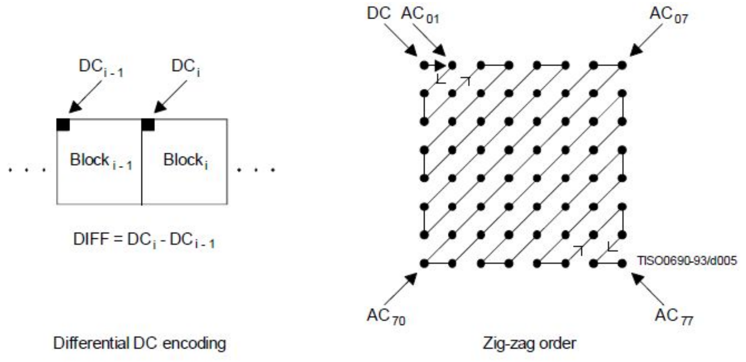


Figure 5 – Preparation of quantized coefficients for entropy encoding

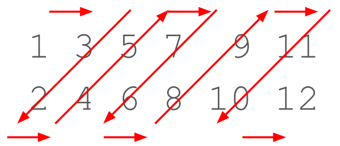
Implement the function `jpeg_zigzag` and the function `jpeg_izigzag` according to the followings:

- In our case, we are not going to deal with the DC-components separately, we will just leave them at the beginning of the zig-zag scan.
- Initial step: how to create an index-array by indexing an existing data-array column-wise?

```
idxs = reshape(1:numel(array), size(array));
```

- The zig-zag sequence of these indices:

  `1,3,2,4,5,7,6,8,9,11,10,12`



- Matlab recap:
  - if only one index present, it will be treated *columnwise*
  - `fliplr`: flips the array with respect to a vertical line
  - `flipud`: flips the array with respect to a horizontal line
  - `spdiags`: extracts the diagonals, from left to right, padding with zeros if necessary

source of the image: https://www.w3.org/Graphics/JPEG/itu-t81.pdf

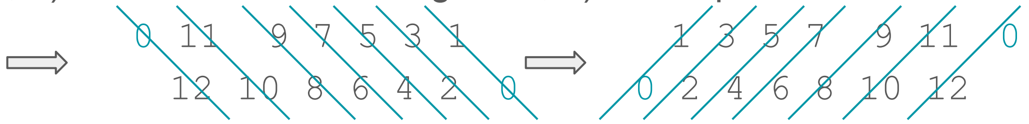
Exercise 6

Description of **function** `jpeg_zigzag` and **function** `jpeg_izigzag` continued:

- How to get *anti-diagonals* of an array?

  1) flip it left-to-right, 2) extract the normal diagonals, 3) then flip it back:

```
1 3 5 7  9 11
2 4 6 8 10 12
```

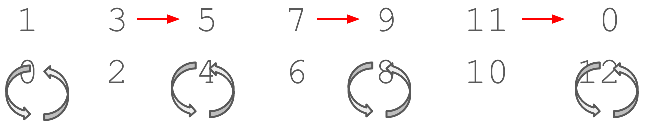


```
0 11  9 7 5 3 1          1 3 5 7  9 11 0
 12 10 8 6 4 2 0       0 2 4 6 8 10 12
```

  and you have the sequence columnwise:

```
1,0,3,2,5,4,7,6,9,8,11,10,0,12
```

- The `idxs` array again (after the `fliplr` - `spdiags` - `fliplr` triplet):



```
1   3 → 5   7 → 9   11 → 0
0   2   4   6   8   10   12
```

- As you see, you have to flip every odd column with respect to a horizontal line:

```
0   3   4   7   8   11   12
1   2   5   6   9   10   0
```

`idxs(:,1:2:end) = flipud(idxs(:,1:2:end));`

and this is now the standard MATLAB columnwise indexing!

unfortunately, the idea is not mine, but: http://stackoverflow.com/questions/3024939/matrix-zigzag-reordering

Exercise 6

Description of **function** `jpeg_zigzag` and **function** `jpeg_izigzag` continued:

- The only thing left is to remove zero elements: `idxs(idxs==0) = [];`
- You can use this index vector to select the elements from your original data array in zig-zag order:

```
zig_zag_vector=array(idxs);
```

end of
function
`jpeg_zigzag`

- And you can use this index vector to reorganize your matrix from a zig-zag ordered data sequence:

```
new_array=zeros(desired_size);
new_array(idxs)=zig_zag_vector;
```

end of
function
`jpeg_izigzag`

Specifications:

- **function** `jpeg_zigzag`
  - input: `array` - the data array to be zig-zag ordered
  - output: `zig_zag_vector` - the elements of the array, extracted in zig-zag order
- **function** `jpeg_izigzag`
  - input: `zig_zag_vector` - a vector, containing the elements from a rectangular array
    `desired_size` - the size vector of the original rectangular array before zig-zag
  - output: `new_array` - the reconstructed rectangular array

unfortunately, the idea is not mine, but: http://stackoverflow.com/questions/3024939/matrix-zigzag-reordering

Exercise 6

Description of function `jpeg_zigzag` and function `jpeg_izigzag` continued:

- Beginning of **function** `jpeg_zigzag`:

```
idxs = reshape(1:numel(array), size(array));
```

- Beginning of **function** `jpeg_izigzag`:

```
idxs = reshape(1:prod(desired_size), desired_size);
```

Run `jpeg_test_zigzag_izigzag.m` and examine the results.

unfortunately, the idea is not mine, but: http://stackoverflow.com/questions/3024939/matrix-zigzag-reordering

Exercise 7

Implement the **function** `jpeg_rle_encoder` and the **function** `jpeg_rle_decoder` according to the followings:

We will implement a modified version of the *run-length encoding*. The idea behind the zig-zag ordering is to have long runs of zero coefficients (the important elements of the DCT coeffs appear in the upper-left corner of the coefficients' matrix).

Initially the data we want to encode looks like this:

```
[1 0 0 0 0 0 0 0 0 21 10 3 0 0 0 0 0 0 0 5 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 1 0 9]
```

The goal is to get something that is represented on less bytes. We are going to use (*value, skip*) pairs to describe the data. The idea is that in a pair the

- *'value'* is the value of the next non-zero element, and
- *'skip'* tells the number of zeros in front of it.

With this encoding the data above becomes `[1 0 21 8 10 0 3 0 5 7 8 3 2 12 1 2 9 1]`

a more accurate run-length encoding: https://en.wikipedia.org/wiki/Run-length_encoding

Exercise 7 – spectacular explanation

Raw data:

`[1 0 0 0 0 0 0 0 0 21 10 3 0 0 0 0 0 0 0 5 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 1 0 9]`

Encoded data:

`[1 0 21 8 10 0 3 0 5 7 8 3 2 12 1 2 9 1]`

You can see that the (value, skip) pairs (shown underlined for better understanding) are concatenated into a vector. An issue with this encoding is that we cannot encode the raw data if it ends with a zero. Let's solve this by appending an extra number to the back telling the number of zeros after the last nonzero data point:

`[1 0 21 8 10 0 3 0 5 7 8 3 2 12 1 2 9 1 0]`

With this trick we can encode `[0 0 1 0 0 0 0 2 0 0 0]` as `[1 2 2 4 3]` where the last number tells us that there are 3 extra zeros at the end of the original data.

a more accurate run-length encoding: https://en.wikipedia.org/wiki/Run-length_encoding

Exercise 7

In the implementation of the functions:

function `jpeg_rle_encoder`

input:	`in_vector`	decimal sequence
output:	`rle_encoded_vector`	odd-length vector containing *skip-value* pairs + num of trailing zeros

function `jpeg_rle_decoder`

input:	`rle_encoded_vector`	odd-length vector containing *skip-value* pairs + num of trailing zeros
output:	`decoded_vector`	the reconstructed, original sequence

Run `jpeg_test_rle_encoder_decoder.m` and examine the results.

a more accurate run-length encoding: https://en.wikipedia.org/wiki/Run-length_encoding

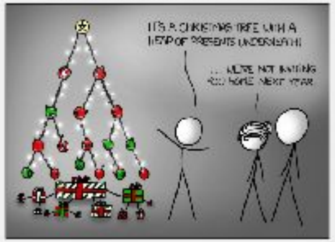
The final test

Run script `test_jpeg` and examine the results - see next slide for reference.

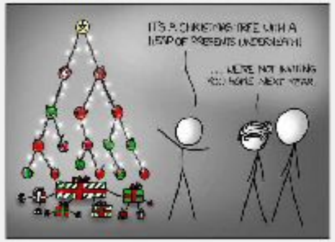
Please be advised that the DCT and IDCT implementations are quite inefficient and therefore the script may run for a longer time!

As we already discussed, the encoding-decoding pairs are embedded into each other. If you do not have some of the ‘middle-steps’ still you can run `test_jpeg` if the non-implemented functions are commented out.

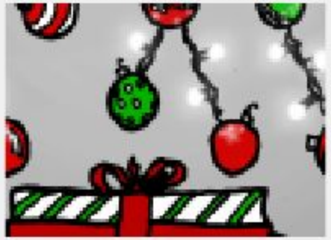
Original image
Size: 5503104 bits


IT'S A CHRISTMAS TREE WITH A HEAP OF PRESENTS UNDERNEATH!
... WE'RE NOT INVITING YOU HOME NEXT YEAR.

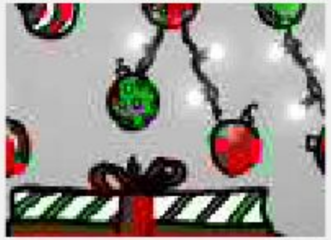
Decompressed image
Size: 5483520 bits


IT'S A CHRISTMAS TREE WITH A HEAP OF PRESENTS UNDERNEATH!
... WE'RE NOT INVITING YOU HOME NEXT YEAR.

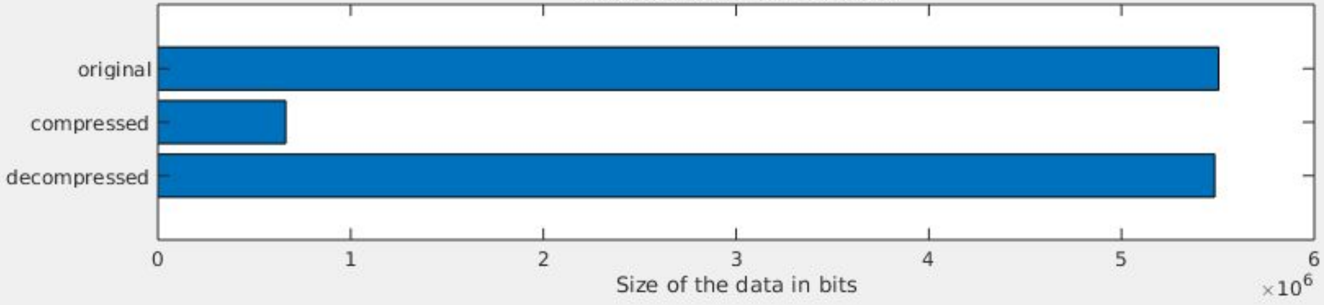
Part of the original



Part of the decompressed



Statistics
Compression ratio: 8.2968



THE END