Assignment 6

Basic Image Processing Algorithms
Fall 2016

Assignments

- assignment-points:
  - required minimum level: 50 points
  - necessary condition to Offered Final Grade: 80 points
- Assignment3:
  - will be published today,
  - 20 points,
  - deadline 23:59, 08/December/2016 (Thursday midnight)
- Assignment4:
  - will be published today,
  - 40 points,
  - deadline 18:00, 15/December/2016 (Thursday evening)
- Assignment5:
  - will be published tomorrow (Friday),
  - 10 points,
  - deadline 23:59, 11/December/2016 (Sunday midnight)
- Assignment6:
  - will be published Sunday (04/12/2016),
  - 40 points,
  - deadline 18:00, 15/December/2016 (Thursday evening)

Assignment 6: JPEG image-compression

- the maximum score of this assignment: 40 points
- deadline: **18:00, 15/December/2016 (Thursday evening)**
- please send to: bipa2016fall@gmail.com
  with the subject: ASSIGNMENT6

Transform Coding

- Encoding:

- Decoding:

JPEG

- Joint Photographic Experts Group:
  - International standard since 1991.
  - Capable of compressing continuous-tone still images (grayscale and color images) with ratio 10-50
- The algorithm:
  - Uses DCT on 8x8 blocks:
    - The blocks' grey level is shifted by -128 to the range [-128, 127].
    - The first coefficient is called DC the rest of the coefficients AC coefficient.
  - 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, using quantization tables with different step sizes for the different frequencies (in general higher step sizes for the higher frequency coefficients).

Overview and disclaimer

In this assignment we would like to realize a complete sequence of JPEG compression-decompression steps. The aim is not at implementing a specific standard strictly, rather just to have an impression how a modern transform coding sequence is built up.

It is recommended to implement every coding/processing step with its counterpart in the same time. In this way you can immediately localize any problem right after arising. The slides will also follow this scheme.

Altogether you have to realize **5x2 functions** and **1 script**, latter one calling the different steps of the encoding and decoding sequences. Please follow the naming conventions as marked in the upcoming slides.

Overview

About your script:

- please name it as **`{NEPTUN}_assignment6_script.m`**
- load the image *AlfredoBorba_TuscanLandscape.jpg* and convert it to **`double`**
- we will work on small blocks with size of 8x8, for this reason you have to crop your image to have the side-lengths as the integer multiples of 8
- please allocate space to your output image (same as the size of the cropped input)
- process the blocks from *left-to-right*, from *top-to-bottom*
- after processing all of the blocks, please show the original input and the decompressed output on the same figure

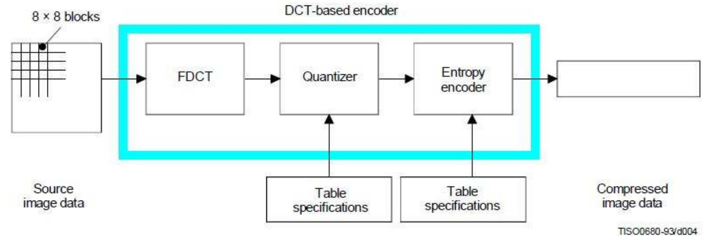


Figure 4 – DCT-based encoder simplified diagram

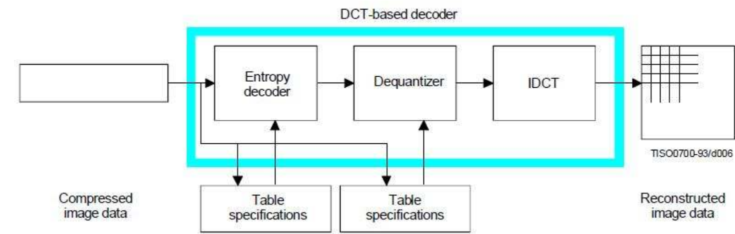


Figure 6 – DCT-based decoder simplified diagram

the Holy Grail in this topic: https://www.w3.org/Graphics/JPEG/itu-t81.pdf , images are also taken from there

Overview

About your script (continued):

- when processing one block, it should realize the following steps with the following functions:
    - `{NEPTUN}_RGB_to_YCbCr`: convert the color domain to luminosity and chrominance values
    - `{NEPTUN}_DCT`: applies Discrete Cosine Transform on the supplied channel
    - `{NEPTUN}_quantizer`: quantize the DCT-components with quantizer-tables
    - `{NEPTUN}_zigzag`: reorder the quantized elements to have long runs of zero-valued coeffs
    - `{NEPTUN}_rle_encoder`: a simple run-length encoder to make compact representation to the 0s
    - *at this point of the code, you have a condensed representation of your specific image-block*
    - `{NEPTUN}_rle_decoder`: reconstructs the long runs of zeros
    - `{NEPTUN}_izigzag`: reorders the vector to have the quantized values as a normal 2D array
    - `{NEPTUN}_dequantizer`: reconstructs the DCT-components (*of course, with quantization error*)
    - `{NEPTUN}_IDCT`: applies Inverse Discrete Cosine Transform
    - `{NEPTUN}_YCbCr_to_RGB`: convert the luminosity and chrominance values back to an RGB array
    - *place the reconstructed block in the appropriate space on the output image*

a useful overview (but we create slightly different realization): http://www.dmi.unict.it/~battiato/EI_MOBILE0708/JPEG%20(Bruna).pdf

{NEPTUN}_RGB_to_YCbCr

- please decompose your 3D **`RGB`** array to 2D arrays, in accordance with the color channels
- create the luminosity array (**`Y`**) and the two chrominance arrays (**`Cb`**, **`Cr`**) with respect to these formulas:

```
Y  =     0.299 R +  0.587 G +  0.114 B
Cb = - 0.1687 R - 0.3313 G +    0.5 B + 128
Cr =       0.5 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)
- *optionally* sample down your chrominance arrays (eg. from 8x8 to 2x2, you can do it with simple matlab-indexing applying step-size)

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

{NEPTUN}_YCbCr_to_RGB

- *if you have downsampled* your chrominance arrays, it's time to upsampling them (eg. with **`imresize`**, with **`nearest`** option)
- shift their range back with +128
- create the 3D **`RGB`** array with the following layers

```
R = Y                         +   1.402 (Cr-128)
G = Y - 0.34414 (Cb-128) - 0.71414 (Cr-128)
B = Y +   1.772 (Cb-128)
```

{NEPTUN}_DCT and {NEPTUN}_IDCT

- there are different versions with minor differences (DCT-I to DCT-IV)
- please use the following formulas (*of course, on the different channels/arrays separately*):

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

where

u is the vertical spatial frequency

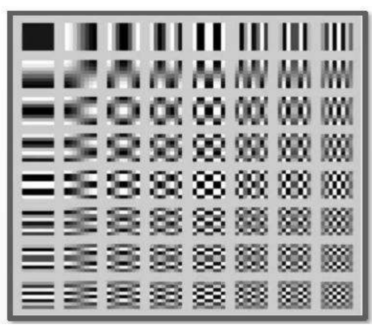
v is the horizontal spatial frequency

$$\alpha(u) = \begin{cases} \frac{1}{\sqrt{2}}, \text{ if } u = 1 \\ 1, \text{ otherwise} \end{cases} \quad \text{normalizing factor}$$

$g_{x,y}$ is the pixel value at coordinates (x, y)

$G_{u,v}$ is the DCT coefficient at coordinates (u, v)

- The 8 by 8 basis matrices for DCT:
  - In the first raw we have a cosine function with increasing horizontal frequency
  - In the first column we have a cosine function with increasing vertical frequency



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

{NEPTUN}_DCT and {NEPTUN}_IDCT

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

where

x is the pixel row

y is the pixel column

$\alpha(u)$ as defined above

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

$f_{x,y}$ is the pixel value at coordinates (x, y)

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

{NEPTUN}_quantizer and **{NEPTUN}_dequantizer**

- please use different quantization tables for the luminance DCT-coefficients and for the chrominance DCT-coefficients

```
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];
```

- *optionally*, you can use a quality-factor denoted as **`QF`** in the following

`{NEPTUN}_quantizer` and `{NEPTUN}_dequantizer`

- `quantized_values = round(original_values / (QF*quantization_values))`
  `reconstructed_values = QF*quantization_values*quantized_values`
- please be careful with the followings:
  - if you have downsampled your *chrominance channels*, you should downsample your *chrominance quantization table* as well
  - the multiplication and division should be realized *elementwise**

*Hadamard product, to mistify it a bit :) https://en.wikipedia.org/wiki/Hadamard_product_(matrices)

{NEPTUN}_zigzag and {NEPTUN}_izigzag

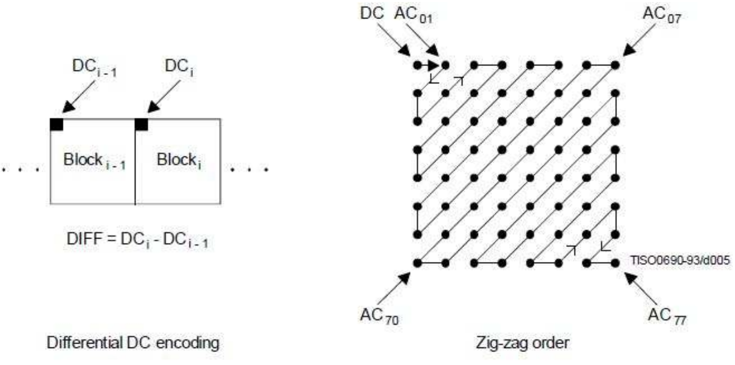


Figure 5 – Preparation of quantized coefficients for entropy encoding

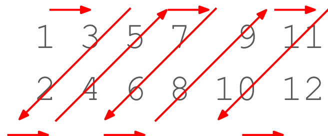
- in our case, we are not going to deal with the DC-components separately, we will just leave them at the beginning of the zig-zag scan

initial step: how to create an index-array with columnwise indexing an existing data-array?

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


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

{NEPTUN}_zigzag and {NEPTUN}_izigzag



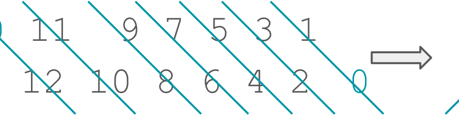
1,3,2,4,5,7,6,8,9,11,10,12 the zig-zag sequence of the indices

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

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



1 3 5 7 9 11 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

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

`{NEPTUN}_zigzag` and `{NEPTUN}_izigzag`

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          0 11  9 7 5 3 1          1 3 5 7  9 11 0
2 4 6 8 10 12   ⇒       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 array again (after the **`fliplr`** - **`spdiags`** - **`fliplr`** tiplet):

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

as you see, you have to flip every odd columns with respect to a horizontal line:

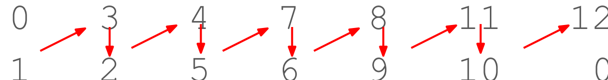
```
0  3  4  7  8  11  12
1  2  5  6  9  10   0
```

```
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

{NEPTUN}_zigzag and {NEPTUN}_izigzag



and this is now the standard MATLAB columnwise indexing!

The only thing is left: 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=arr(idxs);
```

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

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

In your *izigzag* function,

- either you should assume you have only squared blocks (even when downsampling in the chrominance layers, so `desired_size=[sqrt(length(zig_zag_vector)), sqrt(length(zig_zag_vector))];`)
- or you should receive this **`desired_size`** parameter as the size of the arrays before '*zigzaging*'

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

`{NEPTUN}_rle_encoder` and `{NEPTUN}_rle_decoder`

We will implement a modified version of the *run-length encoding*.
As you remember: 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).

How to code these long runs of zeros? With *{'skip', 'value'}* pairs:

- *'value'* is the next non-zero element, and
- *'skip'* means the number of zeros before it.

You have to fix the case when the last element is zero - let's code it as a *{1,0}*.
You can concatenate these pairs into one long row vector.

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

Closing remarks

Although we are not going to generate valid JPEG-files (the appropriately ordered byte-sequence with the necessary header-information), it is still worth to see how the main steps builds up.

You can examine the efficiency of your code-sequence: sum up the lengths of *rle-encoded* data layers in every block, then compare it to the total number of data points in the image (*width x height x channel_number*) .