Unit III: Image Compression

Signal Processors Lab

Markus Quaritsch
markus.quaritsch@tugraz.at

SS 2017
Insitute for Technical Informatics

The topic of the third unit of the DSP lab is image compression using the TMS320C6713 DSP. The goal is to develop and implement a codec (coder/decoder) for a compression chain similar to JPEG using Code Composer Studio, C programming language and the DSK as evaluation environment.

A project framework which implements skeleton functions and some ready-to-use modules is provided. An overview of the compression chain as well as implementation details and evaluation results of the processing steps can be found in the appendix.

1. Overview

Figure 1 gives an overview of the processing steps of JPEG, the same chain as you have to implement in this lab unit. For more details consult the extra tasksheet.

Figure 2 shows how the compression chain is implemented by the project framework. It names memory regions, global variable names and functions involved. Have a look on it to get an idea about the dataflow of the framework.

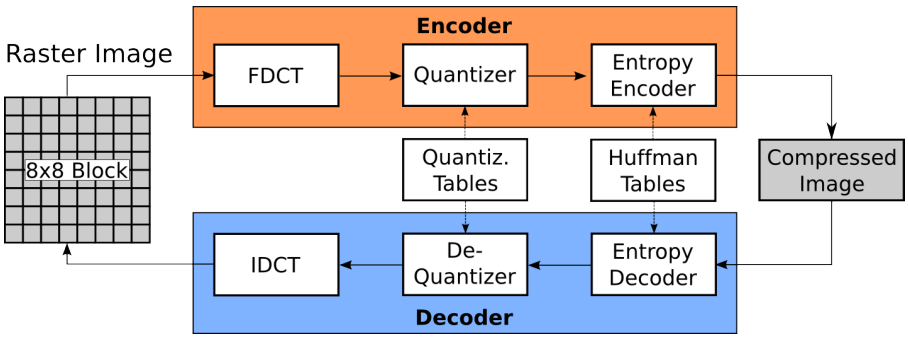


Figure 1: Overview of the compression chain which is implemented in this task.

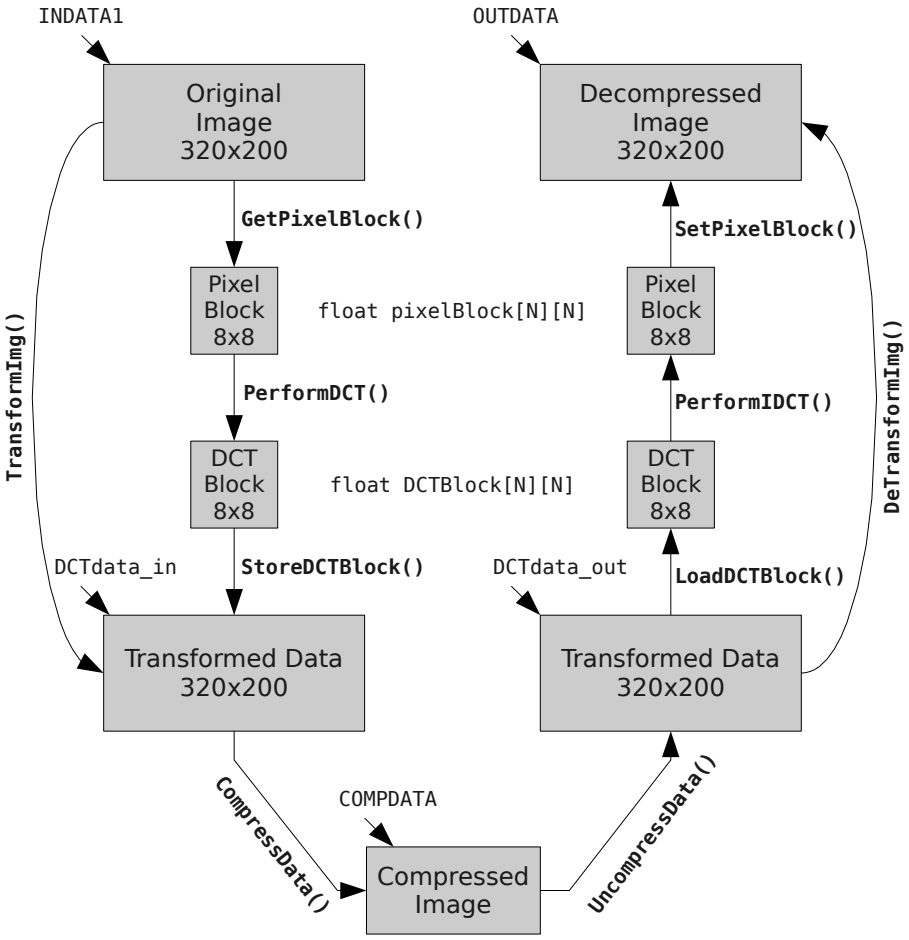


Figure 2: Overview of the implementation of the compression chain.

Because the DSK lacks support for a video sensor and to obtain repeatable and comparable results the data has to be copied into the DSP memory for algorithm evaluation. An image, which is stored in a file on the workstation, is downloaded to the algorithm's input buffer (e.g., address `0x80000000` in SDRAM) via Code Composer Studio. The JPEG compression (i.e., `TransformImg()`) is applied on this data and the result is written to a different memory region (e.g., at address `COMPDATA` in SDRAM). The decompression works in the same way: the compressed image is decompressed via `DeTransformImg()` and copied into `OUTDATA`. To check the final result of the compression/decompression chain, the output image is transferred to the workstation and displayed using the viewer embedded in Code Composer Studio.

CCS integrates some functionality to support this evaluation process, and also integrates a scripting language (General Extension Language, GEL) which allows automatization of complex processes in the environment.

$startaddr$	$pixel_{1,1}$
$startaddr + 1$	$pixel_{1,2}$
$\dots$	
$startaddr + width$	$pixel_{2,1}$
$\dots$	
$startaddr + (height * width)$	$pixel_{height,width}$

Table 1: Mapping between pixels and memory addresses.

Download Images to the DSP: A GEL-script is provided to automate the downloading process to the DSP. It can be found in *gel/images.gel*. To load the gel-file, start the Debugger, click *Tools→GelFiles*, and load the file with a right-click on the appearing GEL-Dialog. When successfully loaded, it extends the user interface with functions in the menu *Scripts*.

There are 7 different image files provided, 5 containing objects, one containing noise, one containing black pixels only (for cleaning the memory) and one containing white pixels only. Use the functions from the Scripts menu to download images to the appropriate memory region.

Three different memory regions of size 64k are reserved in SDRAM for storing images: Two input memory regions starting at `0x80000000` and `0x80010000` and one output memory region starting at `0x80020000`, respectively. The preprocessor macros `INDATA1`, `INDATA2` and `OUTDATA` make the addresses available in C; use a cast to `(unsigned char *)` to get a pointer.

Access Image Data: The images downloaded to SDRAM are directly addressable by the image processing functions. All images are 320 by 200 in size, and 8-bit in color resolution (luminance channel only, no compression). Each of the 64000 pixels is stored in a single byte, line-by-line, starting from top-left. When accessing all pixels in sequence, a pointer to the starting address and the increment function can be used to iterate over all pixels. When addressing a single pixel, the location (x, y) of the pixel has to be transformed into the corresponding location in memory. Table 1 illustrates the mapping between pixels and memory addresses.

Display Images: CCS integrates an image viewer, which can display images directly from DSP memory. The viewer is available via GUI at *View→Other...* and then select *Image* under *Analysis Views*. In the new view you need to set the properties of the image to show (i.e., start address, resolution, etc.). The image is automatically refreshed at a breakpoint. Improved control over image viewer updates can be achieved by using *Probe Points*[1].

Suggested Implementation Approach

Implementing the whole JPEG compression and decompression chain at once is a big step and thus error prone. Thus the following tasks give you a guidance to break down the processing and implement the compression/decompression chain in smaller steps. The following description of the task is just to give an overview of the following tasks, more details are given in the next section.

[1] As downloading an image from the DSK to CCS via JTAG takes some time, it is wise to update only when the final result is available

In Task 1 you implement the first step of the compression/decompression. `TransformImg` simply copies blocks of 8×8 pixels from the input image into the processing buffer `PixelBlock[][]` and `DeTransformImg()` copies the processing buffer `PixelBlock[][]` to the output image. To test your implementation you need to modify the `main()` function and use `GetPixelBlock()` and `SetPixelBlock()` to copy the input image to the output buffer, i.e. "shortcut" the whole compression/decompression.

In Task 2 you implements the discrete cosine transformation and inverse cosine transformation. For the transformation, the block of 8×8 pixels from `PixelBlock[][]` is transformed and stored into `DCTBlock[][]` while for the inverse transformation data from `DCTBlock[][]` is de-tranformed and stored into `PixelBlock[][]`. Include both, the transformation and in inverse transformation (i.e., `PerformDCT()` and `PerformIDCT()`) in your processing chain.

In Task 3 the actual lossy compression/de-compression is done by quantizing/de-quantizing the transformed block of pixels in `DCTBlock[][]`. For the transformation the quantized data is stored into the `DCTdata_in` buffer, for the de-tranformation the data is loaded from `DCTdata_out`. Use the global variable `dctdataoff` to store the current position in `DCTdata_in`/`DCTdata_out`. The `DCTdata_in` buffer is still the same size as the input image. As a last step Huffman encoding/decoding is applied to reduce the required storage capacity. The according functions are provided in the skeleton. As a final step, chain all compression steps and all de-compression steps together in `TransfomrImg()` and `DeTransformImg()`.

2. Tasks

Task 1 – Basic functionality: In the first task you should implement helper functions for reading and writing blocks of pixels.

(a) Implement `GetPixelBlock()`: This function copies a block of data, given by a pointer to the image area and the coordinates of the upper-left pixel within this area, to the global pixel buffer `PixelBuffer[][]`. The dimension of the block is given by the preprocessor define *N*. Avoid multiplications!

(b) Implement `SetPixelBlock()`: This function is the inverse of `GetPixelBlock()`. It copies the pixel block stored in `PixelBuffer[][]` to an image, which is given by a pointer to the image area and the coordinates of the upper-left pixel within this area. Avoid multiplications!

(c) To test both functions, implement a testfunction for copying `INDATA1` to `OUTDATA` by using `GetPixelBlock()` and `SetPixelBlock()` only.

Task 2 – Discrete Cosine Transformation FDCT/IDCT: In this task the cosine transformation (DCT) and its inverse (IDCT) must be implemented. Information about the transformation, the equation and some results for testing can be found in the extra tasksheet.

(a) Implement `PerformDCT()`: This functions performs the DCT. It takes the pixel data from the global pixel buffer `PixelBlock[][]`, transforms the pixel block to frequency domain and stores the result to the global DCT buffer `DCTBlock[][]`.

(b) Implement `PerformIDCT()`: This function performs the inverse DCT. It takes the DCT transformed block from the global buffer `DCTBlock[][]`, transforms the block into spacial domain and stores the result in the global pixel buffer `PixelBlock[][]`.

- Implement both functions in the file `dct.c`.
- Take care of possible rounding errors in the IDCT (avoid them by using clipping).
- Check the results using the memory view of CCS. Furthermore call the IDCT on the results of the DCT, and check whether the final result is the same as the origin block.

Task 3 – Quantization and Huffman-Coder: : In this task quantization and entropy coding of the compression chain are implemented.

(a) Implement `StoreDCTBlock()`: This function quantizes the DCT-transformed data block (`DCTBlock[][]`), does a saturated conversion to `signed char` (except the DC value, which is an `unsigned char`) and stores the result to linear output memory.

(b) Implement `LoadDCTBlock()`: This function, which is the inverse of the previous, loads a block of quantized data from the linear input memory, requantizes the values and stores them to the global DCT block buffer (`DCTBlock[][]`).

(c) Implement `TransformImg()`: This function controls the transformation process of the total image. It loads each block of the input image from `INDATA1` using `GetPixelBlock()`, transformates the blocks using `PerformDCT()`, and stores the quantized blocks to the DCT image buffer `DCTdata_out` using `StoreDCTBlock()`.

(d) Implement `DeTransformImg()`: This function is the inverse from the previous. It uses `DCTdata_in` as input memory and `OUTDATA` as output memory.

(e) Implement `CompressData()` and `UncompressData()`: These functions implement the compression and the uncompression using the available adaptive huffman coder. The according huffman functions to call are `ahuff_CompressBlock()` and `ahuff_ExpandBlock()`. The memory area at `COMPDATA` is used for storing the compressed data.

(f) Test the proper function of your implementation. The compression and uncompression chain should now be fully working. Do performance analysis using compiler optimization.

Task 4 – Quality and Compression Rate: After finalizing the implementation of the compression chain in task 3, the results should be evaluated in regarding compression rate and image quality.

(a) Implement `CalcErrorRMS()`. This function returns the root mean square (RMS) of the original image at `INDATA1` and the decompressed image at `OUTDATA`. The RMS value is an indicator for the image quality.

(b) The size of the compressed image is returned by the huffman coder. Use this value for calculating the compression rate.

(c) Analyse the compression rate and the RMS using different quality levels (the quality can is set by the preprocessor define `picquality`).

Task 5* – Runtime Improvements: The runtime of the DCT can be improved by using lookup tables (LUTs) for complex calculations.

(a) Initialize LUTs: The function `InitTables()` initializes the quantization table only. Extend this function to initialize the LUTs also for the cosines and for the coefficients.

(b) Implement DCT and IDCT functions which make use of the LUTs (`PerformDCT_opt()` and `PerformIDCT_opt()`).

(c) Measure the runtime (without `InitTables()`) and compare them with the prior implementation (speed-up).

Task 6* – Improving Image Quality: To reduce the blocking effect of the block-based DCT, the image can be smoothed using a lowpass filter after the decompression step. Use the box-filter of unit 3 to fullfill this step. Evalute the results by using the RMS function.

3. Questionaire

Generally

(1) Which step implements the compression?

(2) What is the lossy step of this compression method? Why?

(3) Which properties of the compression chain are influenced by the values in the quantization table? In which way?

Task 2

(1) Runtime analysis using compiler optimization (-O3):

PerformDCT(): ..... cycles PerformIDCT(): ..... cycles

Task 3

(1) What is the largest possible DC value after performing the DCT (before quantization)? What is the size of an according quantization value to avoid overflow?

Task 4

(1) Give the following evaluation results using the tiger image:

Quality = 8 ErrorRMS = ... Compression Rate = ...

Quality = 16 ErrorRMS = ... Compression Rate = ...

Quality = 32 ErrorRMS = ... Compression Rate = ...

(2) Runtime analysis using compiler optimization (-O3):

CompressData(): ..... cycles UncompressData(): ..... cycles

Task 5*

(1) Runtime analysis using compiler optimization (-O3):

PerformDCT_opt(): ..... cycles PerformIDCT_opt(): ..... cycles

Task 6*

(1) Give the following evaluation results using the tiger image:

Quality = 16 ErrorRMS = ...

Quality = 32 ErrorRMS = ...

A. Appendix

Introduction

The resolution of typical digital images increased over the last few years. For example, an RGB image with XGA resolution and 24-bit color depth needs about 2.25 MB of memory (1024x768x3Byte). The high memory usage drives researchers and developers to investigate methods for reducing the amount of data needed to store images. Compression methods, which are based on redundancy in the image data, can be divided into lossless and lossy methods. The difference in general is, that lossy compression methods remove information from the image (e.g., for web applications) while lossless methods do not remove any information (e.g., for archival storage).

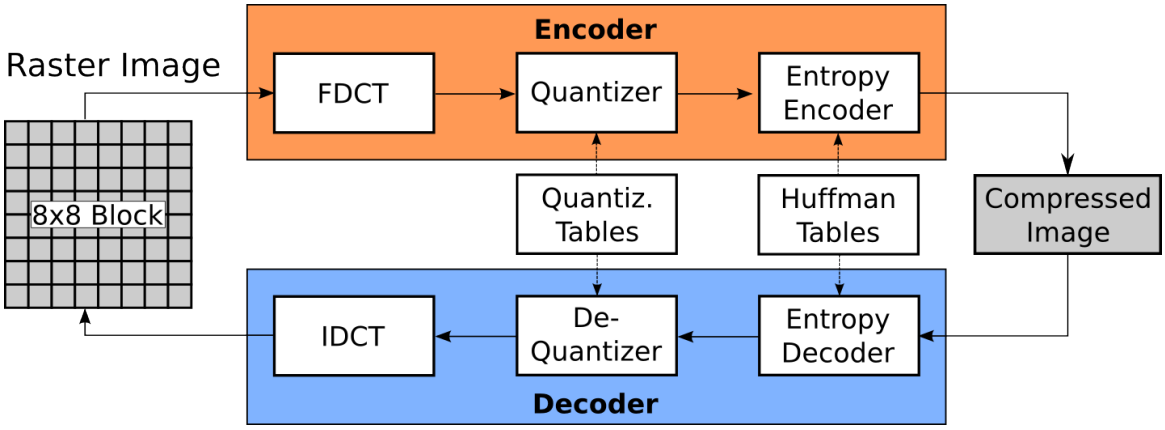
Usually, lossy compression methods consist of two stages. In the first stage the least important information is removed from the image (lossy step). The amount of information removed influences the quality of the compressed image and the compressiong rate. In the second stage the remaining information is compressed lossless using e.g., Huffman Coder or Arithmetic Coder.

The most common image compression standard is JPEG (Joint Picture Expert Group). The first research activities began in the late 80's and became more and more popular with the increasing processing power of state-of-the-art CPUs. JPEG defines lossy and lossless processes, the lossy method is the more prominent (higher compression rates).

In this document, essential steps of the lossy JPEG method are explained and a guideline for implementation are demonstrated. Details about the standard can be found online.

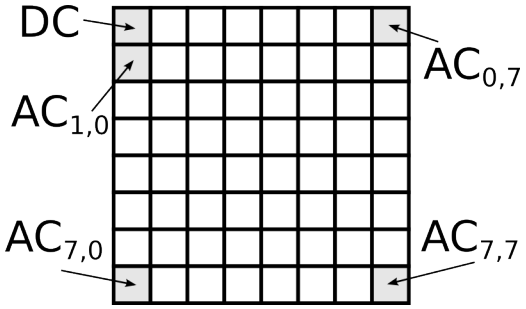
Details about JPEG

JPEG, as we use it in this lab, is a lossy compression standard. This means, that information is lost when compressing with the consequence that the original image cannot be restored after compression. The following figure gives an overview of the compression/decompression chain, followed by a description of each step.



Raster Image: The raster image is the original, uncompressed image which is seperated into 8x8 blocks. This seperation is needed for speeding up the complex 2-dimensional DCT of runtime $O(n^4)$.

Forward Discrete Cosine Transformation (FDCT): The FDCT transforms an 8x8 block of image data from the spatial domain into the frequency domain. The result is an 8x8 block containing the spectral components of the original block, with lower frequencies at the top-left, and higher frequencies at the bottom-right. By applying the transformation on the block, no information is lost, except potential rounding errors.



Quantizer: In the quantizer step a block of data in frequency domain is quantized using a predefined quantization table. The quantization table is defined in a way that lower frequencies, which are of more importance for the image quality, are quantized with small values, and high-frequency components with large values. The result of the quantization is that high frequency components become zero – redundancy which leads to high compression rates using an entropy coder. Therefore the quantization table defines the achievable compression rate as well as the quality of the compressed image.

Entropy Coder: The entropy coder compresses the the data by finding redundant information, e.g. by using a dictionary for recurrent code sequences or by using short codes for most frequent values. For JPEG, an adaptive Huffman coder is used.

Discrete Cosine Transformation (DCT)

The *Discrete Cosine Transformation* was discovered in 1974 as a fast alternative to the discrete fourier transformation. As the runtime of the DCT is of $O(n^4)$, applying the transformation on large images would lead to huge runtimes. To avoid this, the image is divided into blocks of pixels (for JPEG 8x8), which are processed separately.

$$\textbf{DCT:} \quad z(k,l) = \frac{2}{N}\alpha(k)\alpha(l)\sum_{m=0}^{N-1}\sum_{n=0}^{N-1} x(m,n)cos\left(\frac{\pi(2m+1)k}{2N}\right)cos\left(\frac{\pi(2n+1)l}{2N}\right)$$

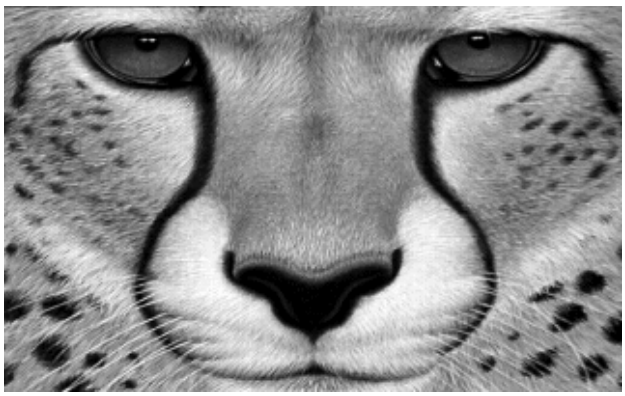
$$\textbf{IDCT:} \quad x(m,n) = \frac{2}{N}\sum_{k=0}^{N-1}\sum_{l=0}^{N-1} \alpha(k)\alpha(l)z(k,l)cos\left(\frac{\pi(2m+1)k}{2N}\right)cos\left(\frac{\pi(2n+1)l}{2N}\right)$$

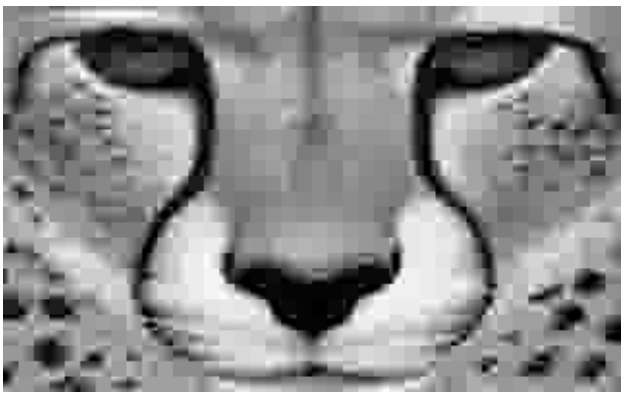
$$\alpha(k) = \begin{cases} \frac{1}{\sqrt{2}} & |k=0 \\ 1 & |k\neq 0 \end{cases} \qquad \alpha(l) = \begin{cases} \frac{1}{\sqrt{2}} & |l=0 \\ 1 & |l\neq 0 \end{cases}$$

$z(0,0)\ldots$DC component of the block in frequency domain

$x(0,0)\ldots$pixel at position (0/0) in the spatial domain

But the processing based on blocks leads also to some problems. When using high quantization values, edges between blocks could get visible in the decompressed image. The reason for that is that neighboring pixels which belong to different blocks loose their correlation information. The influence of this blocking effect can be reduced by applying a lowpass filter after decompression. The following images show the blocking effect:





Example for DCT and Quantization

The following tables shows an example for the DCT and the quantization by using the top-left 8x8 block of the tiger image.

NOTE: The tables are only 4 columns wide, so each line of the 8x8 block corresponds to two lines in each table.

First pixelblock (8x8) of the tiger image:

PixelBlock

95.0	88.0	88.0	87.0
95.0	88.0	95.0	95.0
143.0	144.0	151.0	151.0
153.0	170.0	183.0	181.0
153.0	151.0	162.0	166.0
162.0	151.0	126.0	117.0
143.0	144.0	133.0	130.0
143.0	153.0	159.0	175.0
123.0	112.0	116.0	130.0
143.0	147.0	162.0	183.0
133.0	151.0	162.0	166.0
170.0	188.0	166.0	128.0
160.0	168.0	166.0	159.0
135.0	101.0	93.0	98.0
154.0	155.0	153.0	144.0
126.0	106.0	118.0	133.0

Quantization matrix (quality=8):

QuantT

8.0	16.0	24.0	32.0
40.0	48.0	56.0	64.0
16.0	24.0	32.0	40.0
48.0	56.0	64.0	72.0
24.0	32.0	40.0	48.0
56.0	64.0	72.0	80.0
32.0	40.0	48.0	56.0
64.0	72.0	80.0	88.0
40.0	48.0	56.0	64.0
72.0	80.0	88.0	96.0
48.0	56.0	64.0	72.0
80.0	88.0	96.0	104.0
56.0	64.0	72.0	80.0
88.0	96.0	104.0	112.0
64.0	72.0	80.0	88.0
96.0	104.0	112.0	120.0

Block after DCT but before quantization:

DCTBlock

1115.5	3.435298	-8.865819	-6.531707
2.999798	-0.6832078	0.1543182	1.641178
-38.51787	-57.54152	11.68813	17.25033
-2.266859	2.023286	4.133494	2.253073
-84.23492	62.16578	0.9393768	-18.13004
3.484955	3.854451	-5.006079	4.655408
-52.48856	-36.17615	-10.46086	13.78166
-9.841528	3.560306	-2.018753	0.4507229
-86.50018	-40.2334	49.13938	-7.396041
17.49998	-6.105496	-2.224156	4.708854
-62.35625	64.91502	-11.84682	-1.56534
3.26086	-8.229655	-1.861565	0.3905859
-16.52263	14.07991	-36.0061	17.18448
-11.18503	2.882492	3.060687	-1.151351
-53.86305	32.12744	-8.615198	-8.590064
22.30747	-0.1154237	0.8724148	2.989478

DCT Block after requantization:

DCTBlock

1112.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
-32.0	-48.0	0.0	0.0
0.0	0.0	0.0	0.0
-96.0	64.0	0.0	0.0
0.0	0.0	0.0	0.0
-64.0	-40.0	0.0	0.0
0.0	0.0	0.0	0.0
-80.0	-48.0	56.0	0.0
0.0	0.0	0.0	0.0
-48.0	56.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	-72.0	0.0
0.0	0.0	0.0	0.0
-64.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

Pixel block after IDCT (compression rate 80%):

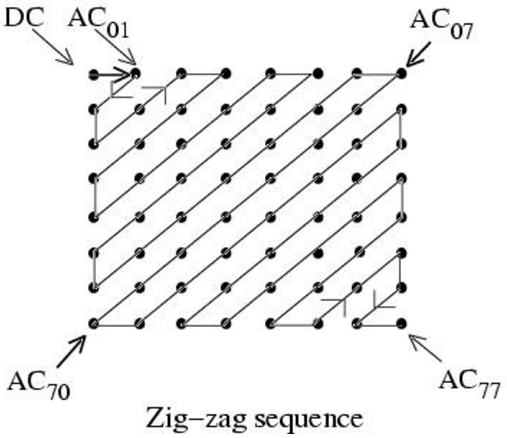
PixelBlock

88.0	88.0	87.0	87.0
90.0	94.0	98.0	100.0
154.0	151.0	148.0	147.0
150.0	156.0	163.0	168.0
136.0	149.0	167.0	179.0
175.0	158.0	135.0	120.0
161.0	153.0	143.0	137.0
140.0	152.0	167.0	178.0
114.0	110.0	108.0	113.0
127.0	150.0	174.0	188.0
131.0	146.0	167.0	182.0
184.0	171.0	151.0	138.0
172.0	163.0	148.0	131.0
117.0	108.0	103.0	101.0
156.0	152.0	144.0	136.0
129.0	124.0	121.0	120.0

Entropy Coding

Zig Zag Sequence

The values of the DCT matrix are read in zig-zag order, which means that components with the same quantization are read consecutively, e.g., $(0, 1)$ and $(1, 0)$. This zig-zag order leads to many consecutive zeros in the output, which can be very easily compressed by using runlength encoding.



Zig–zag sequence

Adaptive Huffman Coding

After the transformation and the lossy quantization, image data is compressed using an entropy coder. A widely used method is the *adaptive huffman coder*. The idea is that symbols which appear very frequently are coded with fewer bits than symbols which appear rather infrequently. The special thing on the adaptive huffman is, that the coding table is generated dynamically (single pass).

Functions for adaptive huffman coding and decoding are already implemented for the lab, and are ready for being used. More information about the adaptive huffman can be found in the web.

Weblinks

JPEG	http://www.cs.cf.ac.uk/Dave/Multimedia/node234.html
DCT	http://www.cs.cf.ac.uk/Dave/Multimedia/node231.html http://en.wikipedia.org/wiki/Discrete_cosine_transform http://www.icaen.uiowa.edu/ dip/LECTURE/LinTransforms.html
Adaptive Huffman	http://www.cs.duke.edu/csed/curious/compression/adaptivehuff.html http://www.cs.cf.ac.uk/Dave/Multimedia/node212.html