# LAB 4: JPEG Compression

Harald Nautsch, Maria Magnusson, uppdaterad av Michael Felsberg
Avdelningen för Datorseende, Institutionen för Systemteknik,
Linköpings Universitet

March 2017

## 1 Introduction

### 1.1 Overview

A transform coder consists of three distinct parts: The *transform*, the *quantizer* and the *variable length coder*. In this laboratory work you will study all three parts and see how the choice of transform/quantizer/variable length coder affects the performance of the transform coder. Sections marked as *preparation tasks* (2, 3, 5) must be solved before the laboratory session starts.

The laboratory work runs in a PYTHON environment. In PYTHON images are naturally represented as matrices. During the laboratory work certain test images will be compressed. The test images can be thought of as *original images* in the sense that they are stored as raw samples with $3 \times 8 = 24$ bits/pixel. This usually gives more colors than the human eye can discern on a computer screen.

### 1.2 A note on data rate

For images, we usually measure the data rate in bits/pixels. This assumes that we want to view the image at a certain resolution, typically the original resolution of the image. For example, if the original image is $512 \times 512$ pixels, and we use 50000 bits to store it, the data rate is $50000/512^2 \approx 0.19$ bits per pixel.

## 1.3 A note on distortion

We will measure the degradation of the image quality, the distortion, by the simple *mean square error*. In PYTHON , if we have the original image in the variable `orig` and a distorted image in the variable `coded`, the mean square error can be calculated as

```
mse = np.mean((orig-coded)**2)
```

Normally in image coding, the distortion is given as the *peak signal to noise ratio*, PSNR, which is defined by

```
psnr = 10*np.log10(255**2/mse)
```

where 255 is the maximum difference in grey-level for an image stored with 8 bits per pixel.

If you want, you can have `psnr=35dB` and `psnr=39dB` as reference values. `psnr=35dB` gives a "half-good" image and and `psnr=39dB` gives a "good" image. In the following, you are also going to give a subjective estimation of the image quality. Always check if your subjective estimation agrees with the PSNR value.

## 1.4 Starting PYTHON

Copy all the files on

```
/site/edu/bb/SigInfBild/JPEGLab/
```

to a suitable place at your own directory and start PYTHON :

```
> module add optpython
> ipython
```

Similar as in Lab 3, we import a number of packages

```
import numpy as np
from scipy import signal, misc
from matplotlib import pyplot as plt
plt.rcParams['image.interpolation'] = 'nearest'
import jpeglab as jl
```

preferably saved to a file `preambel.py` that is loaded by executing `execfile('preambel.py')`. In addition to Lab 3, we import some helper functions in the last line.

## 2 *Preparation task*: Color image manipulation in PYTHON

We will be working with both color and gray-scale images. An image can be read into PYTHON using the command `imread`.

```
im1 = misc.imread('image1.png')
```

The color image is stored as a $512 \times 768 \times 3$ matrix, meaning that we have one $512 \times 768$ matrix for each of the three RGB color components (red, green and blue). The image can be viewed using the command `imshow`:

```
plt.imshow(im1), plt.show()
```

We can also view each color plane separately:

```
im1r = im1[:,:,0]
im1g = im1[:,:,1]
im1b = im1[:,:,2]
plt.figure(1), plt.imshow(im1r,'gray')
plt.figure(2), plt.imshow(im1g,'gray')
plt.figure(3), plt.imshow(im1b,'gray')
plt.show()
```

**QUESTION 1**: Try to become more familiar with the color components. Red corresponds to high value in the R-image and low value in the G- and B-image. What about green, blue, yellow, cyan (turquoise), magenta (pink), white and black?

---

When coding color images, we usually use the YCbCr color space rather than the RGB color space. To convert the image back and forth between the different color spaces, use the functions `jl.rgb2ycbcr` and `jl.ycbcr2rgb`:

```
y, cb, cr = jl.rgb2ycbcr(im1)
```

The luminance component is basically a grey-scale version of the color image, while the two chrominance components contain information about the color of the image. See what the luminance and chrominance components look like:

```
plt.figure(2), plt.imshow(y, 'gray', clim=(0, 255))
plt.figure(3), plt.imshow(cb, 'gray')
plt.figure(4), plt.imshow(cr, 'gray')
plt.show()
```

**QUESTION 2**: In which image can you see most details? We will exploit the different information contents later.

---

In the YCbCr color space, luminance information is represented by a single component, Y, and color information is stored as two color-difference components, Cb and Cr. Component Cb is the difference between the blue component and a reference value and component Cr is the difference between the red component and a reference value. The transformation used to convert from RGB to YCbCr is

$$\begin{pmatrix} Y \\ Cb \\ Cr \end{pmatrix} = \left( \begin{pmatrix} 16 \\ 128 \\ 128 \end{pmatrix} + \begin{pmatrix} 65.481 & 128.553 & 24.966 \\ -37.797 & -74.203 & 112.000 \\ 112.000 & -93.786 & -18.214 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} \right) /255$$

There are six images (named `image1.png` to `image6.png`) with varying content that you can use for your experiments. Since experiments in image processing should never be restricted to a single image, repeat your analysis of question 2 for the other five images, too. In the following, we will mostly concentrate on one of the three components, namely the luminance component, Y, which is quite similar to a grey-scale image.

Write PYTHON code for computing the mean-square error of two images and the PSNR.

## 3 *Preparation task*: Reducing the number of grey-levels

We will now look at a simple data reduction method: Reducing the number of grey-levels in the image (i.e. quantization). To reduce the number of grey-levels from 256 to 128, i.e. going from 8 bpp (bits per pixel) to 7 bpp do the following:

```
plt.figure(2), plt.imshow(y, 'gray', clim=(0, 255))
y2 = 2*np.floor_divide(y,2)
plt.figure(3), plt.imshow(y2, 'gray', clim=(0, 255))
plt.show()
```

Note that a pixel here contains 8 bits, which can give 256 different grey-scale levels. One example is

$\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1} = 255$

since $1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 255$.

Another example is

$\boxed{0}\boxed{0}\boxed{0}\boxed{1}\boxed{0}\boxed{0}\boxed{0}\boxed{0} = 16$

since $1 \cdot 2^4 = 16$.

In the y2-image, the last bit is always 0 and is not necessary to code, giving 7 bpp.

**QUESTION 3**: To which number do you change `X` in `y2=X*np.floor_divide(y,X)` if you want only 5 bpp?

---

**QUESTION 4**: Now write `psnr`, `bpp`, and subjective image quality assessment for at least two choices of parameters, one “half-good” and one “good”! Repeat the assessment for other image examples of “half-good” and “good”.

---

## 4 Introductory experiment: Down-sampling the image

Down-sampling can be done by using the function `misc.imresize`. To down-sample (and then up-sample) the image by a factor `2.` (note: `2` would mean 2%) in each dimension, do:

```
y3 = np.floor(misc.imresize(y, 0.5, interp='bicubic', mode='F'))
y4 = misc.imresize(y3, 2., interp='bicubic', mode='F')
plt.figure(3), plt.imshow(y4, 'gray', clim=(0, 255)), plt.show()
```

The down-sampling can be considered as the encoding of the image and the up-sampling as the decoding. The down-sampled image in the example has only one quarter of the number of pixels, compared to the original image. Each pixel still needs to be stored with 8 bits. This means that we essentially have gone from 8 bpp to 2 bpp ($256^2 \cdot 8/512^2 = 2$).

**QUESTION 5**: Now write `psnr`, `bpp`, and subjective image quality assessment for at least two choices of parameters, one “half-good” and one “good”! Repeat the assessment for other image examples of “half-good” and “good”.

---

**QUESTION 6**: Compute the absolute difference image between `y` and `y4` and visualize it. Where in the image are the largest errors located?

---

## 5 *Preparation task*: Transforming the whole image

Now we will try a simple transform coding method. First we transform the whole image using a two-dimensional discrete cosine transform

```
import cv2
Y = cv2.dct(y)
plt.figure(3), plt.imshow(np.log(np.abs(Y)+1),'gray')
plt.show()
```

We throw away most of the high frequency coefficients and round the remaining ones to integers, then do inverse transformation

```
Yq = np.zeros((512,768))
Yq[0:128,0:196] = np.round(Y[0:128,0:196])
plt.figure(4), plt.imshow(np.log(np.abs(Yq)+1),'gray')
yq = cv2.idct(Yq)
plt.figure(5), plt.imshow(yq,'gray',clim=(0,255))
plt.show()
```

In this example we kept 1/16 of the coefficients ($128^2/512^2$) and we need to use 17 bits per coefficient. This means that we achieve a data rate of 1.06 bpp.

**QUESTION 7**: Check the maximum and minimum value of `Yq`. Why do we need to use 17 bits per coefficient?

---

**QUESTION 8**: Now write `psnr`, `bpp`, and subjective image quality assessment for at least two choices of parameters, one "half-good" and one "good"! Repeat the assessment for other image examples of "half-good" and "good".

---

The two-dimensional discrete cosine transform (cv2.dct) is a close relative to the two-dimensional discrete fourier transform (implemented as np.fft.fft2). The program `costrans3.py` shows how they are related. Run the program, look at the images and the code. The code is given in the end of this paper. Then, answer the following two questions. If you need, discuss the task with the laboratory assistant.

**QUESTION 9**: How are dct and fft2 related?

---

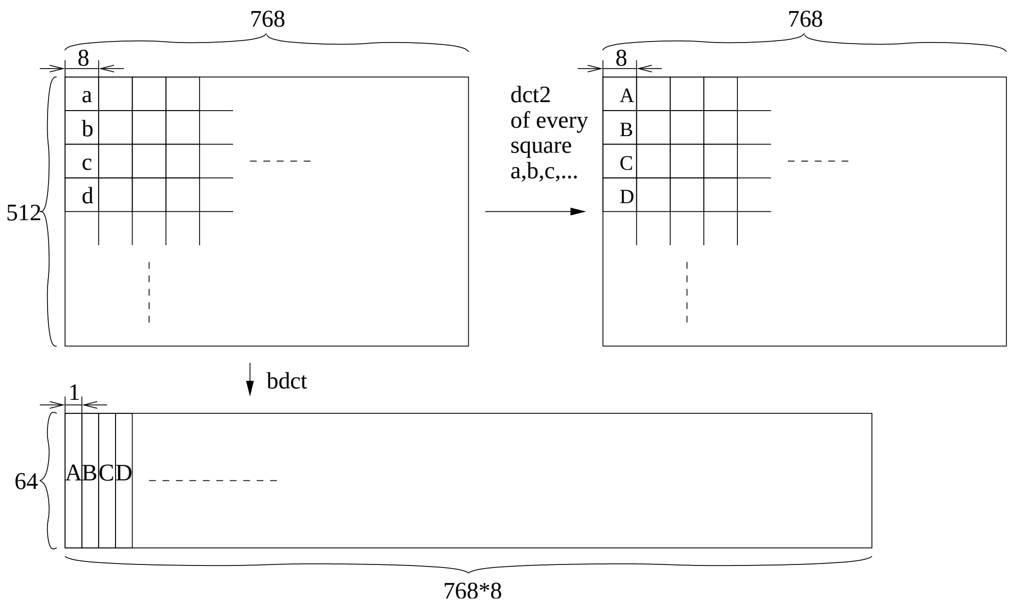
**QUESTION 10**: Why is it preferable to use the dct compared to the fft2 in image coding?

---

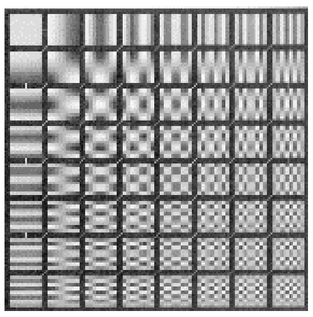
## 6 Block-based transforms

Normally in image coding, the whole image is not transformed with a single transform. Rather, the image is divided into small blocks (typically $8 \times 8$ pixels) that are transformed separately.

Usually, the transform components are arranged in a block of the same size as the image block. The components are sorted in increasing frequency order, with the DC component in the upper left corner. The transform function in this lab (`jl.bdct`) arrange the components of the transformed block into column vectors instead, to make it easier to do quantization and *variable length coding.* The task is explained in the figure below.



The reshaped image is then transformed using the DCT basis functions for $8 \times 8$ signals, depicted below.



Example:

```
plt.figure(2), plt.imshow(y, 'gray', clim=(0, 255))
Yb = jl.bdct(y, (8, 8))
ulim = np.max(np.abs(Yb))/10
plt.figure(3), plt.imshow(np.abs(Yb), 'gray', clim=(0, ulim))
plt.show()
```

The inverse transformation is performed with the function `jl.ibdct`:

```
yn = jl.ibdct(Yb, (8, 8), (512, 768))
plt.figure(4), plt.imshow(yn, 'gray', clim=(0, 255)), plt.show()
```

Without quantization, `y` and `yn` should be identical. Due to rounding errors in the computer, there might be a slight (but negligible) difference. Check this:

```
np.max(np.abs(y-yn))
```

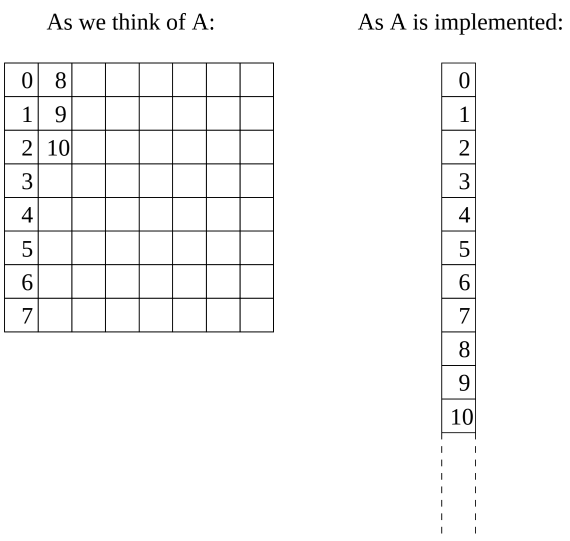
**QUESTION 11**: How large is the maximal difference?

---

A simple coding method would be to just throw away the high frequency coefficients in each block, similar to our experiment with the full image transform.

```
Yb = jl.bdct(y, (8, 8))
Ybq = np.zeros(Yb.shape)
Ybq[(0, 1, 8, 9), :]  = np.round(Yb[(0, 1, 8, 9), :])
yq2 = jl.ibdct(Ybq, (8, 8), (512, 768))
plt.figure(3), plt.imshow(yq2, 'gray', clim=(0, 255)), plt.show()
```

Each block of $8 \times 8$ transform components is stored as one column of 64 values in the transformed matrix. That's why the four lowest frequency components can be found on rows 0, 1, 8, and 9. The nine lowest frequency components can be found on rows 0, 1, 2, 8, 9, 10, 16, 17, and 18, and so on. The task is explained in the figure below.



Since we kept 4 out of 64 coefficients in each block, we have kept $1/16$ of the total number of coefficients. Moreover, we need to use 12 bits per coefficient. This means that we achieve a data rate of $12/16 = 0.75$ bpp.

**QUESTION 12**: Check the maximum and minimum value of `Ybq`. How many bits per coefficient are required?

---

**QUESTION 13**: Compare the resulting image to the one we got when transforming the whole image. Compare `psnr` and `bpp` for the same number of DCT coefficients.

---

**QUESTION 14**: Change the number of coefficients that are kept and see how the image quality varies. Write `psnr`, `bpp`, and subjective image quality assessment for at least two choices of parameters, one "half-good" and one "good"! Repeat the assessment for other image examples of "half-good" and "good".

---

## 7 Quantization

Instead of just throwing away transform components, it is smarter to keep all of them, but quantize them harder than just rounding to nearest integer. We will first be looking at uniform quantization, using the functions `jl.bquant` and `jl.brec`. Quantize all transform components with the same quantization step:

```
Q1 = 50
Ybq = jl.bquant(Yb, Q1)
Ybr = jl.brec(Ybq, Q1)
yr = jl.ibdct(Ybr, (8, 8), (512, 768))
plt.figure(3), plt.imshow(yr, 'gray', clim=(0, 255)), plt.show()
```

Now you have the quantized transform `Ybq`, the reconstructed transform `Ybr` and the reconstructed image `yr`.

**QUESTION 15**: Vary the quantization step length `Q1` and look at the reconstructed images. Write `psnr`, `Q1`, and subjective image quality assessment for at least two choices of parameters, one "half-good" and one "good"! Repeat the assessment for other image examples of "half-good" and "good".

---

We can also use different quantization steps for different transform components. The function `jl.jpgqmtx` returns one of the suggested quantization vectors for the JPEG standard. To see the different steps, do

```
Qm = jl.jpgqmtx()
Qm.reshape(8, 8)
```

**QUESTION 16**: Write this matrix here!

---

**QUESTION 17**: Which frequency components (low or high) are quantized to longer steps? And consequently, which frequency components (low or high) are more valuable according to the JPEG standard?

---

Compute the mean of the JPEG matrix.

```
JPEGMEAN = np.mean(Qm)
```

**QUESTION 18**: What is the mean of the JPEG matrix?

---

By multiplying this quantization vector with different constants, you can vary the quantization, for example:

```
Q2 = 2.8
Ybq = jl.bquant(Yb, jl.jpgqmtx()*Q2)
Ybr = jl.brec(Ybq, jl.jpgqmtx()*Q2)
yr = jl.ibdct(Ybr, (8, 8), (512, 768))
plt.figure(4), plt.imshow(yr, 'gray', clim=(0, 255)), plt.show()
```

**QUESTION 19**: Now write `psnr`, `JPEGMEAN*Q2`, and subjective image quality assessment for at least two choices of parameters, one “half-good” and one “good”! Repeat the assessment for other image examples of “half-good” and “good”.

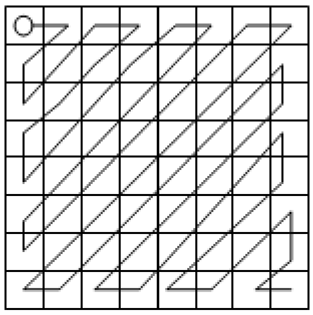
---

**QUESTION 20**: How much more can you quantize with the technique of Q2 and the JPEG matrix compared to the technique with only Q1? Give a factor for a certain level of quality, e.g. “good”!

---

## 8 Variable length coding

The third part of a transform image coder is the variable length coder (or source coder). Here, we are only going to study the JPEG standard style. The quantized transform components are rearranged in zig-zag scan order, and all consecutive zeros are run-length encoded, see figure below.



Write PYTHON code for zig-zag scanning the image block. Test the function using a matrix like
`A=np.arange(M*N).reshape(N,M)`
for suitable `M,N`.

**QUESTION 21**: Why is it beneficial to use zig-zag scanning? How can the explicit execution of the scanning function be avoided?

---

The source coding is done on the quantized transform image. Typically, Huffman coding is applied as source coding. Usually, the DC components of the blocks are coded separately, applying DPCM (differential pulse code modulation). DPCM converts a signal to its differences (plus the initial value) and applies variable length coding, which is more efficient than coding the signal directly.

## 9 Chrominance subsampling (optional)

The chrominance components (Cb and Cr) can be coded in a similar way as the luminance component. However, Cb and Cr can usually be subsampled before coding, without giving noticeable effects on the image quality. Subsampling can be done using the function `misc.imresize`.

Example:

```
plt.figure(3), plt.imshow(cb, 'gray'), plt.show()
cb2 = misc.imresize(cb, 0.5, interp='bicubic', mode='F')
```

This will subsample the chrominance image with a factor 2 both horizontally and vertically. The subsampled should then be coded (transformed, quantized and variable length coded). In the decoder, the reconstructed chrominance image is upsampled before transformation back into the RGB color space

```
cbnew = misc.imresize(cb2, 2., interp='bicubic', mode='F')
plt.figure(5), plt.imshow(cbnew, 'gray'), plt.show()
```

Perform this procedure for all components. Do a visual inspection of the images before resampling and after down- and up-sampling. Also measure `psnr` for the RGB image reconstructed with the original Y component and the resampled Y component.

**QUESTION 22**: According to your measurements and visual inspection, which component, the luminance or the chrominance components, contains more detail information? What are the `psnr` for the two choices of Y?

## 10 Python functions

Here is a list of all the special Python functions you will use in the laboratory work

| Function | Short description |
|---|---|
| `cv2.dct` | Discrete Cosine Transform |
| `cv2.idct` | Inverse DCT |
| | |
| `jl.bdct` | Block based Discrete Cosine Transform |
| `jl.ibdct` | Inverse block based DCT |
| | |
| `jl.bquant` | Block uniform quantizer |
| `jl.brec` | Block uniform reconstruction (inverse quantization) |
| `jl.jpgqmtx` | JPEG quantization vector |
| | |
| `misc.imread` | Read image from file |
| `plt.imshow` | Display an image |
| `misc.imresize` | Resize an image |
| | |
| `jl.rgb2ycbcr` | Convert from RGB to YCbCr colorspace |
| `jl.ycbcr2rgb` | Convert from YCbCr to RGB colorspace |

## 11 costrans3.py

```
#---------------------------------------------------------------------
# Program for studying the relation between the discrete cosine
# transform and the discrete fourier transform.
# Written by Maria Magnusson 2003-05-04
# Updated by Maria Magnusson 2007-04-30
# Ported to Python by Michael Felsberg 2016-03-01
#---------------------------------------------------------------------

import numpy as np
from matplotlib import pyplot as plt
import cv2

a = np.load('goldhill128.npy')                 # inimage
N = 128                                        # inimage size
b = np.vstack((np.hstack((a, a[:,::-1])), np.hstack((a[::-1,:], a[::-1,::-1]))))
# mirror inimage
u, v = np.meshgrid(np.arange(-N,N),np.arange(-N,N))
# (u,v) coordinate system

# factors to compensate for translation in the spatial domain
Ev = np.exp(-0.5j*v*np.pi/N)
Eu = np.exp(-0.5j*u*np.pi/N)

A = cv2.dct(a)                                  # discrete cosine transform

AA = np.fft.fftshift(np.fft.fft2(a))            # FFT
AA = AA/N;                                      # extra scaling

B = np.fft.fftshift(np.fft.fft2(b))             # fft + weighting
B = B*Ev*Eu
B[N,:] = B[N,:]/np.sqrt(2)
B[:,N] = B[:,N]/np.sqrt(2)
B = B/(2*N)
imagB = np.max(np.imag(B))                      # check that imag part is 0
B = np.real(B)

# image limits
IMLIM = (0, np.max(a))
lim = np.max(A)/100
FLIM = (0, lim)

# show inimage, its FFT and its DCT
plt.figure(8)
plt.subplot(221), plt.imshow(a, 'gray', clim=IMLIM)
plt.title('original image, a(x,y)'), plt.colorbar()
plt.subplot(222), plt.imshow(np.abs(AA), 'gray', clim=FLIM)
plt.title('AA(u,v) = abs(FFT[a(x,y)])'), plt.colorbar()
plt.subplot(223), plt.imshow(np.abs(A), 'gray', clim=FLIM)
plt.title('A(u,v) = abs(DCT[a(x,y)])'), plt.colorbar()

# show inimage and mirrored inimage
# show fft + weighting of mirrored inimage and show lower right part
plt.figure(9)
plt.subplot(221), plt.imshow(a, 'gray', clim=IMLIM)
plt.title('original image, a(x,y)'), plt.colorbar()
plt.subplot(222), plt.imshow(b, 'gray', clim=IMLIM)
plt.title('mirrored image, b(x,y)'), plt.colorbar()
plt.subplot(223), plt.imshow(np.abs(B), 'gray', clim=FLIM)
plt.title('B(u,v) = almost abs(FFT[b(x,y)])'), plt.colorbar()
BPART = B[N:,N:]
plt.subplot(224), plt.imshow(np.abs(BPART), 'gray', clim=FLIM)
plt.title('lower right part of B(u,v)'), plt.colorbar()
plt.show()

maxDiffAoB = np.max(A-BPART)                     # check that they are equal
```