LAB 4: JPEG Compression

Harald Nautsch, Maria Magnusson, Michael Felsberg, Johan Edstedt, 2017-2020
Avdelningen för Datorseende, Institutionen för Systemteknik,
Linköpings Universitet

Latest update: Maria Magnusson, February 2024

1 Introduction

Read this booklet before the laboratory work. Questions marked with a pointing hand should be solved as preparation before the laboratory work. Suggested answers for all questions, as well as a multiple choise table is in the end of this lab booklet. In case of any wrong answer, the problem should be discussed with the teacher.

A computer symbol means that a PYTHON script (or code block if Jupyter is used) should be written and demonstrated for the teacher.

A double teacher symbol means that you are advised to fill in parts of the multiple choise table and show some demos to the teacher. Continue with new exercises while you wait!

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, the first two more carefully and the third more concisely. We will see how the choice of transform/quantizer/variable length coder affects the performance of the transform coder.

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.

1.1 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 512×512 pixels, and we use 50000 bits to store it, the data rate is $50000/512^2 \approx 0.19$ bits per pixel.

1.2 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. In the lab booklet, we have used `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. The objective metric PSNR tends to agree well with subjective test groups.

1.3 Starting PYTHON

Copy all the files on `/courses/TSKS24/JPEGLab/` (or in the Lab4Filer folder in Lisam) to a suitable place at your own directory. You can either run `python3`, `ipython3`, or a Jupyter Notebook (similarly as in Lab 3). Import a number of packages:

```
import numpy as np
from scipy import signal, misc, ndimage
import cv2
from matplotlib import pyplot as plt
plt.rcParams['image.interpolation'] = 'nearest'
import jpeglab as jl
```

preferably saved to a file `preamble.py` that is loaded by executing:
`from preamble import *` or in a codeblock if using Jupyter.
In the last line we import `jpeglab`, which contains some helper functions. (`jpeglab.py` is at `/courses/TSKS24/JPEGLab/` and in the Lab4Filer folder, make sure to put it in the same folder as your scripts.)

2 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_bgr = cv2.imread('image1.png')
```

Here we used opencv, denoted by cv2 to open the image. This produces an image which is ordered by blue, green and red, instead of the usual red, green, blue. To reorder the colors we run the command below:

```
im1 = cv2.cvtColor(im1_bgr,cv2.COLOR_BGR2RGB)
```

The color image is stored as a $512 \times 768 \times 3$ matrix, meaning that we have one 512×768 matrix for each of the three RGB color components (red, green and blue). The image can be viewed using the commands `imshow` and `show`, see below. (For `ipython`, `plt.show(block=False)` works better.)

```
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, yellow, cyan (turquoise), white and black?

color	R	G	B
red	hi	lo	lo
green			
blue	lo	lo	hi
yellow			

color	R	G	B
cyan			
magenta	hi	lo	hi
white			
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?

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

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

There are six images (named `image1.png` to `image6.png`) with varying content available. In general, experiments in image processing should not be restricted to a single image. For simplicity, however, in this laboratory work we will concentrate on `image1.png`. If you have time left at the end of the lab, you can repeat your analysis for some of 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.

3 Introductory experiment: 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:
$\begin{array}{|c|c|c|c|c|c|c|c|}\hline 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \hline \end{array} = 255$
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:
$\begin{array}{|c|c|c|c|c|c|c|c|}\hline 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ \hline \end{array} = 16$
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?

DEMO A: Write a Python function for computing the mean-square error of two images and the `psnr`. Then write code to show for images to the teacher, the original (y) as well as images with bpp = 6, 5 and 4, respectively. These should be "good", "half-good", and "rather bad" respectively.

QUESTION 4: Write `psnr`, `bpp`, and subjective image quality for one "good" and one "half-good" image!

---

4 Introductory experiment: Down-sampling the image

Down-sampling can be done by using the function `ndimage.interpolation.zoom`, if you run a recent on your own machine, also `misc.imresize` might work. To down-sample (and then up-sample) the image by a factor `2.` in each dimension, do:

```
y3 = np.floor(ndimage.interpolation.zoom(y, 0.5, order=3))
y4 = ndimage.interpolation.zoom(y3, 2., order=3)
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 for at least two choices of parameters, one "half-good" and one "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 Introductory experiment: Transforming the full image

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

```
plt.figure(2), plt.imshow(y, 'gray', clim=(0, 255))
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()
```

DEMO B: Add titles with explanatory text to the images in figure 2, 3, 4, and 5. Show the figures to the teacher and explain what you see in the images.

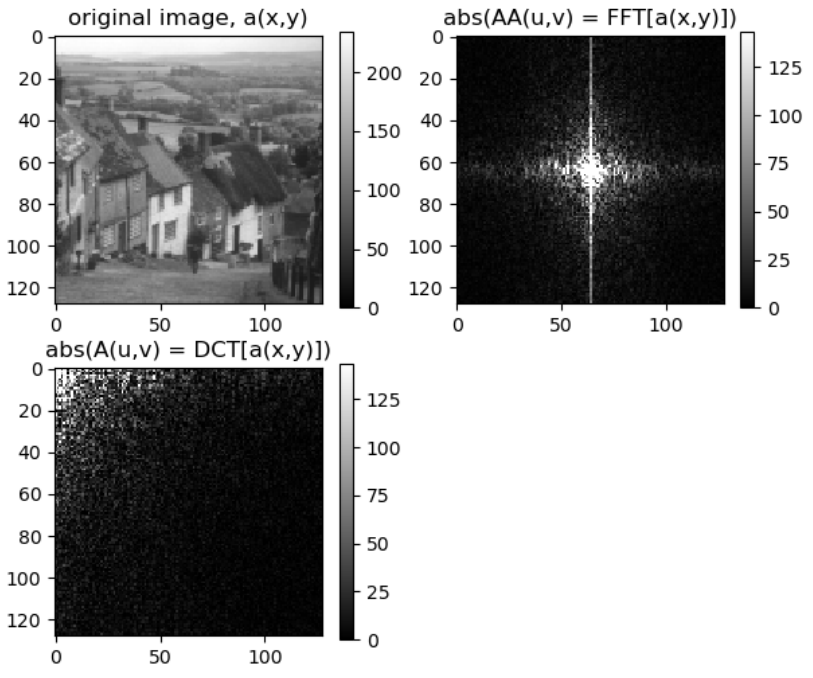
In this example we kept 1/16 of the coefficients ($128^2/512^2$) and we need to use 18 bits per coefficient.

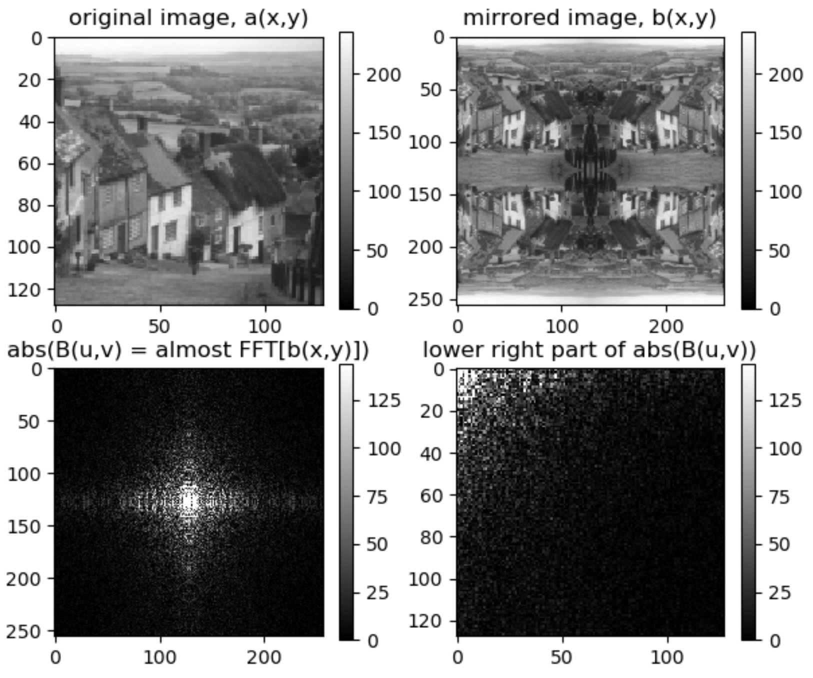
QUESTION 7: Check the maximum and minimum value of `Yq`. Why do we need to use 18 bits per coefficient?

QUESTION 8: Now write `psnr`, `bpp`, and subjective image quality for the coded image above.

---

6 Introductory experiment: DCT versus DFT





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 `dctdft.py` shows how they are related. The code is given in the end of this booklet. The images generated by the code are shown above.

QUESTION 9: How are dct and fft2 related?

QUESTION 10: The dct is purely real-valued and considers reflective boundary conditions. The dft is complex-valued and considers periodic boundary conditions.
Why is it preferable to use the dct compared to the fft2 in image coding?

Run the program `dctdft.py`, in jupyter we simply run `import dctdft`, and check that you get similar images as here in the booklet.

QUESTION 11: What is the maximum and minimum diffence between `A(u,v) = DCT[a(x,y)])` and `the lower right part of B(u,v)`?

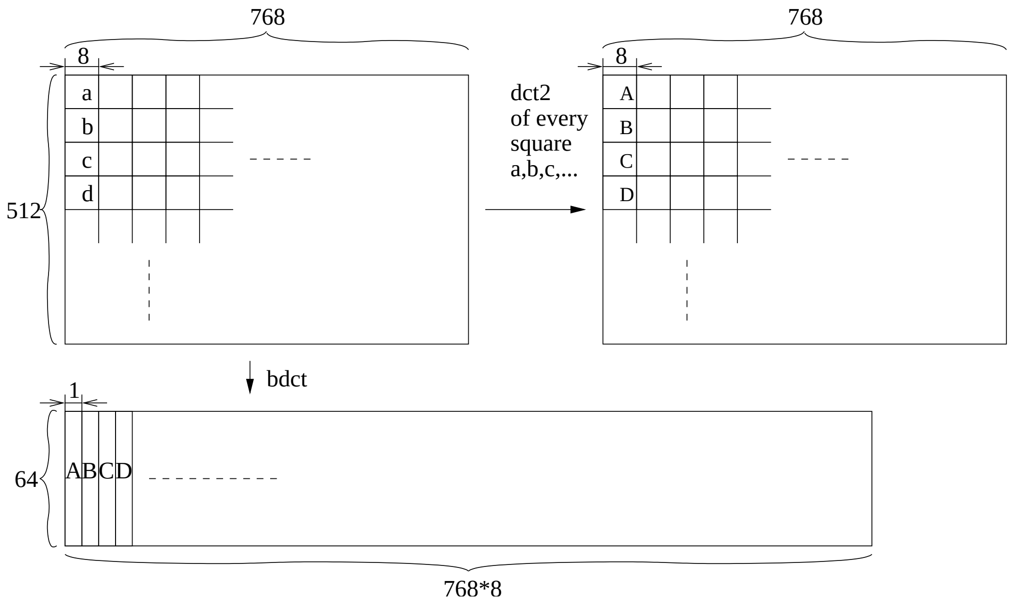
QUESTION 12: Where are the low frequencies located for the (symmetrical) DFT and the DCT, respectively?

Fill in the first column in the multiple choise table in the end of this lab booklet and show demo A to the teacher!

7 Block-based transforms

Normally in image coding, the full image is not transformed with a single transform. Rather, the image is divided into small blocks (typically 8×8 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 dicrete cosine transform for 8×8 signals. Consequently, `A = dct2(a)`, `B = dct2(b)`, etc.

Execute the following code.

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

Note that, as predicted by the figure, `Yb` is very long i the horizontal direction.

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:

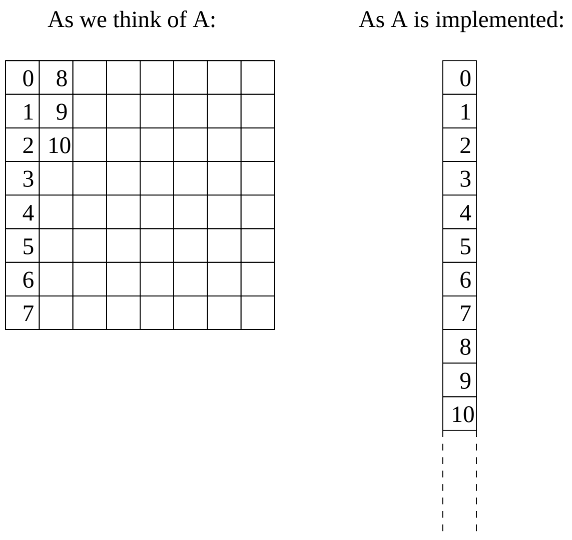
```
np.max(np.abs(y-yn))
```

QUESTION 13: 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_like(Yb)
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 8×8 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 14: Check the maximum and minimum value of `Ybq`. How many bits per coefficient are required?

QUESTION 15: Compare the resulting image to the one we got when transforming the full image. Write down `psnr` and `bpp` for both images, as well as the subjective image quality.

QUESTION 16: Instead of keeping 4 out of 64 DCT coefficients, keep 9 coefficients out of 64 DCT coefficients. Write down `bpp`, `psnr`, and subjective image quality.

DEMO C: Show the two latest images to the teacher, one when 4 out of 64 DCT coefficients were kept and one when 9 out of 64 DCT coefficionts were kept. Be sure to see the small 8×8 block artefacts in the images!

8 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`. It is not apparent how to measure bpp here. We will also get help from the variable length coding. A very uncertain estimated bpp is Ebpp = 12/Q1.

QUESTION 17: Vary the quantization step length `Q1` and look at the reconstructed images. Write `Q1`, `psnr`, subjective image quality, `Ebpp`, for at least two choices of parameters, one "half-good" and one "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 18: Complete the JPEG matrix below!

$$\begin{bmatrix} & & 14 & 14 & 18 & 24 & 49 & 72 \\ & & 13 & 17 & 22 & 35 & 64 & 92 \\ 10 & 14 & 16 & 22 & 37 & 55 & 78 & 95 \\ 16 & 19 & 24 & 29 & 56 & 64 & 87 & 98 \\ 24 & 26 & 40 & 51 & 68 & 81 & 103 & 112 \\ 40 & 58 & 57 & 87 & 109 & 104 & 121 & 100 \\ 51 & 60 & 69 & 80 & 103 & 113 & & \\ 61 & 55 & 56 & 62 & 77 & 92 & & \end{bmatrix}$$

QUESTION 19: 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?

QUESTION 20: What is the mean of the JPEG matrix? Compute it by `JPEGMEAN = np.mean(Qm)`. What does this number represent?

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()
```

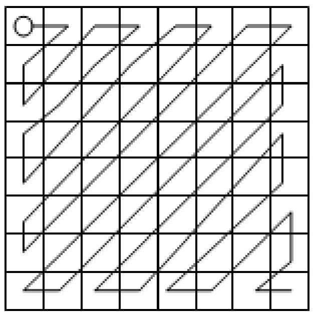
QUESTION 21: Now write `Q2`, `JPEGMEAN*Q2`, psnr, subjective image quality, `Ebpp=12/(JPEGMEAN*Q2), for at least two choices of parameters, one ‘‘half-good’’ and one ‘‘good’’!`

DEMO D: `Show the two latest images to the teacher. Point out the artifacts in the ‘‘half-good’’ image.`

QUESTION 22: `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’’!`

9 Variable length coding

The third part of a transform image coder is the variable length coder. The quantized transform components are rearranged in zig-zag scan order, and all consecutive zeros are run-length encoded, see figure below.



Our goal with the zig-zag scan is to order the elements in such a way that similar values end up next to each other. This makes run-length encoding more efficient, especially when we have long sequences of only zeros.

QUESTION 23: Why is it beneficial to use zig-zag scanning for run-length encoding?

Variable length coding is a more advanced relative to run-length encoding. Typically, Huffman coding is used. It is applied on the quantized transform image. 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.

9.1 (Optional) Zig-zag investigation

We can rewrite the zig-zag scanning as a matrix which reorders the elements in a vector. Such a matrix is called a permutation matrix.

To ensure that the permutation matrix produces the same results as a zig-zag scan, we can test the matrix for a block where we keep track of where the indices go.

```
dct_block=np.arange(64)
Z = jl.zigzag_matrix()
zigzagged_block = Z@dct_block #@ means matrix multiplication in numpy
zigzag_indices = jl.zigzag_indices()
```

`Check that the zigzagged block and the zigzag indices are the same.`

QUESTION: `How can we use the zigzag matrix to avoid the explicit execution of the scanning function?`
`Hint: Remember that the BDCT already performs a matrix multiplication for the transform, how can we combine these matrices?`

Now we will see the result of zig-zag scanning on the actual transformed image! Write the following commands:

```
plt.plot(np.mean(np.abs(Ybq),axis=1)[1:])
plt.plot((Z@np.mean(np.abs(Ybq),axis=1))[1:])
plt.legend(["original","zig-zag"])
plt.show()
```

Here we clearly see that the original block has a clear periodic nature because the scan goes column by column. In the zig-zag however we end up smoothly going to 0.

10 Chrominance subsampling

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 `ndimage.zoom` (or `misc.imresize`). Example:

```
plt.figure(3), plt.imshow(cb, 'gray'), plt.show()
cb2 = ndimage.zoom(cb, 0.5, order=3)
```

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 = ndimage.zoom(cb2, 2., order=3)
```

```
plt.figure(4), 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 all three components.

QUESTION 24: According to your measurements and visual inspection, which component contains the most detailed information? What are the `psnr` for the three components Y, Cb and Cr?

Fill in the second column in the multiple choise table in the end of this lab booklet and show demo B and C to the teacher!

11 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
`ndimage.zoom`	Resize an image
`jl.rgb2ycbcr`	Convert from RGB to YCbCr colorspace
`jl.ycbcr2rgb`	Convert from YCbCr to RGB colorspace

12 Suggested answers

ANSWER 1:

a)

color	R	G	B
red	hi	lo	lo
green	lo	hi	lo
blue	lo	lo	hi
yellow	lo	hi	hi

color	R	G	B
cyan	hi	hi	lo
magenta	hi	lo	hi
white	hi	hi	hi
black	lo	lo	lo

b)

color	R	G	B
red	hi	lo	lo
green	lo	hi	lo
blue	lo	lo	hi
yellow	lo	hi	hi

color	R	G	B
cyan	hi	lo	hi
magenta	hi	lo	hi
white	hi	hi	hi
black	lo	lo	lo

c)

color	R	G	B
red	hi	lo	lo
green	lo	hi	lo
blue	lo	lo	hi
yellow	hi	hi	lo

color	R	G	B
cyan	lo	hi	hi
magenta	hi	lo	hi
white	hi	hi	hi
black	lo	lo	lo

ANSWER 2:
a) Most information is contained in the Y-component.
b) Most information is contained in the Cb-component.
c) Most information is contained in the Cr-component.

ANSWER 3:
a) $X = 4$
b) $X = 5$
c) $X = 8$

ANSWER 4:
a)
4 `bpp` $\Rightarrow$ `psnr`=40.9 dB and “good”
3 `bpp` $\Rightarrow$ `psnr`=34.9 dB and “half-good”
b)
5 `bpp` $\Rightarrow$ `psnr`=40.9 dB and “good”
4 `bpp` $\Rightarrow$ `psnr`=34.9 dB and “half-good”
c)
6 `bpp` $\Rightarrow$ `psnr`=40.9 dB and “good”
5 `bpp` $\Rightarrow$ `psnr`=34.9 dB and “half-good”

ANSWER 5:
a)
Downsample by 2 $\Rightarrow$ 2 `bpp`, `psnr`=35.1 dB and “half-good”
Downsample by 4/3=1.334 $\Rightarrow$ 4.5 `bpp`, `psnr`=42.3 dB and “good”
b)
Downsample by 2 $\Rightarrow$ 2 `bpp`, `psnr`=42.3 dB and “good”
Downsample by 4/3=1.334 $\Rightarrow$ 4.5 `bpp`, `psnr`=35.1 dB and “half-good”
c)
Downsample by 2 $\Rightarrow$ 2 `bpp`, `psnr`=30.1 dB and “half-good”
Downsample by 4/3=1.334 $\Rightarrow$ 4.5 `bpp`, `psnr`=49.3 dB and “good”

ANSWER 6:
a) The largest errors is located in the high-frequency areas around the eyes of the birds.
b) The largest errors is located in the beaks of the birds.
c) The largest errors is located in the background, behind the birds.

ANSWER 7:
a) Two bits for the sign. 16 bits ($2^{16} = 65536$) to represent the largest value (65423).
b) 18 bits ($2^{18} = 131072$) is needed to represent the largest value (68923).
c) One bit for the sign. 17 bits ($2^{17} = 131072$) to represent the largest value (68923).

ANSWER 8:
a) 18/16 = 1.125 `bpp`, `psnr`=39.0 dB, “good”
b) 18/16 = 1.125 `bpp`, `psnr`=35.0 dB, “half-good”
c) 18/16 = 1.125 `bpp`, `psnr`=31.0 dB, “worse than half-good”

ANSWER 9:
a) Extending the image by mirroring at both axes and applying the fft2 gives, up to some modulation, the dct.
b) Applying the fft2 and taking the lower right part gives the dct.
c) Applying the fft2, then mirroring and removing 3/4 of the data gives the dct.

ANSWER 10:
a) Since the dct is purely real-valued, it is easier to compress.
b) The dft othen gives a sharp horizonal and/or vertical line. This is avoided with the dct. Consequently, the dct is easier to compress.
c) The size of the dct is only 1/4 compared to the size of the dft. Consequently, the dct can be compressed 4 times more than the dft.

ANSWER 11:
a) The maximum is a very small positive value and the minimum is a very small negative value.
b) The maximum is a very small positive value and the minimum is 0.
b) Both the maximum and the minimum are equal to 0.

ANSWER 12:
a) The dft has the low frequences in the middle and the dct has the low frequences in the lower right part.
b) The dft has the low frequences in upper left part and the dct has the low frequences in the lower right part.
c) The dft has the low frequences in the middle and the dct has the low frequences in the upper left part.

ANSWER 13:
a) Around 7e-7. b) Around 7e-10. c) Around 7e-13.

ANSWER 14:
a) The maximum is 1880 and the minimum -470. Thus 12 bits i.e. a sign-bit plus 11 bits ($2^{11} = 2048$) are sufficient.
b) The maximum is 3602 and the minimum -470. Thus 12 bits ($2^{12} = 4096$) are sufficient.
c) The maximum is 120 and the minimum -27. Thus 12 bits $= 7 + 5$ bits are sufficient ($2^7 = 128 > 120$ and $2^5 = 32 > 27$).

ANSWER 15:
Full image: 1.125 bpp, psnr=31.0 dB, worse than half-good.
a) Block-wise: 0.75 bpp, psnr=30.1 dB, worse than half-good.
The block-wise can be more compressed!
b) Block-wise: 0.75 bpp, psnr=35.1 dB, half-good.
The block-wise can be much more compressed!
c) Block-wise: 0.75 bpp, psnr=27.5 dB, rather bad.
It is unclear which can be compressed the most.

ANSWER 16:
a) 9 coefficients $\Rightarrow$ 1.69 bpp, psnr=32.7 dB, almost half-good
b) 9 coefficients $\Rightarrow$ 1.69 bpp, psnr=34.7 dB, half-good
c) 9 coefficients $\Rightarrow$ 1.69 bpp, psnr=38.7 dB, good

ANSWER 17:
a) $Q1 = 50 \Rightarrow$ psnr = 34.7 dB, half-good. Ebpp = 12/50 = 0.24.
$Q1 = 20 \Rightarrow$ psnr = 39.9 dB, good. Ebpp = 12/20 = 0.6.
b) $Q1 = 50 \Rightarrow$ psnr = 34.7 dB, half-good. Ebpp = 12/50 = 0.24.
$Q1 = 18 \Rightarrow$ psnr = 39.9 dB, good. Ebpp = 12/18 = 0.67.
c) $Q1 = 50 \Rightarrow$ psnr = 34.7 dB, half-good. Ebpp = 12/50 = 0.24.
$Q1 = 15 \Rightarrow$ psnr = 39.9 dB, good. Ebpp = 12/15 = 0.8.

ANSWER 18:
a)

$$\begin{bmatrix} 12 & 16 & 14 & 14 & 18 & 24 & 49 & 72 \\ 12 & 11 & 13 & 17 & 22 & 35 & 64 & 92 \\ 10 & 14 & 16 & 22 & 37 & 55 & 78 & 95 \\ 16 & 19 & 24 & 29 & 56 & 64 & 87 & 98 \\ 24 & 26 & 40 & 51 & 68 & 81 & 103 & 112 \\ 40 & 58 & 57 & 87 & 109 & 104 & 121 & 100 \\ 51 & 60 & 69 & 80 & 103 & 113 & 103 & 120 \\ 61 & 55 & 56 & 62 & 77 & 92 & 99 & 101 \end{bmatrix}$$

b)

$$\begin{bmatrix} 16 & 12 & 14 & 14 & 18 & 24 & 49 & 72 \\ 11 & 12 & 13 & 17 & 22 & 35 & 64 & 92 \\ 10 & 14 & 16 & 22 & 37 & 55 & 78 & 95 \\ 16 & 19 & 24 & 29 & 56 & 64 & 87 & 98 \\ 24 & 26 & 40 & 51 & 68 & 81 & 103 & 112 \\ 40 & 58 & 57 & 87 & 109 & 104 & 121 & 100 \\ 51 & 60 & 69 & 80 & 103 & 113 & 120 & 103 \\ 61 & 55 & 56 & 62 & 77 & 92 & 101 & 99 \end{bmatrix}$$

c)

$$\begin{bmatrix} 11 & 12 & 14 & 14 & 18 & 24 & 49 & 72 \\ 16 & 12 & 13 & 17 & 22 & 35 & 64 & 92 \\ 10 & 14 & 16 & 22 & 37 & 55 & 78 & 95 \\ 16 & 19 & 24 & 29 & 56 & 64 & 87 & 98 \\ 24 & 26 & 40 & 51 & 68 & 81 & 103 & 112 \\ 40 & 58 & 57 & 87 & 109 & 104 & 121 & 100 \\ 51 & 60 & 69 & 80 & 103 & 113 & 101 & 99 \\ 61 & 55 & 56 & 62 & 77 & 92 & 120 & 103 \end{bmatrix}$$

ANSWER 19:
a) The low frequency components have smaller quantization steps than the high frequency components. Thus low frequency components are more valuable according to the JPEG standard.
b) The high frequency components have smaller quantization steps than the low frequency components. Thus high frequency components are more valuable according to the JPEG standard.
c) The low frequency components have smaller quantization steps than the high frequency components. Thus high frequency components are more valuable according to the JPEG standard.

ANSWER 20:
a) The mean is 37.625. The average quantization step size.
b) The mean is 47.625. The average quantization step size.
c) The mean is 57.625. The average quantization step size.

ANSWER 21:
`Q2` = 2.8, `JPEGMEAN*Q2` = 161.35 ⇒ `psnr=34.8 dB`, "half-good".
`Ebpp` = 12/161.35 = 0.074.
a) `Q2` = 0.9, `JPEGMEAN*Q2` = 51.86 ⇒ `psnr=38.9 dB`,"good".
`Ebpp` = 12/51.86 = 0.23.
b) `Q2` = 1.9, `JPEGMEAN*Q2` = 109.49 ⇒ `psnr=38.9 dB`,"good".
`Ebpp` = 12/109.49 = 0.11.
c) `Q2` = 2.9, `JPEGMEAN*Q2` = 167.11 ⇒ `psnr=38.9 dB`,"good".
`Ebpp` = 12/167.11 = 0.07.

ANSWER 22:
a) For "good". Q1: Ebpp = 0.67. Q2,JPEG: Ebpp = 0.23 ⇒ A factor 3.
b) For "good". Q1: Ebpp = 0.67. Q2,JPEG: Ebpp = 0.11 ⇒ A factor 6.
c) For "good". Q1: Ebpp = 0.67. Q2,JPEG: Ebpp = 0.07 ⇒ A factor 10.

ANSWER 23:
a) For the run-length coding it is beneficial to have **long** sequences of the same value. In the DCT domain, zig-zag scanning sorts coefficients such that similar frequencies are neighbored and the likelihood for having **the same** value is maximized.
b) For the run-length coding it is beneficial to have **short** sequences of the same value. In the DCT domain, zig-zag scanning sorts coefficients such that similar frequencies are neighbored and the likelihood for having **the same** value is maximized.
c) For the run-length coding it is beneficial to have **long** sequences of the same value. In the DCT domain, zig-zag scanning sorts coefficients such that different frequencies are neighbored and the likelihood for having **different** values is maximized.

ANSWER 24:
a) Luminance Y. The resampled Y lets the image looking slightly blurry.
Y: psnr = 35.2, Cb: psnr = 47.4, Cr: psnr = 50.0.
b) Chrominace Cb. The resampled Cb lets the image looking slightly blurry.
Y: psnr = 50.0, Cb: psnr = 35.2, Cr: psnr = 47.4.
c) Chrominace Cr. The resampled Cr lets the image looking slightly blurry.
Y: psnr = 47.4, Cb: psnr = 50.0, Cr: psnr = 35.2.

13 Examination

Multiple choise table

	a	b	c
1:			
2:			
3:			
4:			
5:			
6:			
7:			
8:			
9:			
10:			
11:			
12:			
13:			
14:			
15:			
16:			
17:			
18:			
19:			
20:			
21:			
22:			
23:			
24:			

Questions and Demonstrations approved by the teacher

Column 1 Demo A, B	Column 2 Demo C, D