ExerciseSheet2

November 7, 2021

1 IPSA 2021 - Exercise 2 - Superpixels and tiling effects

1.1 0 - Practical advice

In [1]:
```
import imageio
import numpy as np
import numpy.random as rnd
import scipy.ndimage as img
import matplotlib.pyplot as plt

import timeit, functools
```

In [2]:
```
#!wget https://github.com/FabriceBeaumont/1-MA-INF_2314_IPSA_Repo/blob/main/Exercises/
```

In [3]:
```
def imageRead(imgname, pilmode ='L', arrtype=np.float):
    """
    Read an image file into a numpy array

    imgname: str
        name of image file to be read
    pilmode: str
        for luminance / intesity images use L
        for RGB color images use RGB

    arrtype: numpy dtype
        use np.float, np.uint8, ...
    """
    return imageio.imread(imgname, pilmode=pilmode).astype(arrtype)

def imageWrite(arrF, imgname, arrtype=np.uint8):
    """
    Write a numpy array as an image file
    the file type is inferred from the suffix of parameter imgname, e.g. .png
    arrF: array_like
        array to be written
    imgname: str
        name of image file to be written
    arrtype: numpy dtype
```

```
        use np.uint8, ...
    """
    imageio.imwrite(imgname, arrF.astype(arrtype))
```

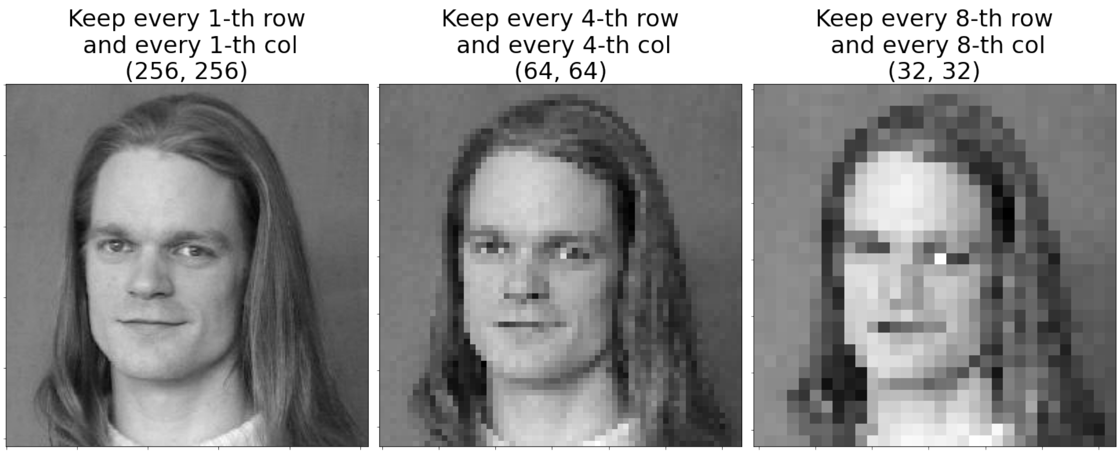
1.2 1 - (Nave) downsampling

A rather simple idea for how to reduce the resolution of a digital (intensity) image is to keep only every m-th row and every n-th column of its pixels.

Without using for loops, implement a function downsample with three parameters `arrF`, `m`, and `n` that realizes this manner of downsampling.

```
In [5]: def naive_downsampling(arrF, m, n):
            deleted_rows = arrF[::m, :]
            deleted_rows_and_columns = deleted_rows[:, ::n]

            return deleted_rows_and_columns
```

```
In [27]: filename = "portrait.png"
         arrF = imageRead(filename)

         parameters = [(1, 1), (4, 4), (8, 8)]

         fig, axs = plt.subplots(1, len(parameters), figsize=(20,20))

         for i, (m, n) in enumerate(parameters):
             arrG = naive_downsampling(arrF, m, n)

             ax = axs[i]
             ax.imshow(arrG, cmap='gray')
             ax.set_title(f"Keep every {m}-th row\n and every {n}-th col\n{arrG.shape}", fonts

             ax.set_xticklabels([]); ax.set_yticklabels([])
             fig.tight_layout()
```



1.3 2 - Kronecker products for (nave) upsampling

The Kronecker product of an ordered pair of matrices (or 2D *numpy* arrays) A, B of sizes $k \times l$ and $m \times n$ respectively is defined as

$$C = A \otimes B = \begin{bmatrix} a_{1,1}B & a_{1,2}B & \dots & a_{1,l}B \\ a_{2,1}B & a_{2,2}B & \dots & a_{2,l}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{k,1}B & a_{k,2}B & \dots & a_{k,l}B \end{bmatrix}$$

and therefore produces a matrix (or 2D array) C of size $km \times ln$.

Conveniently, numpy provides the function `np.kron()` for the computation of Kronecker products. This allows us to realize a rather simple idea for upsampling a small (intensity) image: assuming that the given image is stored in an array F, we may simply compute

$$G = F \otimes O$$

where O denotes an $m \times n$ array of all ones.

Now, without using for loops, implement an appropriately parameterized function upsample. Then, load image `portrait.png` into `arrF` and compute it.

```
In [25]: def naive_upsampling(arrF, m, n):
             O = np.ones((m, n))

             return np.kron(arrF, O)
```

```
In [28]: filename = "portrait.png"
         arrF = imageRead(filename)

         parameters = [(1, 1), (2, 2), (4, 4), (8, 8)]

         fig, axs = plt.subplots(1, len(parameters), figsize=(20,20))

         for i, (m, n) in enumerate(parameters):
             arrG = naive_upsampling(naive_downsampling(arrF, m, n), m, n)

             ax = axs[i]
             ax.imshow(arrG, cmap='gray')
             ax.set_title(f"m=n={m}\n{arrG.shape}", fontsize=30)
             ax.set_xticklabels([]); ax.set_yticklabels([])
             fig.tight_layout()
```