exercise 2

superpixels and tiling effects

solutions due

until **November 14, 2021** at **23:59** via **ecampus**

students handing in this solution set

last name	first name	student ID	enrolled with
Bach	Franziska	123456	B-IT / RWTH Aachen
Wolfe	Frank	654321	Uni Bonn

practical advice

The problem specifications you'll find below assume that you work with python / numpy / scipy. They also assume that you have imported

```
import imageio
import numpy as np
import numpy.random as rnd
```

To read- and write images from- and to disc, you may use these functions

```
def imageRead(imgname, pilmode='L', arrtype=np.float):
    return imageio.imread(imgname, pilmode=pilmode).astype(arrtype)


def imageWrite(arrF, imgname, arrtype=np.uint8):
    imageio.imwrite(imgname, arrF.astype(arrtype))
```

To display an intensity image on your screen, you could use the following

```
import matplotlib.pyplot as plt

arrF = imageRead('portrait.png')
plt.imshow(arrF / 255, cmap='gray')
plt.show()
```

To display an (RGB) color image on your screen, you might want to use

```
import matplotlib.pyplot as plt

arrF = imageRead('../exercise1/Data/asterixRGB.png', pilmode='RGB')
plt.imshow(arrF / 255)
plt.show()
```

task 2.1

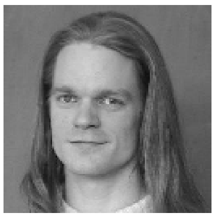
(naïve) downsampling

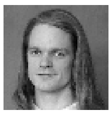
A rather simple idea for how to reduce the resolution of a digital (intensity) image is to keep only every m-th row and every n-th column of its pixels. For example, the figure below shows the effect of *downsampling* by a factor of 2, i.e. *downsampling* by means of choosing $m = n = 2$.

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

For $(m, n) \in \{(4, 4), (8, 8)\}$, apply your method to image `portrait.png` and enter your results here

256×256


128×128


64×64

32×32

Also, paste your code here

```
def downsample(arrF, mn=(1, 1)):
    m, n = mn
    arrG = arrF[::m, ::n]
    return arrG
```

task 2.2

Kronecker products for (naïve) upsampling

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

$$\boldsymbol{C} = \boldsymbol{A} \otimes \boldsymbol{B} = \begin{bmatrix} a_{11}\boldsymbol{B} & a_{12}\boldsymbol{B} & \cdots & a_{1l}\boldsymbol{B} \\ a_{21}\boldsymbol{B} & a_{22}\boldsymbol{B} & \cdots & a_{2l}\boldsymbol{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{k1}\boldsymbol{B} & a_{k2}\boldsymbol{B} & \cdots & a_{kl}\boldsymbol{B} \end{bmatrix}$$

and therefore produces a matrix (or 2D array) $\boldsymbol{C}$ of size $k \cdot m \times l \cdot n$.

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 $\boldsymbol{F}$, we may simply compute

$$\boldsymbol{G} = \boldsymbol{F} \otimes \boldsymbol{O}$$

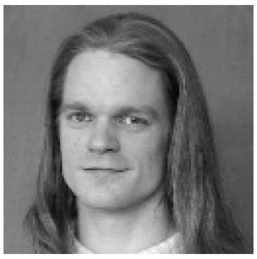
where $\boldsymbol{O}$ denotes an $m \times n$ array of all ones.

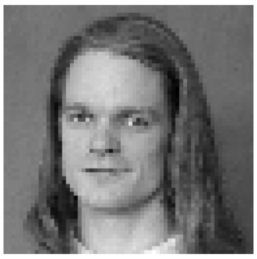
Now, without using `for` loops, implement an appropriately parameterized function `upsample`. Then, load image `portrait.png` into `arrF` and compute

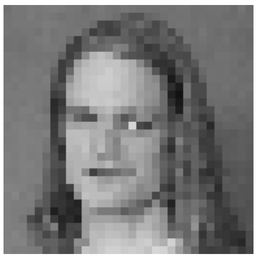
```
arrG = upsample(downsample(arrF, m, n), m, n)
```

Choose $(m, n) \in \{(2,2), (4,4)\}$ and enter your results here (the figure already shows how your result should look like for $m = n = 8$)

$m = n = 1$


$m = n = 2$


$m = n = 4$


$m = n = 8$

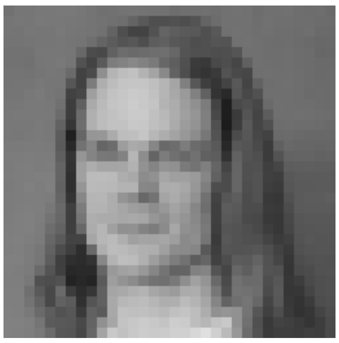
task 2.3

superpixels (part 1)

In the previous task, we have already seen examples of images composed of *superpixels*. These are images of size $M \times N$ whose content consists of uniformly colored tiles of size $m \times n$ where $m \ll M$ and $n \ll N$.

In the previous task, the superpixels resulted from (naïvely) upsampling a small image into a larger one. However, superpixels also arise in an image effect called *pixelize* (which is often used to render faces unrecognizable). In this effect, each pixel in each $m \times n$ tile of a given image is set to the average intensity value within the tile.

given image


8×8 pixelize

16×16 pixelize

As always, there are many ways for how to implement this effect. Here is an intuitive C-style implementation that involves four nested `for` loops

```
def meanSuperPixelV1(arrF, m, n):
    M, N = arrF.shape

    arrG = np.zeros((M, N))

    for i in range(0, M, m):
        for j in range(0, N, n):
            intensity_sum = 0
            for k in range(m):
                for l in range(n):
                    intensity_sum += arrF[i+k,j+l]
            intensity_avg = intensity_sum / (m*n)
            arrG[i:i+m,j:j+n] = intensity_avg

    return arrG
```

We don have to be quite so naïve but can replace the two innermost `for` loops using the *numpy* function `np.mean()`

```
def meanSuperPixelV2(arrF, m, n):
    M, N = arrF.shape

    arrG = np.zeros((M, N))

    for i in range(0, M, m):
        for j in range(0, N, n):
            arrG[i:i+m,j:j+n] = np.mean(arrF[i:i+m,j:j+n])

    return arrG
```

As almost always, *numpy* allows us to get rid of `for` loops alltogether. Here is a solution inspired by the kind of (unfortunately often stupid) answers you will get when asking the **stackoverflow** community (we will not even bother discussing this messy mumble . . .)

```
def meanSuperPixelV3(arrF, m, n):
    M, N = arrF.shape

    arrG = np.reshape(arrF, (M*N//n, n))
    arrG = np.mean(arrG, axis=1)
    arrG = np.reshape(arrG, (M,N//n))

    arrH = np.reshape(arrG, (m, (M//m)*(N//n)), 'F')
    arrH = np.mean(arrH, axis=0)
    arrH = np.reshape(arrH, (M//m, N//n), 'F')

    return np.kron(arrH, np.ones((m,n)))
```

Finally, here is how *numpy* black belts implement the pixelize effect (we will discuss this coding pattern later in the course)

```
def meanSuperPixelV4(arrF, m, n):
    M, N = arrF.shape

    arrA = np.add.reduceat(arrF, np.arange(0,N,n), axis=1) / n
    arrB = np.add.reduceat(arrA, np.arange(0,M,m), axis=0) / m
    arrC = np.repeat(arrB, n, axis=1)
    arrD = np.repeat(arrC, m, axis=0)

    return arrD
```

Here is your task: letting $m = n = 8$, perform runtime measurements for the above four methods on `portrait.png` and enter your results here

meanSuperPixelV1 took 0.026523045859939885 s
meanSuperPixelV2 took 0.02531885611009784 s
meanSuperPixelV3 took 0.0009389014900079929 s
meanSuperPixelV4 took 0.00030844773005810565 s

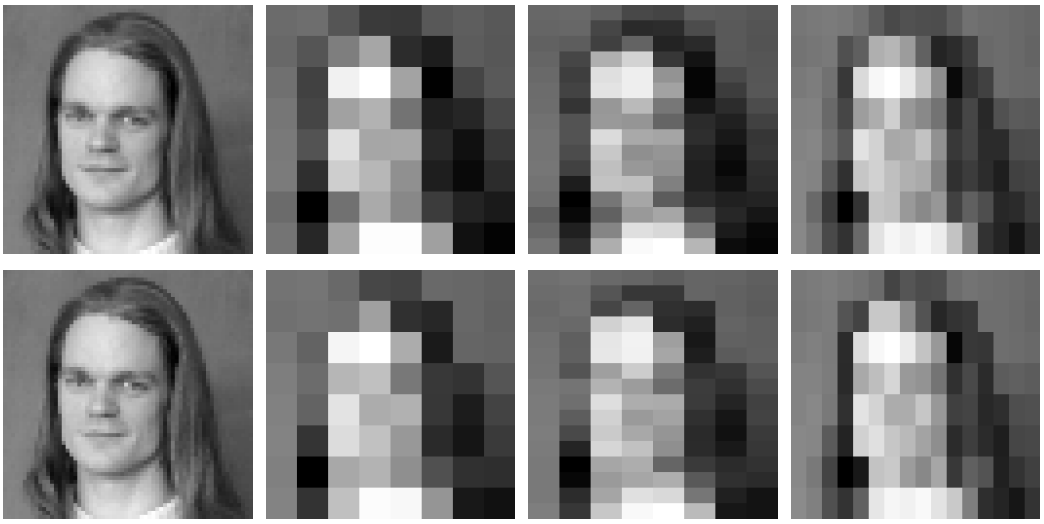
task 2.4

superpixels (part 2)

Instead of working with the *mean* intensity value in each $m \times n$ tile of an image, we can also pixelize by using *median* intensities.

Implement a function `medianSuperPixel` simply by replacing `np.mean()` in `meanSuperPixelV2` with `np.median()`.

For $(m, n) \in \{(4,4),(32,32),(16,32),(32,16)\}$, run `meanSuperPixelV2` and `medianSuperPixel` on `portrait.png` and paste your results here



task 2.5

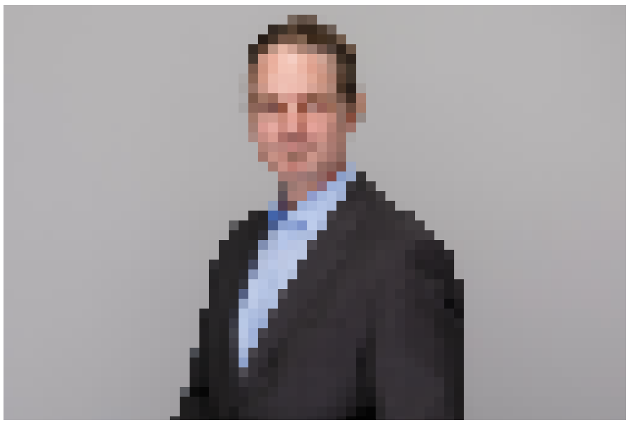
why do we always work with small images ?

Some of you may wonder why the example images we consider in these exercises are always rather small (say of resultion 256×256 pixels) ... To make a long story short, this is in order not to tax you patience. We already saw that large images take longer to process. Just for the fun it, let us therefore test your patience and pixelize a larger color image.

In the `Data` folder for this exercise, you will find the image

```
bauckhage.jpg
```

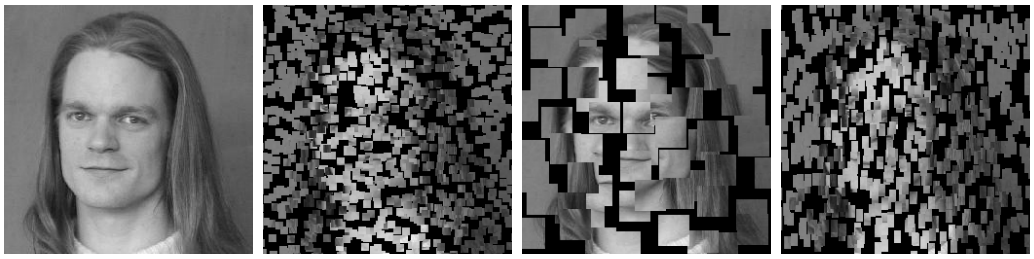
This image has a resolution of 4068×2712 pixels and is therefore still of moderate size given present day standards. Read this color image into a *numpy* array `arrF`, run *`medianSuperPixel`* on each of its color layers, and write your result as a PNG image. Experiment with different tile sizes $m \times n$ and paste one of your results here



task 2.6

“simple” tiling effects

Speaking of tiles of size $m \times n$, can you think of a piece of code that creates images like these where tiles have been moved around randomly?


input image tile size 8×8 tile size 32×32 tile size 13×7

Note: the example with the 13×7 tiles is just to show off . . . for your own solution, you may simply focus on square input images and square tiles whose side lengths are proper divisors of the side lengths of the image.

Note: *numpy* code for the above effect can again be realized without any `for` loops . . . However, this requires truly advanced insights into the capabilities of *numpy* and has nothing to do with understanding how image processing works. You can therefore devise an intuitive solution with as many `for` loops as you deem necessary. (But brace yourself for likely slow execution times.)

Paste your code here

```
def random_tiling(arrF, tile_size=(13, 7), max_offset=10):
    m, n = arrF.shape
    arrG = np.zeros_like(arrF)
    # get indices of meshgrid
    indices = np.indices((m, n))
    # get random offsets in both x and y directions tilewise
    n_offsets = np.ceil(m / tile_size[0]).astype(int), \
                np.ceil(n / tile_size[1]).astype(int)
    offsets = np.random.randint(-max_offset, max_offset + 1,
                                (2, *n_offsets))
    # repeat each offset tile_size times
    offsets_tiled = np.kron(offsets, np.ones((1, *tile_size))).astype(int)[:, :m, :n]
    # add offsets
    indices += offsets_tiled
    # make sure indices are in bounds
    indices = np.clip(indices, 0, 255)
    # set input to the offset position in the output array
    arrG[indices[0], indices[1]] = arrF
    return arrG
```

task 2.7

outer products

The outer product of an ordered pair of vectors (or 1D *numpy* arrays) $\boldsymbol{x}, \boldsymbol{y}$ of sizes m and n respectively is the Kronecker product of $\boldsymbol{x}$ and $\boldsymbol{y}^\intercal$

$$\boldsymbol{Z} = \boldsymbol{x} \otimes \boldsymbol{y}^\intercal = \begin{bmatrix} x_1\,\boldsymbol{y}^\intercal \\ x_2\,\boldsymbol{y}^\intercal \\ \vdots \\ x_m\,\boldsymbol{y}^\intercal \end{bmatrix} = \begin{bmatrix} x_1y_1 & x_1y_2 & \cdots & x_1y_n \\ x_2y_1 & x_2y_2 & \cdots & x_2y_n \\ \vdots & \vdots & \ddots & \vdots \\ x_my_1 & x_my_2 & \cdots & x_my_n \end{bmatrix}$$

and therefore produces a matrix (or 2D array) $\boldsymbol{Z}$ of size $m \times n$.

Note that we typically write the vector outer product $\boldsymbol{x} \otimes \boldsymbol{y}^\intercal$ as $\boldsymbol{x}\boldsymbol{y}^\intercal$ and that the entries of $\boldsymbol{Z}$ are given by $z_{ij} = x_i y_j$.

Conveniently, *numpy* provides the function `np.outer()` for the computation of outer products of vectors (or 1D arrays). This will come in handy later in this course. For now, execute the following snippet

```
sigma = 5.
msize = int(np.ceil(sigma * 2.575) * 2 + 1)

xs    = np.arange(msize)
vecG  = np.exp(-0.5 * ((xs-msize/2) / sigma)**2).reshape(msize,1)
vecG /= np.sum(vecG)
```

determine the size of array `vecG` and enter your result here

(27, 1)

Next, execute the following snippet

```
matG  = np.outer(vecG, vecG)
matG /= np.sum(matG)
```

determine the size of array `matG` and enter your result here

(27, 27)

Note: There is a difference between the outer product of two vectors and the Kronecker product of two vectors. To see this difference for yourself, execute the following snippet

```
vecH  = np.kron(vecG, vecG)
vecH /= np.sum(vecH)
```

determine the size of array `vecH` and enter your result here

(729, 1)

Finally, turn the two arrays `vecG` and `matG` into two arrays `arrg` and `arrG` which you can save as PNG images. To this end, execute the following snippet

```
arrg = np.interp(vecG, (vecG.min(), vecG.max()), (0, 255))
arrG = np.interp(matG, (matG.min(), matG.max()), (0, 255))
```

then write the resulting images to disc and have a look at them.

Can you “see” what you just computed? That is, do you recognize what the two images visualize?

The `arrg` is a 1d-array that resembles a discretized version of a univariate gaussian distribution. `arrG` is a 2d-array that is a discretized version of a joint distribution of two normally i.i.d. random variables, i.e. the outer product of `arrg` with itself. That gives us a bivariate gaussian distribution with a spherical covariance matrix