exercise 1

warm up

solutions due

until **November 7, 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

general remarks

As you know, your instructor is an avid proponent of open science and education. Therefore, **MATLAB implementations will not be accepted** in this course.

The goal of this exercise is to get used to practical image processing in python / numpy / scipy. There are numerous Web resources related to python programming; numpy and scipy are mostly well documented and matplotlib, too, comes with numerous tutorials. Play with the code that is provided. The tasks below are rather simple; if you do not have any ideas for how to solve them, just look around for ideas as to how it can be done.

Also, **do NOT use additional third party libraries such as `OpenCV` or `scikit-image` for the coding tasks in this course!**

Why not? Because our goal in this course and its exercises is to learn about theory and practice of image processing on a reasonably foundational level. Regarding practical implementations of image processing algorithms, solutions in C or even assembler would constitute the most foundational level but likely be unreasonable. While working with python / numpy / scipy is still foundational enough, working with libraries such as `OpenCV` or `scikit-image` is definitely not.

Think of it like this: if you train yourself to become a library monkey then what are you going to do when you are supposed to solve a problem for which there is no convenient library function available? How can you be sure that you really learned how to turn mathematics into computer code if all you ever do is stitching together other people's solutions to seemingly related problems?

When handing solutions, always strive for excellence! Your code and results will be checked and need to be convincing, reproducible, and comprehensible. If your solutions meet these criteria and you can demonstrate that they work in practice, it is a *satisfactory* solution.

A *very good* solution requires additional efforts especially w.r.t. to readability of your code. If your code is neither commented nor well structured, your solution is not good! The same holds for your discussion of your results: these should be concise and convincing and demonstrate that you understood what the respective task was all about. Striving for very good solutions should always be your goal!

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 scipy.ndimage as img
```

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

```
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))
```

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 screen, you might use

```
import matplotlib.pyplot as plt

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

task 1.1

the emboss effect

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

`portrait.png`

Read it into a numpy array `arrF` and print the shape of this array to determine its number of rows and columns.

enter your result here . . .

In the lecture, we discussed the idea of "embossing" an image such that the resulting image resembles a copper engraving. In fact, we discussed 4 different methods to accomplish this, namely

```
def embossV1(arrF):
    M, N = arrF.shape
    arrG = np.zeros((M,N))

    for i in range(1,M-1):
        for j in range(1,N-1):
            arrG[i,j] = 128 + arrF[i+1,j+1] - arrF[i-1,j-1]
            arrG[i,j] = np.maximum(0, np.minimum(255, arrG[i,j]))

    return arrG


def embossV2(arrF):
    M, N = arrF.shape
    arrG = np.zeros((M,N))

    arrG[1:M-1,1:N-1] = 128 + arrF[2:,2:] - arrF[:-2,:-2]
    arrG = np.maximum(0, np.minimum(255, arrG))

    return arrG


def embossV3(arrF):
    mask = np.array([[-1, 0,  0],
                     [ 0, 0,  0],
                     [ 0, 0, +1]])

    arrG = 128 + img.correlate(arrF, mask, mode='reflect')
    arrG = np.maximum(0, np.minimum(255, arrG))

    return arrG


def embossV4(arrF):
    arrG = 128 + arrF[2:,2:] - arrF[:-2,:-2]
    arrG[arrG<  0] =   0
    arrG[arrG>255] = 255

    return arrG
```

Apply each of the above methods to `arrF` to produce a corresponding array `arrG` and write each of your results as a PNG image.

Does the result you obtain from *`embossV4`* differ from the results produced by the other methods? It should! Discuss the difference!

enter your discussion here . . .

task 1.2

timing the emboss effect

In the lecture, we also performed experiments to determine the minimum average runtime of our different methods for the *emboss* effect. In this task you are supposed to conduct these experiments on your own machines.

Assuming that you have read an input intensity image into an array `arrF`, you may use the following code snippets for this purpose

```
import timeit, functools


mtds = [embossV1, embossV2, embossV3, embossV4]

nRep =   3
nRun = 100

for mtd in mtds:
    ts = timeit.Timer(functools.partial(mtd, arrF)).repeat(nRep, nRun)
    print (min(ts) / nRun)
```

task 1.2(a): In the `Data` folder, you will find the intensity image

`portrait.png`

Read it into an array `arrF`, print its shape, and run the above timing script.

enter your result here . . .

task 1.2(b): In the `Data` folder, you will find the intensity image

`asterix.png`

Read it into an array `arrF`, print its shape, and run the above timing script.

enter your result here . . .

task 1.2(c): Discuss your results. What do your experiments reveal? Is there any noteworthy difference between the runtimes for the two images? If so, what is the difference? What causes this difference? What does this tell you about image processing in general?

enter your discussion here . . .

task 1.3

working with RGB color images

In the `Data` folder, you will find the color image

`asterixRGB.png`

Read it into an array `arrF` using

```
arrF = imageRead('asterixRGB.png', pilmode='RGB')
```

Print the shape parameters of `arrF` and run the following snippet

```
arrG = np.copy(arrF)
arrG[:,:,0] = 0
```

Write the resulting array `arrG` as a PNG file and enter it here

put your figure here

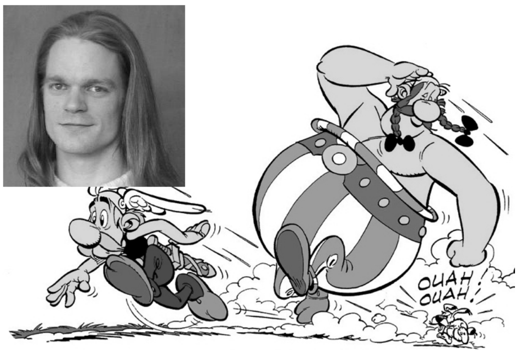
task 1.4

getting used to slicing (part 1)

Read image `asterix.png` into an array `arrF` and image `portrait.png` into an array `arrG`.

Now, create a copy `arrH` of `arrF` and then —without using *for* loops— paste `arrG` into `arrH` such that the upper left corner of `arrG` is at array coordinate $[i, j] = [100, 200]$ in `arrH`. Write your result as a PNG image.

To illustrate how this image should look like, here is the result you would obtain from using $[i, j] = [10, 10]$



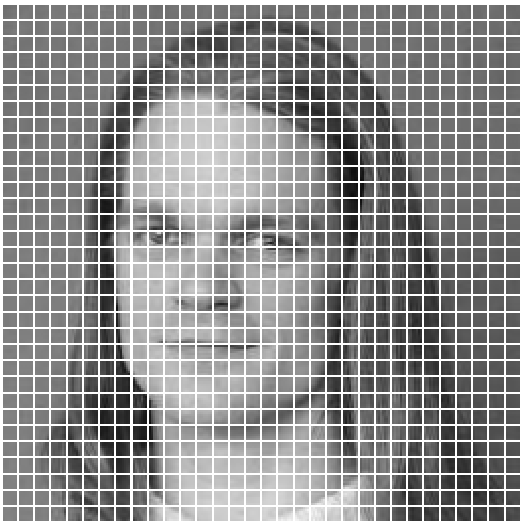
Simply replace the above image with the image you just created.

task 1.5

getting used to slicing (part 2)

Read image `portrait.png` into an array `arrF`. Then —without using `for` loops— set all the intensities of the pixels in in every 16th column and every 16th row of `arrF` to 0. Write your result as a PNG image.

To illustrate how this image should look like, here is the result you would obtain from working with every 8th row and column and setting intensities to 255.



Simply replace the above image with the image you just created.

task 1.6

getting used to meshgrids (part 1)

Read image `portrait.png` into an array `arrF`. Then —without using `for` loops— set the intensities of all its pixels situated within an ellipse of width $2 \cdot 50$ and height $2 \cdot 85$ which is centered at array coordinates $[c_i, c_j] = [128, 110]$ to 255. Write your result as a PNG image.

```
# paste your code here
```

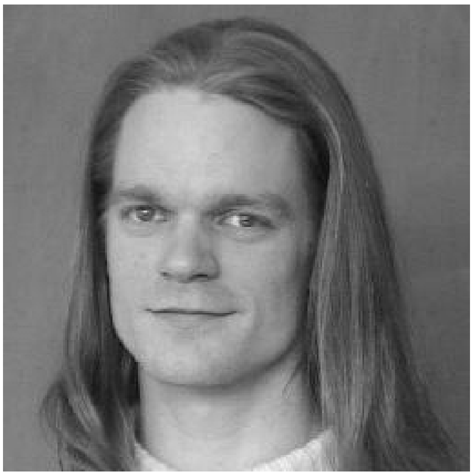
To illustrate how your resulting image should look like, here is the result you would obtain from working with width $2 \cdot 100$, height $2 \cdot 50$ and center point $[128, 128]$

Simply replace the above image with the image you just created.

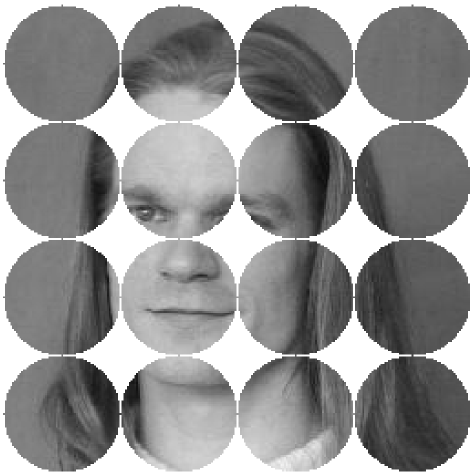
task 1.7

getting used to meshgrids (part 2)

Read image `portrait.png` into an array `arrF`. Then —without using *`for`* loops— create an image array `arrG` whose content looks like shown below



`arrF`



`arrG`

Note: the circles in the above figure have a radius of $r = 32$ pixels. Implement your code such that it produces circles of radius $r \in \{16, 64\}$ pixels.

```
# paste your code here
```

Perform runtime measurements for your code and paste your result here. (When implemented "properly", your solution should be able to process `portrait.png` in only $O(10^{-5})$ seconds . . .)

enter your result here . . .

Finally, enter your resulting images (for $r \in \{16, 64\}$) here

put your figure here

put your figure here

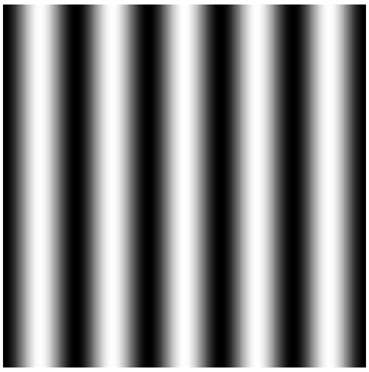
task 1.8

getting used to meshgrids (part 3)

Again —without using *for* loops— create an image of size $M \times N$ where $M = N = 256$ that displays the following function

$$f\,[x, y] = \frac{1}{2}\Big(\sin\big(2\,\pi\,\nu\,\tfrac{x}{N-1} + \tfrac{\pi}{2}\,(N-1)\big) + 1\Big) \cdot 255$$

For instance, when choosing the free parameter $\nu = 5$, your image should look like this



Create such images for $\nu \in \{1, 2, 3, 9\}$ and paste them here

put your figure here

put your figure here

put your figure here

put your figure here

Also, paste your code here

```
# paste your code here
```

task 1.9

getting used to meshgrids (part 9)

Again —without using *for* loops— create an image of size $M \times N$ where $M = N = 256$ that displays the following function

$$f\,[x, y] = \frac{1}{2}\Big(e^{-\frac{4\,y}{M}} \sin\big(2\,\pi\,\nu\,\tfrac{y}{M-1} + \tfrac{\pi}{2}\,(M-1)\big) + 1\Big) \cdot 255$$

For instance, when choosing the free parameter $\nu = 7$, your image should look like this

Create such images for $\nu \in \{5, 9, 11, 13\}$ and paste them here

put your figure here	put your figure here	put your figure here	put your figure here

Also, paste your code here

```
# paste your code here
```

task 1.10

getting used to universal functions

Again —without using `for` loops— implement a method that turns a given intensity image function $f[x,y]$ into another function $g[x,y]$ where

$$g\left[x,y\right] = \cos\left(f\left[x,y\right] \cdot \nu \cdot \tfrac{2\pi}{255}\right) \cdot 127.5 + 127.5$$

Choose $\nu \in \{0.5, 1.0, 1.5, 2.0\}$, apply your method to `portrait.png`, and enter your resulting images here

put your figure here	put your figure here	put your figure here	put your figure here

Also, paste your code here

```
# paste your code here
```