

exercise 5

image filtering

solutions due

until **January 9, 2022** 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 scipy.signal as sig
import scipy.ndimage as img

import timeit, functools
```

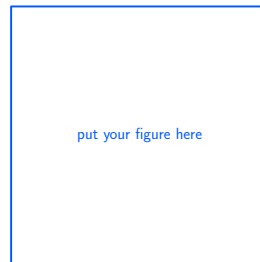
task 5.1

mean filtering in $O(1)$

In the lecture, we discussed the idea of mean filtering. The following function wraps the *scipy ndimage* function *uniform_filter* which implements a mean filter.

```
def meanFilterV1(arrf, m):  
    return img.uniform_filter(arrf, (m,m), mode='constant', cval=0.0)
```

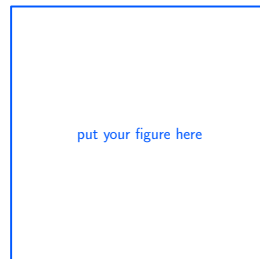
Read the image `portrait.png` into a *numpy* array `arrf`, set the size of the filter mask to `m=5`, and run *meanFilterV1* to produce a smoothed image array `arrg`. Write your result as a PNG file and enter it here



In the lecture, we also discussed the idea of (per pixel) $O(1)$ mean filtering based on integral images. The following function realizes this (in a rather naïve manner from the point of view of efficient array computation).

```
def meanFilterV2(arrf, m):  
    M, N = arrf.shape  
  
    # pad input image with zeros and compute integral image  
    arrF = np.pad(arrf, [m//2, m//2], 'constant', constant_values=[0,0])  
    arrC = np.cumsum(np.cumsum(arrF, axis=1), axis=0)  
  
    arrg = np.zeros((M,N))  
    for i in range(M):  
        for j in range(N):  
            arrg[i,j] = arrC[i+m-1, j+m-1]  
            if i > 0:  
                arrg[i,j] -= arrC[i-1, j+m-1]  
            if j > 0:  
                arrg[i,j] -= arrC[i+m-1, j-1]  
            if i > 0 and j > 0:  
                arrg[i,j] += arrC[i-1, j-1]  
  
    return arrg / m**2
```

To convince yourself that the idea works, once again consider `arrf`, set the size of the filter mask to `m=5`, and run `meanFilterV2` to produce a smoothed image array `arrg`. Write your result as a PNG file and enter it here

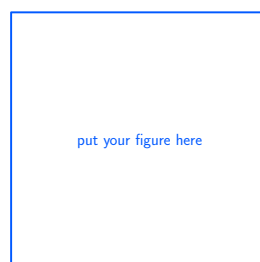


The above implementation of the integral image based mean filter is naïve, because it involves nested *python for* loops. By now, you should be able to see that these *for* can be avoided by making use of *numpy*'s array processing capabilities.

Implement a *for* loop free version of the integral image based mean filter. That is, appropriately complete the following function definition

```
def meanFilterV3(arrf, m):  
    # pad input image with zeros and compute integral image  
    arrF = np.pad(arrf, [m//2, m//2], 'constant', constant_values=[0,0])  
    arrC = np.cumsum(np.cumsum(arrF, axis=1), axis=0)  
    ...
```

To verify that your code works, once again consider `arrf`, set the size of the filter mask to `m=5`, and run `meanFilterV3` to produce a smoothed image array `arrg`. Write your result as a PNG file and enter it here



task 5.2

runtimes of efficient mean filters

In this task, you are supposed to empirically analyze the runtimes of the above functions. Use what you learned in exercises 1 and 2 to perform corresponding runtime measurements and consider the following experimental setting:

Perform your measurements for image `portrait.png`, consider mask sizes $m \in \{5, 11, 23, 47\}$, and determine the average runtimes of functions `meanFilterV1` and `meanFilterV3`. Enter your results (rounded to 5 decimal places) into the following table:

	$m = 5$	$m = 11$	$m = 23$	$m = 47$
<code>meanFilterV1</code>				
<code>meanFilterV3</code>				

Discuss what you observe! What do your runtime measurements tell you about the way the *scipy* developers have apparently implemented the function `uniform_filter`? Enter your discussion here

[enter your discussion here ...](#)

bonus task (for those who are patient)

Should you still not believe that, when it comes to array processing, *python for* loops have to be avoided at all costs, then consider the same setting as above but evaluate the average runtimes of function `meanFilterV2` and enter your results into the following table:

	$m = 5$	$m = 11$	$m = 23$	$m = 47$
<code>meanFilterV2</code>				

task 5.3

bilateral filtering

Implement a bilateral filter

$$h[x, y] = \gamma[x, y] \cdot \sum_{u=-\frac{m}{2}}^{\frac{m}{2}} \sum_{v=-\frac{m}{2}}^{\frac{m}{2}} g_{\rho} \left[f[x-u, y-v] - f[x, y] \right] \cdot G_{\sigma}[u, v] \cdot f[x-u, y-v]$$

where

$$m = \lceil 2.575 \cdot \sigma \rceil \cdot 2 + 1$$

$$g_{\rho}[z] = \exp \left(-\frac{z^2}{2\rho^2} \right)$$

$$G_{\sigma}[u, v] = \exp \left(-\frac{u^2 + v^2}{2\sigma^2} \right)$$

$$\gamma[x, y] = \left(\sum_{u=-\frac{m}{2}}^{\frac{m}{2}} \sum_{v=-\frac{m}{2}}^{\frac{m}{2}} g_{\rho} \left[f[x-u, y-v] - f[x, y] \right] \cdot G_{\sigma}[u, v] \right)^{-1}$$

Note: Efficient implementations of this filter are possible but tricky, see for instance







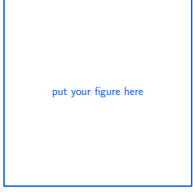
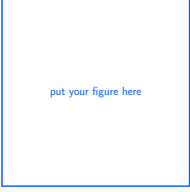

A. Adams, N. Gelfand, J. Olson, and M. Levoy, “Gaussian KD-trees for Fast High-dimensional Filtering”, ACM Transaction on Graphics, 28(3), 2009.

F. Banterle, M. Corsini, P. Cignoni and R. Scopigno, “A Low-Memory, Straightforward and Fast Bilateral Filter Through Subsampling in Spatial Domain”, Computer Graphics Forum, 1(31), 2012.

The methods used in papers like these go beyond what we can study in this course. Having said this, we also point out . . .

Note: At first sight, an implementation of the bilateral filter seems to require many nested *for* loops. When working with languages such as C, FORTRAN, or Julia, this will be no issue. Yet, in *python / numpy*, this will kill performance. However, clever use of *numpy*’s inbuilt array processing capabilities allows for avoiding most of these explicit loops. In fact, if we were willing to accept barely readable code, all of them could be avoided. Since this would be overkill, a suggestion is to try to come with an implementation that involves as few *for* loops as reasonably possible.

Run your implementation on image `portrait.png` using the parameters you need in order to complete the following table

	$\sigma = 3$	$\sigma = 6$	$\sigma = 12$
$\rho = 10$			
$\rho = 20$			
$\rho = 40$			

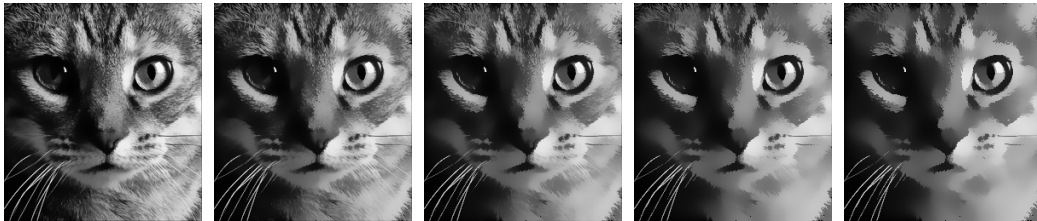
task 5.4

more bilateral filtering

Bilateral filters allow for interesting stylized image effects when they are applied repeatedly. For instance, the script

```
arrF = imageRead('cat.png')  
  
for i in range(5):  
    arrF = bilateral_filter(arrF, rho=20, sig=5)  
    imageWrite(arrF, 't5-4-{num}.png'.format(num=i+1))
```

produces the following sequence of five images



Load image `portrait.png` into array `arrF` and run the above script with your implementation of the bilateral filter. Enter the resulting images below.

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

task 5.5

Deriche's recursive Gaussian filter

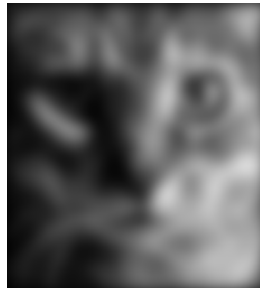
The following function wraps *scipy*'s *ndimage* function *gaussian_filter* which implements a Gaussian filter.

```
def scipy_gaussian_filter(f, sigma):
    return img.gaussian_filter(f, sigma=sigma, mode='constant')
```

If you load image `cat.png` into array `arrF` to compute

```
arrH = scipy_gaussian_filter(arrF, sigma=7.)
```

the resulting image array `arrH` should look like this



In lecture 18, we discussed the following implementation of Deriche's recursive 2D Gaussian filter

```
def deriche_gaussian_filter(f, sigma):
    def filter_rows(x):
        # causal
        xlr = np.copy(x)
        yp = sig.lfilter(np.hstack((ap,0)), np.hstack((1,bp)), xlr[:,m:-m], axis=1)
        # anti-causal
        xrl = np.fliplr(x)
        ym = sig.lfilter(np.hstack((0,am)), np.hstack((1,bm)), xrl[:,m:-m], axis=1)
        ym = np.fliplr(ym)

        return (yp+ym) / (sigma * np.sqrt(2*np.pi))

    ap, am, bp, bm = computeCoefficients(sigma)

    m = 4
    F = np.pad(f, [m,m], mode='constant')

    G = filter_rows(F)
    H = filter_rows(G.T)

    return H.T
```

What we did not discuss in that lecture are the implementation details of function `computeCoefficients` which computes four 1D arrays `ap`, `am`, `bp`, and `bm` of filter coefficients. Implementing this function is your job in this task.

We therefore recall that, given the standard deviation σ of an isotropic Gaussian kernel, the coefficients of the respective causal filter can be computed as

$$\begin{aligned}
 a_0^+ &= \alpha_1 + \alpha_2 \\
 a_1^+ &= e^{-\frac{\gamma_2}{\sigma}} \left(\beta_2 \sin \frac{\omega_2}{\sigma} - (\alpha_2 + 2\alpha_1) \cos \frac{\omega_2}{\sigma} \right) + e^{-\frac{\gamma_1}{\sigma}} \left(\beta_1 \sin \frac{\omega_1}{\sigma} - (2\alpha_2 + \alpha_1) \cos \frac{\omega_1}{\sigma} \right) \\
 a_2^+ &= 2e^{-\frac{\gamma_1+\gamma_2}{\sigma}} \left((\alpha_1 + \alpha_2) \cos \frac{\omega_2}{\sigma} \cos \frac{\omega_1}{\sigma} - \cos \frac{\omega_2}{\sigma} \beta_1 \sin \frac{\omega_1}{\sigma} - \cos \frac{\omega_1}{\sigma} \beta_2 \sin \frac{\omega_2}{\sigma} \right) \\
 &\quad + \alpha_2 e^{-2\frac{\gamma_1}{\sigma}} + \alpha_1 e^{-2\frac{\gamma_2}{\sigma}} \\
 a_3^+ &= e^{-\frac{\gamma_2+2\gamma_1}{\sigma}} \left(\beta_2 \sin \frac{\omega_2}{\sigma} - \alpha_2 \cos \frac{\omega_2}{\sigma} \right) + e^{-\frac{\gamma_1+2\gamma_2}{\sigma}} \left(\beta_1 \sin \frac{\omega_1}{\sigma} - \alpha_1 \cos \frac{\omega_1}{\sigma} \right) \\
 b_1^+ &= -2e^{-\frac{\gamma_2}{\sigma}} \cos \frac{\omega_2}{\sigma} - 2e^{-\frac{\gamma_1}{\sigma}} \cos \frac{\omega_1}{\sigma} \\
 b_2^+ &= 4 \cos \frac{\omega_2}{\sigma} \cos \frac{\omega_1}{\sigma} e^{-\frac{\gamma_1+\gamma_2}{\sigma}} + e^{-2\frac{\gamma_2}{\sigma}} + e^{-2\frac{\gamma_1}{\sigma}} \\
 b_3^+ &= -2 \cos \frac{\omega_1}{\sigma} e^{-\frac{\gamma_1+2\gamma_2}{\sigma}} - 2 \cos \frac{\omega_2}{\sigma} e^{-\frac{\gamma_2+2\gamma_1}{\sigma}} \\
 b_4^+ &= e^{-\frac{2\gamma_1+2\gamma_2}{\sigma}}
 \end{aligned}$$

where the parameters $\alpha_i, \beta_i, \gamma_i, \omega_i$ are given according to the following table

	α_i	β_i	γ_i	ω_i
$i = 1$	1.6800	3.7350	1.7830	0.6318
$i = 2$	-0.6803	-0.2598	1.7230	1.9970

Implement the function `computeCoefficients` such that it returns the coefficients a_m^+ in array `ap` and the coefficients b_m^+ in array `bp`.

Also, recall that, once the causal filter coefficients a_m^+ and b_m^+ have been computed, the coefficient of the respective anti-causal filter can be com-

puted according to

$$\begin{aligned}b_m^- &= b_m^+ \quad \forall m \\a_1^- &= a_1^+ - b_1^+ a_0^+ \\a_2^- &= a_2^+ - b_2^+ a_0^+ \\a_3^- &= a_3^+ - b_3^+ a_0^+ \\a_4^- &= -b_4^+ a_0^+\end{aligned}$$

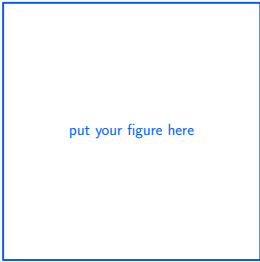
Implement the function `computeCoefficients` such that it returns the coefficients a_m^- in array `am` and the coefficients b_m^- in array `bm`.

Once you have implemented function `computeCoefficients`, you can practically work with `deriche_gaussian_filter`.

To see how it behaves, load image `cat.png` into array `arrF` and compute

```
arrH = deriche_gaussian_filter(arrF, sigma=7.)
```

Write the resulting image array as a PNG file and enter it here.

A blue rectangular box with a thin border, containing the text "put your figure here" in a small, blue, sans-serif font, centered within the box.

task 5.6**timing Deriche's recursive Gaussian filter**

To see what the tedious coding exercise in the previous task will buy you in practice, perform runtime measurements similar to those in task 2 and complete the following table:

	$\sigma = 6$	$\sigma = 12$	$\sigma = 24$	$\sigma = 48$
<i>scipy_gaussian_filter</i>				
<i>deriche_gaussian_filter</i>				
