## exercise 3

**fun with Fourier transforms**

## solutions due

until **November 28, 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.fft as fft
import matplotlib.pyplot as plt
```

For your symbolic computations in task 3.3, the following two definite integral identities may come in handy

$$\int_{-\infty}^{\infty} e^{-a\,y^2}\,dy = \sqrt{\frac{\pi}{a}}$$

$$\int_{-\infty}^{\infty} y\,e^{-a\,y^2}\,dy = 0$$

As integral identities go, these are fairly easy to verify by yourself. For those who are interested in more complicated integrals, there is just one book and one book only you need to know about, namely

> I.S. Gradshteyn and I.M. Ryzhik, *"Table of Integrals, Series, and Products"*, 8th edition, Academic Press, 2014

To stretch the intensity values of a given image array `f` such that they cover the whole range from $0$ to $255$, you may use this function

```
def imageScaleRange(f, vmin=0, vmax=255):
    return np.interp(f, (f.min(), f.max()), (vmin, vmax))
```

**Note:** for now, we will take this function as a given. Later in this course, we will spend quite some time on studyin interpolation techniques. Once we reach that stage, we will understand how `np.interp` works ...

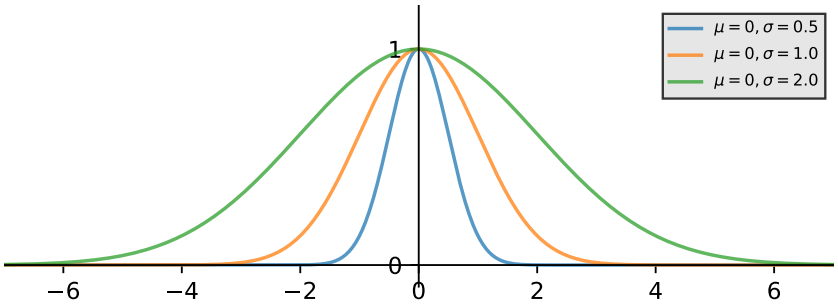
## task 3.1

## uni-variate Gaussians and their derivatives

An unnormalized, uni-variate Gaussian function $f : \mathbb{R} \rightarrow \mathbb{R}$ with location parameter $\mu \in \mathbb{R}$ and scale parameter $\sigma \in \mathbb{R}_+$ is given by

$$f(x \mid \mu, \sigma) = \exp\left(-\frac{(x-\mu)^2}{2\,\sigma^2}\right) = e^{-\frac{1}{2\,\sigma^2}(x-\mu)^2} \tag{1}$$

The following plot was created with parameters $\mu = 0$ and $\sigma \in \{0.5, 1.0, 2.0\}$ and illustrates how such functions look like for $x \in [-7, +7)$




**task 3.1(a):** Compute the (symbolic) derivative of the function in (1) and enter it here

$$\frac{d}{dx} f(x \mid \mu, \sigma) =$$

**task 3.1(b):** Use your result and the same parameters as above to create a plot that shows Gaussian derivatives and enter it here

put your figure here

## task 3.2

## working with libraries for symbolic computing

In this course, we work extensively with the *python* modules *numpy* / *scipy* because they provide numerous functionalities for numerical computing and number crunching. However, during the lectures, we also mentioned software for symbolic computation. Maybe you have come across such software before? Maybe you have heard of Wolfram Mathematica? For *python*, functionalities for symbolic computation are provided by the *sympy* module. To see what it buys you, execute the following snippet

```
from sympy import *

x, mu, sigma = symbols('x mu sigma')

fctF = exp(-(x-mu)**2 / (2*sigma**2))

ddxF = Derivative(fctF, x)

print (ddxF.doit())
```

and enter your result here

enter your result here . . .

**Note:** if you are working within an `ipython` shell or a `jupyter` notebook, you may issue the command

```
init_printing(use_unicode=True)
```

before printing. This will yield a nicer looking, better formatted output on your screen. (However, how to paste the resulting unicode output into a LaTeX document remains for you to figure out . . . your instructor doesn't know this. (I.e. you do *not* have to paste the unicode output at the corresponding location above)).

## task 3.3

## Fourier transforms of Gaussians and their derivatives

**Note:** in what follows, we will (for convenience) simplify the Gaussian in (1) as well as our notation for this function. First of all, we will drop the location parameter $\mu$ (when creating the plots in task 3.1, we did set it to $0$ anyway). Second of all, we will no longer explicitly write the dependence on the scale parameter $\sigma$. In short, in what follows, we are concerned with Gaussians of the form

$$f(x) = \exp\left(-\frac{x^2}{2\,\sigma^2}\right) = e^{-\frac{1}{2\,\sigma^2}x^2} \tag{2}$$

Observe that Gaussian functions are square integrable and therefore have Fourier transforms. In fact, the Fourier transforms of Gaussians centered at $0$, i.e. without location parameter $\mu$, are remarkably simple. While the Fourier transform $F(\omega)$ of a function $f(x)$ is typically a complex valued function, the Fourier transform of a Gaussian centered at $0$ is a real valued function. You can work this out for yourselves ...

**task 3.3(a):** symbolically compute the Fourier transform of the Gaussian in (2); that is complete the following mathematical expression

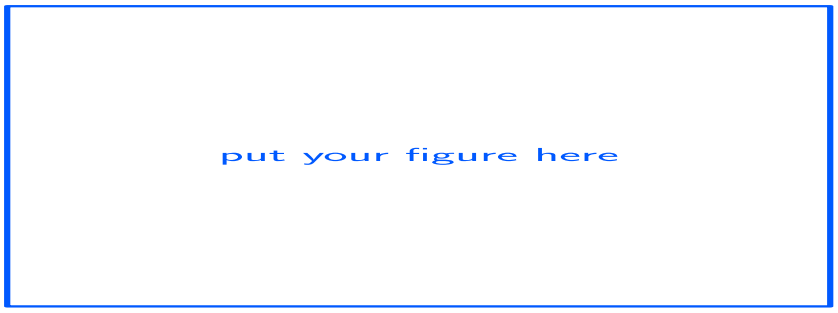
$$\begin{aligned}\mathcal{F}\{f(x)\} = F(\omega) &= \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} e^{-\frac{1}{2\,\sigma^2}x^2}\cdot e^{-i\,\omega\,x}\,dx\\ &=\\ &\vdots\end{aligned}$$

**task 3.3(b):** To make a long story short, above you should have found that the Fourier transform of the Gaussian in (2) amounts to

$$F(\omega) = \sigma \cdot e^{-\frac{1}{2}\sigma^2\omega^2} \tag{3}$$

In other words, the Fourier transform of a zero mean Gaussian is another (scaled) Gaussian!

Use this result to plot the Fourier transforms of zero mean Gaussians with $\sigma \in \{0.5, 1.0, 2.0\}$ over the interval $\omega \in [-7, +7)$ and enter your plot here



**task 3.3(c):** At this point, you might (or better should) be wondering how to compute the Fourier transform of a Gaussian that is not centered at $0$ but at some $\mu \neq 0$ ...

To answer this question, we can use the translation invariance of the Fourier transform we discussed in the lecture.

Note that we can plug $x - \mu$ into (2) to obtain a translated Gaussian

$$f(x - \mu) = e^{-\frac{1}{2\sigma^2}(x-\mu)^2} \tag{4}$$

But the Fourier transform of a translated function is something we have seen before. Use what we discussed in the lecture to compute

$$\mathcal{F}\{f(x - \mu)\} =$$

What do you observe? Is the Fourier transform of a shifted or translated Gaussian still a real valued function? Enter your discussion here

enter your discussion here ...

**task 3.3(d):** now that we know that the Fourier transform of a zero mean Gaussian

$$f(x) = e^{-\frac{1}{2\,\sigma^2}x^2}$$

is given by

$$F(\omega) = \sigma \cdot e^{-\frac{1}{2}\sigma^2\omega^2}$$

we can use the inverse Fourier transform to compute the derivative of $f(x)$, because

$$\frac{d}{dx}\,f(x) = \mathcal{F}^{-1}\left\{\mathcal{F}\left\{\frac{d}{dx}\,f(x)\right\}\right\} = \mathcal{F}^{-1}\big\{i\,\omega\,F(\omega)\big\}$$

Doing this is tedious but entirely possible. So, go ahead and compute

$$\mathcal{F}^{-1}\big\{i\,\omega\,F(\omega)\big\} = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} i\,\omega\,e^{-\frac{1}{2}\,\sigma^2\,\omega^2}\,e^{-i\,\omega\,x}\,d\omega$$

$$=$$

$$\vdots$$

## task 3.4

## Numerically computing Fourier transforms of Gaussians

In all the previous tasks of this exercise sheet, we were concerned with symbolic computations. Now, we will get back to numeric computations.

Our first corresponding task (this one) is a bit unusual in that we provide its solution. However, you really should carefully go through the following because it clarifies a critical issue you must know about.

Above, you saw that the first derivative of a zero mean Gaussian can be computed as

$$\frac{d}{dx} f(x) = \mathcal{F}^{-1}\left\{\mathcal{F}\left\{\frac{d}{dx} f(x)\right\}\right\} = \mathcal{F}^{-1}\{i\,\omega\,F(\omega)\}$$

so let us verify this numerically. To this end, run the following snippet

```
N    = 1024
xmin = -7
xmax = +7
xval = np.linspace(xmin, xmax, N, endpoint=False)
oval = fft.fftfreq(N, (xmax-xmin)/N)

s = 0.5

f = np.exp(-0.5 * xval**2 / s**2)
dfV1 = -xval / s**2 * np.exp(-0.5 * xval**2 / s**2)

F = fft.fft(f)
dfV2 = fft.ifft(1j * oval * F)
```

Given our discussion so far, we would assume that arrays `dfV1` and `dfV2` contain the same values, right? So, go ahead and inspect their content.

What you should find is that they differ. Perhaps this difference is due to the numerical instabilities we discussed in the lecture? Hence, execute the following snippet

```
dfV2 = np.real(dfV2)
```

and inspect `dfV1` and `dfV2` again. Maybe plot their content using something like this

```
plt.plot(xval, dfV1, '-')
plt.plot(xval, dfV2, '-')
plt.show()
```

Do the two resulting curves look similar? Do they differ? If so, then how?

What your inspection should have revealed is that there is still a systematic deviation between the content of array `dfV1` and array `dfV2`. Here is how to remedy this situation

```
dfV2 = fft.ifft(2*np.pi * 1j * oval * F)
dfV2 = np.real(dfV2)
```

Once again, inspect the content of `dfV1` and `dfV2`, for instance, by means of creating yet another plot.

Now you should see that both arrays are basically identical (up to numerical precision). What is going on here? Why would a multiplication by a factor of $2\pi$ cause our practical computation to agree with our theoretical expectation?

The answer has to do with how *we* derived and subsequently defined the Fourier transform

$$F(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x)\, e^{-i\,\omega\,x}\, dx$$

and its inverse

$$f(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} F(\omega)\, e^{i\,\omega\,x}\, d\omega$$

Two key steps during our derivation were the introduction of the *angular frequency* $\omega = \frac{2\pi\,k}{T}$ and the limiting process $T \to \infty$. Note, however, that there is no "canonical" way of writing down the Fourier transform and its inverse. There are only different "conventions". Another common convention is to consider the *simple frequency* $\xi = \frac{k}{T}$ and to let $T \to \infty$.

Considering the definition of $\xi$, we realize that we can write $\omega = 2\pi\,\xi$ and thus $d\omega = 2\pi\,d\xi$. Expressed in terms of $\xi$, the Fourier transform becomes

$$\hat{F}(\xi) = \int_{-\infty}^{\infty} f(x)\, e^{-i\,2\pi\,\xi\,x}\, dx$$

and the corresponding inverse is

$$f(x) = \int_{-\infty}^{\infty} \hat{F}(\xi)\, e^{i\,2\pi\,\xi\,x}\, d\xi$$

With respect to the basic nature of the Fourier transform, the many different notational conventions do not really matter. They are but different ways of writing down the same concept. Yet, they do impact the way statements about properties of the Fourier transform have to be expressed.

For instance, when following the convention we work with in this course, the fact that the Fourier transform turns differentiation into multiplication is expressed as

$$\mathcal{F}\{\tfrac{d}{dx}f(x)\} = i\,\omega\,\mathcal{F}\{f(x)\}$$

However, in the convention we just discussed, it reads

$$\hat{\mathcal{F}}\{\tfrac{d}{dx}f(x)\} = 2\pi\,i\,\omega\,\hat{\mathcal{F}}\{f(x)\}$$

for which we point out the appearance of the factor $2\pi$.

Now consider this: the discrete Fourier transform

$$F[\xi_k] = \sum_{m=0}^{N-1} f[x_m]\; e^{-i\,2\pi\frac{m\,k}{N}}$$

and its inverse

$$f[x_m] = \frac{1}{N}\sum_{k=0}^{N-1} F[\xi_k]\; e^{i\,2\pi\frac{m\,k}{N}}$$

which are computed by the *numpy* functions `fft.fft` and `fft.ifft` are indeed commonly defined w.r.t. to simple frequencies rather than w.r.t. angular frequencies. This is why we have to introduce the factor of $2\pi$ and use the statements

```
dfV2 = fft.ifft(2*np.pi * 1j * oval * F)
dfV2 = np.real(dfV2)
```

in order to numerically obtain the result which theory told us to expect.

What is to be learned from this? It is an unfortunate fact of life in the hard (i.e. math-based) sciences that mathematical notation for more abstract concepts is not always set in stone. We all know this from experience and either did already or still have to learn to live with this. By and large this is no problem and the longer our careers in the sciences last, the

easier it is for us to recognize different notations for what they are, namely nothing but different ways of writing down the same thing. In fact, we are used to something similar in our daily lives: hardly ever will different people we are talking to use the exact same phrases to express the same thing. Nevertheless, we typically understand what they mean.

Here is a danger though: In an age where it has become easy and common place to download software (libraries) from the Web which promise to solve a certain (mathematical) problem, we as programmers must always make sure that our thinking is aligned with the thinking of the people who developed said software (libraries). In other words, we must always verify that a software (library) really computes what we think it computes. Lives can depend on that! For instance, methods such as the Fourier transform and its siblings are important in aerospace engineering. Would you want to be the person whose careless use of a third party library for the design of a more energy efficient wing shape is the cause of a plane crash with hundreds of casualties? If not, then never use software libraries you find on the Web without consideration.

## task 3.5

## visualizing 2D Fourier transforms

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

`clock.jpg`

Read it into a *numpy* array `arrF` and run the following snippet

```
dftF = fft.fft2(arrF)
```

Next, inspect the content of the resulting array (e.g. by means of printing). You will find that `dftF` is a 2D array of complex numbers $z = a + i\,b = r\,e^{i\varphi}$.

How to visualize such an array of complex numbers? The method we usually consider in this course is to consider their *magnitude*

$$r = |z|$$

To compute and inspect these magnitudes, run the following

```
magF = np.abs(dftF)
print (np.max(magF), np.min(magF))
print (magF[0,0])
```

What do you observe? How much larger is the value of `np.max(magF)` than the value of `np.min(magF)`? Only marginally larger? Or are we talking about orders of magnitude larger? Are the values of `np.max(magF)` and `magF[0,0]` the same? What it is that `magF[0,0]` represents?

enter your discussion here . . .

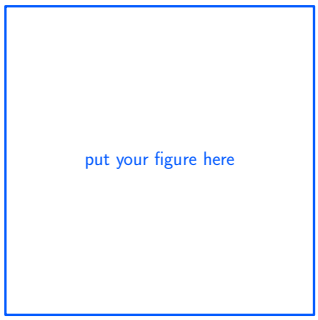
To visualize collections of (positive) numbers of vastly different sizes, it is common to consider their logarithm. So, go ahead and compute

```
logMagF = np.log(magF + 1)
```

Compute the smallest and largest value in array `logMagF` and discuss what you observe

enter your discussion here . . .

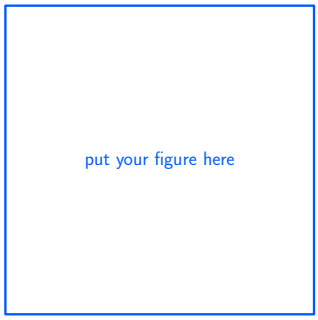
Now write array `logMagF` as a PNG image and paste your result here




Since your result should be a very dark image with hardly any discernible content, you can improve the visualization by scaling its intensity values such that they cover the range from $0$ to $255$. To do so, run

```
scaledLogMagF = imageScaleRange(logMagF)
```

then write your result as a PNG image and paste it here




Another real valued aspect of the complex numbers in array `dftF` which we can visualize are their *phase* values

$$\varphi = \angle z$$

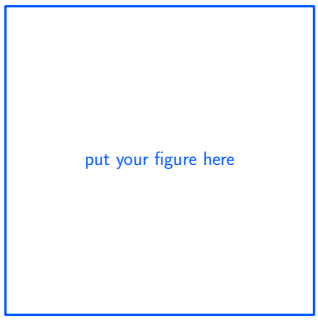
To do so, simply run

```
phsF = np.angle(dftF)
```

Compute the smallest and largest value in array `phsF` and discuss what you observe

enter your discussion here . . .

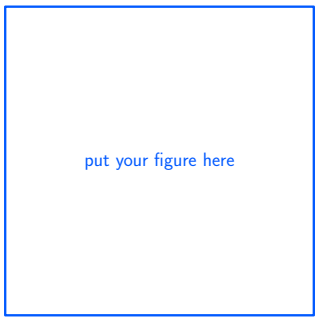
To visualize this phase array, compute `imageScaleRange(phsF)`, write your result as an PNG image, and enter it here




**Note:** the visualization of `logMagF` you computed above looks different than the visualizations we have seen in the lectures. This is because the way in which *numpy*'s `fft` module returns FFT results accommodates the algorithm but not human observers. As a remedy, `fft` provides the function `fftshift` which you may use like this

```
dftFshift = fft.fftshift(fft.fft2(arrF))
```

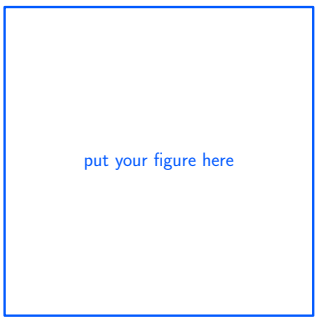
Now compute two arrays `magFshift` and `phsFshift` similar to how you did proceed above. Then apply `imageScaleRange` to both arrays, write them as PNG images and enter your results here

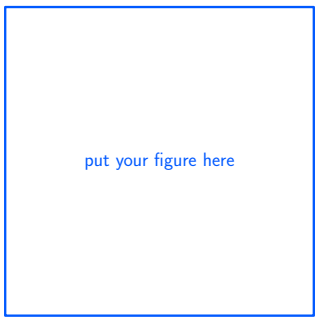



Finally, compute the two arrays

```
arrI = np.real(fft.ifft2(dftF))
arrJ = np.real(fft.ifft2(fft.fftshift(dftFshift)))
```

write them as PNG images and enter your results here



## task 3.6

## exploring the importance of phase

Since the two intensity images

`clock.jpg`
`portrait.png`

in the `Data` folder for this exercise are of the same size ($256 \times 256$ pixels), we can use them for an interesting experiment ...

In what follows, we will refer to the first image as $g[x, y]$ and to the second one as $h[x, y]$.

Write a program that reads the two images $g[x, y]$ and $h[x, y]$ into memory. Then, let you program compute the Fourier transforms

$$G[\xi, \upsilon] = \mathcal{F}\big\{g[x, y]\big\}$$
$$H[\xi, \upsilon] = \mathcal{F}\big\{h[x, y]\big\}$$

In the previous task, you saw that we can think of the resulting complex valued functions as

$$G[\xi, \upsilon] = G_r[\xi, \upsilon] \cdot e^{i\, G_\varphi[\xi, \upsilon]}$$
$$H[\xi, \upsilon] = H_r[\xi, \upsilon] \cdot e^{i\, H_\varphi[\xi, \upsilon]}$$

Hence, let your program compute the magnitude and phase functions $G_r[\xi, \upsilon]$, $G_\varphi[\xi, \upsilon]$, $H_r[\xi, \upsilon]$, and $H_\varphi[\xi, \upsilon]$.

Next, compute a complex valued function

$$P[\xi, \upsilon] = G_r[\xi, \upsilon] \cdot e^{i\, H_\varphi[\xi, \upsilon]}$$

where the magnitude information of $P$ comes from $G$ and the phase information comes form $H$. Also compute a function

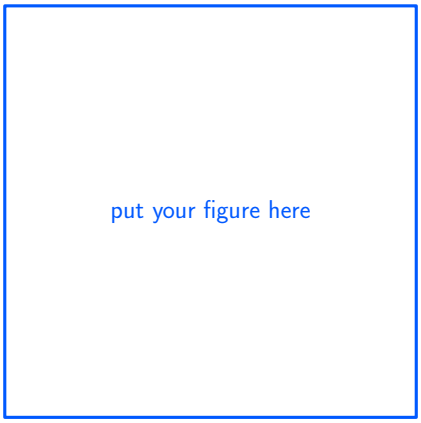
$$Q[\xi, \upsilon] = H_r[\xi, \upsilon] \cdot e^{i\, G_\varphi[\xi, \upsilon]}$$

where the magnitude information of $Q$ comes from $H$ and the phase information comes form $G$.

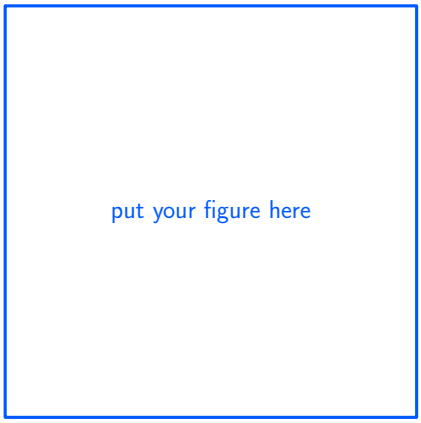
Finally, compute the inverse Fourier transforms

$$p[x, y] = \mathcal{F}^{-1}\big\{P[\xi, \upsilon]\big\}$$
$$q[x, y] = \mathcal{F}^{-1}\big\{Q[\xi, \upsilon]\big\}$$

Save the two resulting intensity image functions to disc (eg. as PNG files) and enter them here

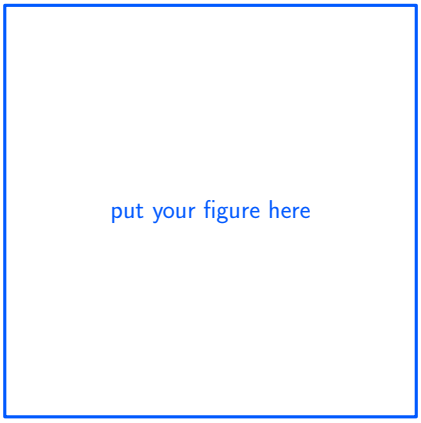



$p[x,y] = \mathcal{F}^{-1}\{G_r[\xi,\upsilon] \cdot e^{i\,H_\varphi[\xi,\upsilon]}\}$

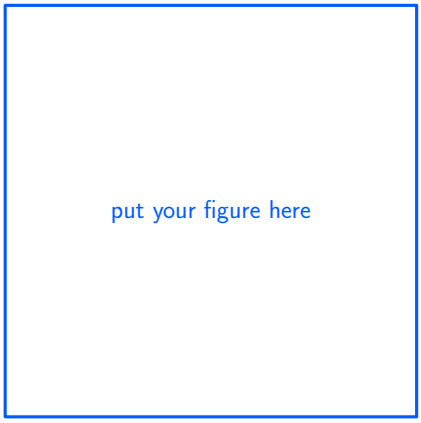



$q[x,y] = \mathcal{F}^{-1}\{H_r[\xi,\upsilon] \cdot e^{i\,G_\varphi[\xi,\upsilon]}\}$

To obtain nicer looking digital images, you may apply `imageScaleRange` from the previous task to $p[x,y]$ and $q[x,y]$. In fact: do this, save your results to disc, and then enter them here




cleaner version of $p[x,y]$




cleaner version of $q[x,y]$

Do you note something peculiar when looking at your results? Discuss what you see and enter your discussion here

enter your discussion here . . .

## task 3.7

## low-pass filtering in the frequency domain

Once again read image

`clock.jpg`

into an array `arrF` and compute

```
dftF = fft.fftshift(fft.fft2(arrF))
```

Now, use what you learned in exercise 1, to set all those entries in array `dftF` to $0$ whose distance to the central element of `dftF` exceeds a value of $r_{\max}$. Recall that the $[i, j]$ coordinates of the central element of `dftF` can be computed as `dftF.shape / 2`.

Store your result in an array `dftH` and compute

```
arrH = np.clip(np.real(fft.ifft2(fft.fftshift(dftH))), 0, 255)
```

Do this for $r_{\max} \in \{1, 5, 10, 50\}$, write the resulting arrays `arrH` as PNG images, and paste them here

put your figure here

put your figure here

put your figure here

put your figure here

What do you observe? How do your results look like? Discuss what you see and enter your discussion here

enter your discussion here . . .