School of Computer Science and Communication

DD2423 Image Analysis and Computer Vision:

Lab 1:
Filtering operations

The goal of this lab is to help you to

- get an understanding of frequency analysis in image data by using the Fourier transform, expressed spatially or in the Fourier domain,
- become familiar with the two-dimensional Fourier transform, by studying and testing its properties in practice,
- learn the relation between the continuous and discrete Fourier transform, and design a smoothing filter for Gaussian convolution,
- understand the effect of different smoothing operations on various kinds of noise,
- understand the effect of smoothing on different image resolutions generated by subsampling,
- get practical experience of differences between synthetic and real data.

The purpose of exercise 1 is for you to become familiar with the characteristics of the Fourier transform. The purpose of exercise 2 is for you to design a Gaussian smoothing filter using the Fourier transform and to understand the relation between continuous and discrete cases. The purpose of exercise 3 is to study different smoothing filters and their effects on noise and subsampling.

Remaining exercises are more problem-oriented and experimental in nature and formulations are thus more concise. If you happen to have difficulties with Python or the environment in general contact the lab assistants during the scheduled lab hours.

Reporting: For you to efficiently report your labs, you need to create a script file that reproduces the experimental results. You should also summarize results and conclusions from your experiments, writing down answers to explicitly stated questions in the answer sheet. For many exercises it is recommended that you create illustrations with multiple images simultaneously shown on screen.

As *prerequisites* to this lab exercise, you should have studied the course material regarding spatial filtering and image restoration. To get started with the lab you need to set the path, in order to access the image samples and functions of the course library.

Python packages The labs rely on a couple of python packages that you need to have installed, before you can begin. Most important is `NumPy` that includes fundamental functions for linear algebra and representation of matrices (and images). Selected functions in `SciPy` will also be used. Finally, we will use `Matplotlib` to display images and graphs. Additional files dedicated to the course are kept in `Functions.py`. At the head of your script for this lab you can include the following lines:

```
import numpy as np
import matplotlib.pyplot as plt
from numpy.fft import fft2, ifft2, fftshift
from scipy.signal import convolve2d
from Functions import *
```

1 Properties of the discrete Fourier transform

1.1 The continuous and discrete Fourier transform

From the definitions used during the lectures the continuous Fourier transform $\hat{f}\colon \mathbb{R}^2 \to \mathbb{C}$ of a two-dimensional signal $f\colon \mathbb{R}^2 \to \mathbb{R}$ is

$$\hat{f}(\omega) = \mathcal{F}_C(f)(\omega) = \int_{x\in\mathbb{R}^2} f(x)\, e^{-i\omega^T x} dx \tag{1}$$

and the Fourier inversion theorem states that

$$f(x) = \mathcal{F}_C^{-1}(\hat{f})(x) = \frac{1}{(2\pi)^2} \int_{\omega\in\mathbb{R}^2} \hat{f}(\omega)\, e^{+i\omega^T x} d\omega. \tag{2}$$

Correspondingly, in the discrete case the Fourier transform $\hat{F}\colon [0..N-1]^2 \to \mathbb{C}$ of a quadratic image $F\colon [0..N-1]^2 \to \mathbb{R}$ of N^2 image elements is defined as

$$\hat{F}(u) = \mathcal{F}_D(F)(u) = \frac{1}{N} \sum_{x\in[0..N-1]^2} F(x)\, e^{\frac{-2\pi i u^T x}{N}} \tag{3}$$

with the corresponding inversion theorem

$$F(x) = \mathcal{F}_D^{-1}(\hat{F})(x) = \frac{1}{N} \sum_{u\in[0..N-1]^2} \hat{F}(u)\, e^{\frac{+2\pi i u^T x}{N}}. \tag{4}$$

There expressions are symmetric with respect to the factor $1/N$ in the Fourier transform and inverse respectively.

1.2 Definition domains

In the continuous case the spatial variable $x = (x_1, x_2)$ and the frequency variable $\omega = (\omega_1, \omega_2)$ are defined in the whole two-dimensional plane, i.e.

$$x, \omega \in \mathbb{R}^2,$$

while the corresponding discrete variables $x = (x_1, x_2)$ and $u = (u_1, u_2)$ are defined in the interval $[0..N-1]^2$, i.e.

$$x_1, x_2, u_1, u_2 \in [0..N-1] = \{0, 1, \ldots, N-1\}.$$

From the periodicity of the basis functions in the discrete case, thus follows that the discrete Fourier transform will become periodic with a period of N. To maintain the consistency with the inversion theorem we similarly have to consider the original signal as periodic with the same period.

By comparing equations (1) and (3) we can relate the continuous angular frequency variable ω to the discrete frequency variable u:

$$\omega_D = 2\pi \frac{u}{N}. \tag{5}$$

Since $u \in [0..N-1]^2$ we may from this relation treat the frequency variable ω_D as being defined in the interval $[0, 2\pi]^2$.

Furthermore, the discrete Fourier transform may similarly be treated as periodic with a period of 2π, and we may change the definition domain to the interval $[-\pi, \pi]^2$ without losing in generality. This corresponds to a *centering* operation in the discrete Fourier transform.

1.3 Basis functions

The Fourier transform can be regarded as a change of basis functions, from compact (discrete) delta functions defined on the Cartesian image domain to complex exponential function with maximum spatial extension. In the discrete case this is even more obvious noting that the discrete Fourier transform can be found by pre- and post-multiplying the image with orthogonal matrices.

Doing so we may consider the complex pixel values in the Fourier transform $\hat{F}$, as components of the discrete image F, with respect to the new basis. Specifically, the basis element corresponding to an image point at coordinates (p, q) in $\hat{F}$ is proportional to

$$\begin{pmatrix} e^{i2\pi\frac{p\cdot 0}{N}} \\ \vdots \\ e^{i2\pi\frac{p\cdot(N-1)}{N}} \end{pmatrix} \begin{pmatrix} e^{i2\pi\frac{q\cdot 0}{N}} & \cdots & e^{i2\pi\frac{q\cdot(N-1)}{N}} \end{pmatrix}$$

These basis vectors are the discrete counterparts to the complex exponential functions

$$e_w(x) = e^{i\omega^T x}$$

You may convince yourself that this is true by calculating the inverse discrete Fourier transform of an image $\hat{F}$, that is zero everywhere except for a single point (p, q) at which the value is one. For such an image the expansion of $\hat{F}$ only contains one term, i.e. the basis vector given by the index (p, q). To visualize this, define a 128×128 pixel image, with elements defined as

```
Fhat = np.zeros((128, 128))
Fhat[p, q] = 1
```

with $(p, q) = (5, 9)$. Display this image on screen using `showgrey()`, which is a convenient function in `Functions.py`, which rescales the image data to make the results easier to see. If second argument of `showgrey()` is set to `display = True`, which is the default, an image will be immediately displayed. Otherwise, it will not be displayed until `plt.show()` is called.

Thereafter, compute the inverse discrete Fourier transform of the image with

```
F = ifft2(Fhat)
```

and look at its real and imaginary parts, as well as its magnitude and phase, by writing

```
Fabsmax = np.max(np.abs(F))
showgrey(np.real(F), True, 64, -Fabsmax, Fabsmax)
showgrey(np.imag(F), True, 64, -Fabsmax, Fabsmax)
showgrey(np.abs(F), True, 64, -Fabsmax, Fabsmax)
showgrey(np.angle(F), True, 64, -np.pi, np.pi)
```

A clever way of organizing the experiment is by writing a small function `fftwave()`, that assembles the resulting images into a single figure in `Python` using `Matplotlib`, with the help of the command `add_subplot()` and illustrative headings.[1] Please use this function as of prototype example of how the experimentation in later exercises can be arranged.

[1] A template of such a function can be found in `fftwave.m` in the course library.

```
import numpy as np
import matplotlib.pyplot as plt
from Functions import showgrey

def fftwave(u, v, sz = 128):
    Fhat = np.zeros([sz, sz])
    Fhat[u, v] = 1

    F = np.fft.ifft2(Fhat)
    Fabsmax = np.max(np.abs(F))

    f = plt.figure()
    f.subplots_adjust(wspace=0.2, hspace=0.4)
    plt.rc('axes', titlesize=10)

    a1 = f.add_subplot(3, 2, 1)
    showgrey(Fhat, False)
    a1.title.set_text("Fhat: (u, v) = (%d, %d)" % (u, v))

    # What is done by these instructions?
    if u < sz/2:
        uc = u
    else:
        uc = u - sz
    if v < sz/2:
        vc = v
    else:
        vc = v - sz

    wavelength = 0.0                # Replace by correct expression
    amplitude  = 0.0                # Replace by correct expression

    a2 = f.add_subplot(3, 2, 2)
    showgrey(np.fft.fftshift(Fhat), False)
    a2.title.set_text("centered Fhat: (uc, vc) = (%d, %d)" % (uc, vc))

    a3 = f.add_subplot(3, 2, 3)
    showgrey(np.real(F), False, 64, -Fabsmax, Fabsmax)
    a3.title.set_text("real(F)")

    a4 = f.add_subplot(3, 2, 4)
    showgrey(np.imag(F), False, 64, -Fabsmax, Fabsmax)
    a4.title.set_text("imag(F)")

    a5 = f.add_subplot(3, 2, 5)
    showgrey(np.abs(F), False, 64, -Fabsmax, Fabsmax)
    a5.title.set_text("abs(F) (amplitude %f)" % amplitude)

    a6 = f.add_subplot(3, 2, 6)
    showgrey(np.angle(F), False, 64, -np.pi, np.pi)
    a6.title.set_text("angle(F) (wavelength %f)" % wavelength)

    plt.show()
```

Question 1: Repeat this exercise with the coordinates p and q set to $(5, 9)$, $(9, 5)$, $(17, 9)$, $(17, 121)$, $(5, 1)$ and $(125, 1)$ respectively. What do you observe?

Question 2: Explain how a position (p, q) in the Fourier domain will be projected as a sine wave in the spatial domain. Illustrate with a figure.

Question 3: How large is the amplitude? Write down the expression derived from Equation (4) in these notes. Complement the code (variable `amplitude`) accordingly.

Question 4: How does the direction and length of the sine wave depend on p and q? Write down the explicit expression that can be found in the lecture notes. Complement the code (variable `wavelength`) accordingly.

Question 5: What happens when we pass the point in the center and either p or q exceeds half the image size? Explain and illustrate graphically with a figure!

Question 6: What is the purpose of the instructions following the question: `What is done by these instructions?` in the code?

1.4 Linearity

Define some rectangular shaped 128×128 pixel test images by

```
F = np.concatenate([np.zeros((56,128)), np.ones((16,128)), np.zeros((56,128))])
G = F.T
H = F + 2*G
```

and display them with `showgrey()`. Then compute the discrete Fourier transform of the images by writing

```
Fhat = fft2(F)
Ghat = fft2(G)
Hhat = fft2(H)
```

and show their Fourier spectra with

```
showgrey(np.log(1 + np.abs(Fhat)))
showgrey(np.log(1 + np.abs(Ghat)))
showgrey(np.log(1 + np.abs(Hhat)))
```

Also try to run the following

```
showgrey(np.log(1 + np.abs(fftshift(Hhat))))
```

and explain why the `fftshift()` and the `log()` commands are used here. Another way of achieving the same effect is using the function `showfs()` located in the course library.

Question 7: Why are these Fourier spectra concentrated to the borders of the images? Can you give a mathematical interpretation? Hint: think of the frequencies in the source image and consider the resulting image as a Fourier transform applied to a 2D function. It might be easier to analyze each dimension separately!

Question 8: Why is the logarithm function applied?

Question 9: What conclusions can be drawn regarding linearity? From your observations can you derive a mathematical expression in the general case?

1.5 Multiplication

With F and G as previously defined

```
F = np.concatenate([np.zeros((56,128)), np.ones((16,128)), np.zeros((56,128))])
G = F.T
```

Try the following commands

```
showgrey(F * G)
showfs(fft2(F * G))
```

and explain the results. The notation `F * G` in `NumPy` refers to point-wise multiplication of corresponding matrix elements, while `F @ G` represents a matrix multiplication.

Question 10: Are there any other ways to compute the last image? Remember what multiplication in Fourier domain equals to in the spatial domain! Try to perform these alternative computations in practice, if you want.

1.6 Scaling

Define a test image

```
F = np.concatenate([np.zeros((60,128)), np.ones((8,128)), np.zeros((60,128))]) * \
    np.concatenate([np.zeros((128,48)), np.ones((128,32)), np.zeros((128,48))], axis=1)
```

and display it with `showgrey()`. Determine the discrete Fourier transform and look at the magnitude with `showfs()`.

Question 11: What conclusions can be drawn from comparing the results with those in the previous exercise? See how the source images have changed and analyze the effects of scaling.

1.7 Rotation

Rotate `F` by for example *alpha*= 30° and show the results with

```
G = rot(F, alpha)
showgrey(G)
```

Then calculate the discrete Fourier transform of the rotated image with

```
Ghat = fft2(G)
```

and display the results with `showfs()`. Do you recognize it? If you activate a new figure with `plt.figure(num)` and display each image with `display = False` and end with `plt.show()`, you can keep all images open at the same time, which makes it easier to see the difference.

Finally, rotate the spectrum back by the same angle with

```
Hhat = rot(fftshift(Ghat), -alpha)
```

and show the results by writing

```
showgrey(log(1 + abs(Hhat)))
```

Assemble the original images and their Fourier spectra into the same figure with `plt.add_subplot()`. Vary the angle with different values of *alpha*, for example $alpha = 30°, 45°, 60°, 90°$ and illustrate with at least one case.

Question 12: What can be said about possible similarities and differences? Hint: think of the frequencies and how they are affected by the rotation.

1.8 Information in Fourier phase and magnitude

Above we have primarily used the magnitude of the Fourier transform when we have visualized the transform as an image. This visualization technique is also what dominates in literature. As an illustration of the limitations of this way of visualizing, we will in this section devote ourselves to a bit of image manipulation, where we simply *replace* the power spectrum for a given image f with a power spectrum of the form [2]

$$|\hat{f}(\omega)|^2 = \frac{1}{a + |\omega|^2} \tag{6}$$

In the course library, there is a function `pow2image()` that performs this (nonlinear) function. You can take a look at the code in `Functions.py`.

Apply this function on for example the following images `phonecalc128`, `few128`, `nallo128` and study the results on screen (for small values of $a \approx 10^{-3}$). An image can be loaded like this:

```
img = np.load("Images-npy/phonecalc128.npy")
```

As a comparison you should apply the function `randphaseimage()` that keeps the magnitude of the Fourier transform, but replaces the phase information with a random distribution. What conclusions can be drawn from the comparison?

Question 13: What information is contained in the phase and in the magnitude of the Fourier transform?

2 Gaussian convolution implemented via FFT

Here we study how a smoothing filter can be designed using the Fourier transform, in order to perform a 2D Gaussian convolution on images.

2.1 Continuous Gaussian convolution

Gaussian convolution means that we convolve a given image f_{in} with a Gaussian kernel. In the continuous case the output image f_{ut} is given by

$$f_{ut}(x, y) = \int_{\xi=-\infty}^{\infty} \int_{\eta=-\infty}^{\infty} f_{in}(x - \xi, y - \eta)\, g(\xi, \eta;\ t)\, d\xi\, d\eta \tag{7}$$

[2] For continuous signals (and in the case $a = 0$) this form of Fourier spectrum can be derived as an idealized model of images that contain different types of image structures distributed on all scales. (The purpose of parameter a is simply to avoid division by zero.)

where

$$g(x, y;\ t) = \frac{1}{2\pi t} e^{-(x^2+y^2)/(2t)}. \tag{8}$$

Correspondingly, in the Fourier domain

$$\hat{f}_{ut}(\omega_1, \omega_2) = \hat{g}(\omega_1, \omega_2;\ t)\, \hat{f}_{in}(\omega_1, \omega_2) \tag{9}$$

where $\hat{f}_{ut}$ and $\hat{f}_{in}$ are the Fourier transforms of f_{ut} and f_{in} respectively, and $\hat{g}(\cdot, \cdot;\ t)$ denotes the Fourier transform of the Gaussian function, that is

$$\hat{g}(\omega_1, \omega_2;\ t) = \int_{\omega_1=-\infty}^{\infty} \int_{\omega_2=-\infty}^{\infty} g(x, y;\ t)\, e^{-i(\omega_1 x + \omega_2 y)}\, dx\, dy = e^{-(\omega_1^2+\omega_2^2)\, t/2}. \tag{10}$$

2.2 Discretization in the spatial domain

To discretize the Gaussian convolution, we may either choose to discretize the convolution operation (7) in the spatial domain, or proceed by discretizing the Fourier transform of the Gaussian function (10). In this exercise we will try the first method that is the *spatial domain.*

If we discretize the integral in (7) with the trapezoidal rule and set the step length to one (corresponding to a unit distance between neighboring image elements), we get

$$f_{ut}(x, y) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} f_{in}(x - m, y - m)\, g(m, n;\ t) \tag{11}$$

which corresponds to a discrete convolution with a sampled version of the Gaussian function. To perform this operation in practice, we may thus use either one of the following methods:

With spatial discretization and spatial convolution:

1. Generate a filter based on a sampled version of the Gaussian function.
2. Convolve the image with this filter using the `SciPy` function `convolve2d()`.

This method is probably the most straight-forward one. Such explicit convolution in the spatial domain generally works very well, if one exploits the fact that the Gaussian function is separable. This method is particularly suitable when the variance of the Gaussian kernel is small, and the kernel is truncated to a small filter of compact support. However, since the computational work grows linearly with the number of non-zero elements in the filter, this method can be computationally expensive for large values of t, in particular if the separability is not exploited. In practice, this is not the case with `convolve2d()`.

With spatial discretization and convolution via FFT:

1. Generate a filter based on a sampled version of the Gaussian function.
2. Fourier transform the original image and the Gaussian filter.
3. Multiply the Fourier transforms.
4. Invert the resulting Fourier transform.

This method has the characteristic that it is essentially independent of the size of the Gaussian kernel. In practice, you always generate a Gaussian kernel of the same size as the original image to be filtered; we assume here that this is a power of two and large enough for the shape of the Gaussian kernel to be kept within the given image size. Specifically, with the image processing system we use, the second method is facilitated by an efficient implementation of FFT available in `NumPy`. For the above mentioned reasons the last method is in this case the preferable one, even if it involves more steps.

2.3 Filtering procedure

Write a function that performs Gaussian filtering using FFT!

Write one function

```
gaussfft(pic, t)
```

that, **using the fast Fourier transform**, convolves the image `pic` with a two-dimensional Gaussian function of arbitrary variance t via a discretization of the Gaussian function in the spatial domain, as described in steps 1-4 above. In `NumPy` you may simply generate a coordinate system with the function `meshgrid()` and move the position of the origin with `fftshift()`.

Proposed test procedure: When you perform this operation you might want to make sure that your convolution function has an approximately correct behavior by analyzing its impulse response. Hence, generate the impulse response explicitly

```
psf = gaussfft(deltafcn(128, 128), t)
```

for different values of t (e.g. $t = 0.1$, 0.3, 1.0, 10.0 and 100.0). Look at the result and compute also the (spatial) covariance matrix of the Gaussian function with

```
variance(psf)
```

Compare your results to the ideal continuous case, for which the covariance matrix is t multiplied by the identity matrix.

$$C(g(\cdot, \cdot;\, t)) = t \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{12}$$

If you find it difficult to get this to properly work properly, a routine `discgaussfft()`, that convolves an image with the discrete version of the Gaussian kernel, is provided in the course library. Your function should be different and your results are not expected to be exactly the same!

Question 14: Show the impulse response and variance for the above mentioned t-values. What are the variances of your discretized Gaussian kernel for $t = 0.1$, 0.3, 1.0, 10.0 and 100.0?

Question 15: Are the results different from or similar to the estimated variance? How does the result correspond to the ideal continuous case? Lead: think of the relation between spatial and Fourier domains for different values of t.

Question 16: Convolve a couple of images with Gaussian functions of different variances (like $t =$ 1.0, 4.0, 16.0, 64.0 and 256.0) and present your results. What effects can you observe?

3 Smoothing

3.1 Smoothing of noisy data

Load the image `office256` from the image database of the course

```
office = np.load("Images-npy/office256.npy")
```

and create two noisy images using the functions `gaussnoise()` and `sapnoise()` that exist in the course function library):

```
add = gaussnoise(office, 16)
sap = sapnoise(office, 0.1, 255)
```

Show the images on screen with `showgrey()`. What kind of noise do they represent?

Try to reduce the noise in the images `add` and `sap` with

- Gaussian smoothing (using function `gaussfft()` that you wrote in previous exercise, alternatively the function `discgaussfft()` available in the course library)
- Median filtering (using the function `medfilt()` in the course library)
- Ideal low-pass filtering (using the function `ideal()` in the course library)

Try suitable values of the parameters of each filter respectively (standard deviation for the Gaussian filter, window size for the median filter, cut-off frequency for the ideal low-pass filter).

Question 17: What are the positive and negative effects for each type of filter? Describe what you observe and name the effects that you recognize. How do the results depend on the filter parameters? Illustrate with `Matplotlib` figure(s).

Question 18: What conclusions can you draw from comparing the results of the respective methods?

3.2 Smoothing and subsampling

We will now study the effect of smoothing when applied to images sampled at lower resolutions. There are many applications to subsampling motivated by performance or storage. Generate a series of subsampled versions of `phonecalc256`, keeping 1 pixel out of 2 (using the `rawsubsample()` function in the course library). Alternatively, smooth the original image before subsampling, using the Gaussian filter (`gaussfft()`) or the ideal low-pass filter (`ideal()`). To display the results you can use this template:

```
img = np.load("Images-npy/phonecalc256.npy")
smoothimg = img
N = 5
f = plt.figure()
f.subplots_adjust(wspace=0, hspace=0)
for i in range(N):
    if i>0: # generate subsampled versions
        img = rawsubsample(img)
        smoothimg = # <call_your_filter_here>(smoothimg, <params>)
        smoothimg = rawsubsample(smoothimg)
    f.add_subplot(2, N, i + 1)
    showgrey(img, False)
    f.add_subplot(2, N, i + N + 1)
    showgrey(smoothimg, False)
plt.show()
```

As the resolution of the image decreases, observe the changes in the original image and its smoothed variants. Try the Gaussian and ideal low-pass filters and find suitable values for their parameters, that preserve most of the characteristics/quality with respect to the original image (iteration $i = 1$).

Question 19: What effects do you observe when subsampling the original image and the smoothed variants? Illustrate both filters with the best results found for iteration $i = 4$.

Question 20: What conclusions can you draw regarding the effects of smoothing when combined with subsampling? Hint: think in terms of frequencies and side effects.

Laboration 1 in DD2423 Image Analysis and Computer Vision

.....

Student's personal number and name (filled in by student)

.....

Approved on (date) Course assistant