Exercise 1: Basic Image Processing

Due: 28.10.2021

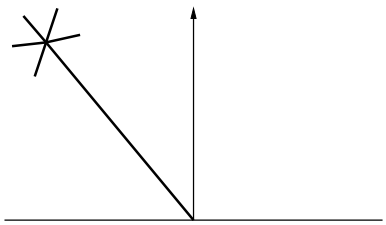
Please note that Exercise 1 is long and may require all five sessions to complete. To save time, first have a brief but complete read of this handout, which describes comprehensively what you have to do. Don't hesitate to ask for help from the assistants!
Please answer the pen-and-paper questions of the exercise on this answer sheet. In case of insufficient space, use your own paper. The order of the three topics of the exercise corresponds to the order these topics are covered in the lecture. The code templates provided for the programming parts are compatible both for Python 2.7 and Python 3.7.
Note: In case you are working on your own PCs, instructions on installing necessary packages can be found at `https://docs.google.com/presentation/d/1Oaus5UefwtIlYKXlFMqCC6nfY3Iv7iVK-KVm2rfsdZE/edit#slide=id.p`.

1 Image Acquisition

1.1 Theoretical Exercises

1.1.1 Light and Matter

a) Complete the following figure such that it illustrates reflection on a Lambertian surface. Draw the direction(s) of the reflected light beam.

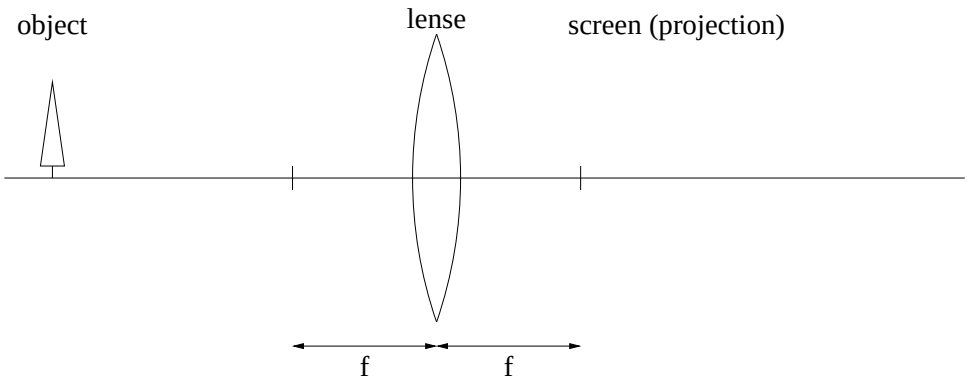


1.1.2 Image Acquisition / Camera Projection

1.1.2.1 Acquisition of an image with a convex lens

You want to take a picture of an object with the following simple camera model (a lens of focal length f and a screen). Where should you place the screen in order to have a sharp image? How large will the object be on the screen?

Draw and label the position of the screen and the size of the object onto the sketch below:



1.1.2.2 Minimal distance between object and screen

Using Gauss's formula for thin lenses,

$$\frac{1}{Z_o} - \frac{1}{Z_i} = \frac{1}{f},$$

where Z_o is the distance between the object and the center of the lens, $-Z_i$ is the distance between the screen and the center of the lens (Z_i is negative) and f is the focal length, compute the minimum distance between the object and the screen for the above camera setup. Assume that the object is in focus and that f is constant. To simplify the calculation, you may introduce positive variables for both the object distance $g = Z_o$ and the screen distance $b = -Z_i$. Hence, you have to minimize $g + b$.

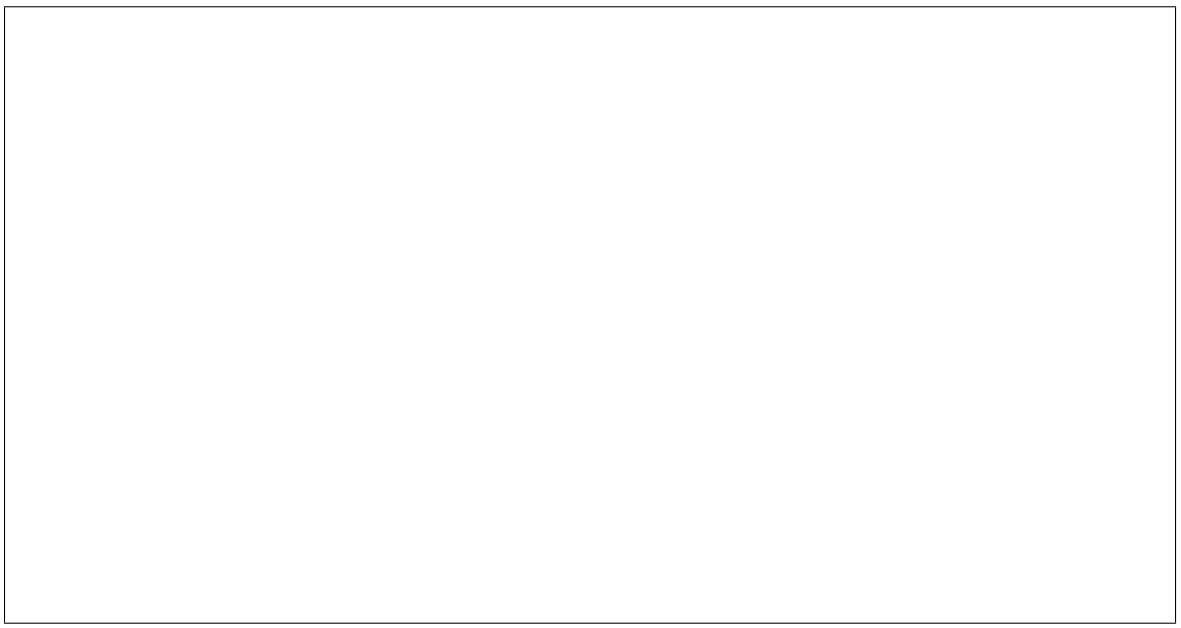
1.1.2.3 Camera projection

You have a (pinhole-) camera, for which the internal calibration parameters are already known.
Now you take a picture with this camera, in which you know the exact position of N points in the "World Coordinate System".
How large must N be to determine all *external* calibration parameters?

Try to answer the question in two ways:

a) Consider how many unknowns this problem has, and how many equations you obtain from each point. The problem is solved if there are at least as many equations as there are unknowns.

b) Consider which degrees of freedom are lost for each additionally considered point. If no more degrees of freedom exist, i.e. the location and orientation of the camera are fixed, then the camera is externally calibrated.



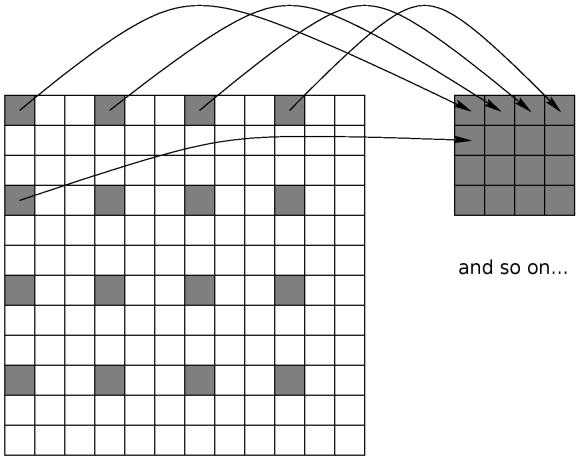
1.2 Practical Exercise: Sampling

This part of the exercise addresses the acquisition of digital, discretely sampled images and helps you to understand how to access individual pixels in a given input image by treating spatial sampling techniques. To complete this part, you may use either the Python script template `1_sample.py` or the Jupyter notebook template `1_sample.ipynb`, which accompany this handout. **Before you start coding for this part, please read carefully the rest of the description below! It includes an exact definition of what your program is expected to do, plus related questions you have to answer in writing.**

Digital images are sampled representations of continuous space. Each sample in the image is a discrete entity known as a pixel. This practical exercise explores the properties of spatial sampling by considering an already sampled input image and resampling it with a lower sampling rate, which is known as *downsampling* or *decimation*.

1.2.1 Naive Downsampling

Starting from the code provided in either of the aforementioned templates, implement a function that extracts every n-th pixel from a given input image (where n is an integer) both in the horizontal and vertical direction to generate an output downsampled image for which the total number of pixels is reduced by a factor of approximately n^2. Parameter n is known as the *decimation factor*. For $n = 3$, the following output image (right part) will result from the input image (left part).



Using your function, experiment with various decimation factors n on the images `window.jpg` and `bricks.jpg`, which accompany this handout.
What happens if you resample an input image containing fine regular structures (e.g. `bricks.jpg`) with a much lower sampling rate, i.e., using very large n?

What is aliasing and why does it occur?

The higher the decimation factor n, the lower the sampling rate. Without modifying the input image, what is the highest decimation factor n_{max} that one can apply to a given image without any information loss incurred?

1.2.2 Pre-filtering for Anti-aliasing

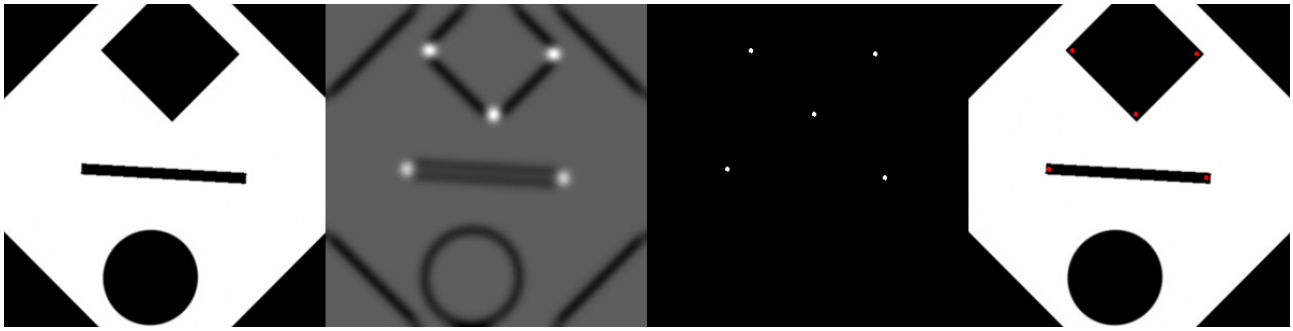
A standard *anti-aliasing* step is to *pre-filter* the input image with a low-pass filter, in order to suppress its high-frequency content, before downsampling it. In this way, it is possible to use quite large decimation factors n and still obtain a meaningful output. Starting from the code provided in either of the aforementioned templates, implement a function that filters the input image with a Gaussian filter, which is an approximation of an ideal low-pass filter. Using a proper built-in Python function to achieve this is highly recommended (hint: an existing import statement in the templates already points you to this function). After that, experiment with large decimation factors n for downsampling the image `bricks.jpg` and compare to the case of no pre-filtering.

How is the scale parameter σ of the Gaussian filter related to the bandwidth of an ideal low-pass filter?

2 Harris Corner Detection

Visual features are attributes or characteristics of an image. Feature extraction is a common step used in many computer vision tasks such as object detection, stereo correspondence, motion extraction, etc. In this exercise, you will explore two types of low-level feature points: **corners** and **edges**.

The Harris corner detector finds corners in an image. Conceptually, corner detection can be thought of as an auto-correlation of an image patch. Consider a window which slides over an image patch. If the image patch is constant or "flat", then there will be little to no intensity changes in the window. If the image patch has an edge, then there will be no intensity changes in the window along the direction of the edge. If a corner is present, however, then there will be a strong intensity change in the window regardless of the direction. **Before you start coding for this part, please read carefully the rest of the description below! It includes the mathematical background of the Harris detector, an exact definition of what your program is expected to do, plus related questions you have to answer in writing.**



Based on the above observation, the formulation of the Harris corner detector on grayscale images is as follows. Denote the grayscale intensity function for the input image with I. First, I_x and I_y, i.e. the partial derivatives of I, are approximated. These partial derivatives can also be thought of as grayscale images. The approximation is done either by employing a two-dimensional isotropic Gaussian filter G_{σ_d} and convolving its closed-form partial derivatives with I (taking advantage of the commutativity of linear operators), or by applying the horizontal and vertical Sobel masks to I. Second, the second-order moments of partial intensity derivatives are computed, i.e., ${I_x}^2$, ${I_y}^2$, and $I_x \cdot I_y$. Third, each of these second-order moments is smoothed isotropically with another two-dimensional Gaussian filter G_{σ_w}, where in principle $\sigma_w > \sigma_d$. Denote the smoothed versions of the second-order moments with $\overline{{I_x}^2}$, $\overline{{I_y}^2}$, and $\overline{I_x \cdot I_y}$. Then, the auto-correlation matrix $\mathbf{A}_{i,j}$ at every pixel (i, j) can be conceptually defined as

$$\mathbf{A}_{i,j} = \begin{bmatrix} \overline{{I_x}^2}(i,j) & \overline{I_x \cdot I_y}(i,j) \\ \overline{I_x \cdot I_y}(i,j) & \overline{{I_y}^2}(i,j) \end{bmatrix}.$$

Using $\mathbf{A}_{i,j}$ to simplify notation, the fourth step is the calculation of the Harris response $H_{i,j} = \det(\mathbf{A}_{i,j}) - k(\mathrm{trace}(\mathbf{A}_{i,j}))^2$, where k is a positive constant. The Harris response $H_{i,j}$ at pixel (i, j) can roughly be thought of as a measure of "cornerness", with negative values indicating edges, close-to-zero values indicating flat regions, and large positive values indicating corners. We can denote the Harris response for the image I as a grayscale image H, where the value $H_{i,j} = H(i, j)$ denotes the Harris response at pixel (i, j). In order to determine the corner locations in I as a set of isolated pixels, the fifth and final step is to apply non-maximum suppression to the Harris response H, so that a pixel with coordinates (i, j) is identified as a corner location if it meets both of the following criteria: $H_{i,j} \geq \theta$ and $H_{i,j} = \max_{(m,n) \in \mathcal{N}(i,j)} \{H_{m,n}\}$, where θ is a positive threshold parameter and $\mathcal{N}(i, j)$ denotes the set of neighboring pixels to (i, j), including itself.

2.1 Theoretical Exercises

- Is the Harris corner detector robust with respect to uniform intensity changes in the image, i.e. uniform intensity shifts and intensity scalings? Why or why not?

- Is the Harris corner detector robust with respect to rotation? Why or why not?

2.2 Practical Exercise

Implement a Harris corner detector for grayscale images. You may use either the Python script template `1_harris.py` or the Jupyter template `1_harris.ipynb`, which accompany this handout. Test your implementation on the provided images `CircleLineRect.png` and `zurlim.png`. Your visual results for each input image I should include: (1) the computed Harris response H as a grayscale image of the same dimensions as I, and (2) the detected corner pixels as red points superimposed on I.

2.2.1 Effect of Rotation on Harris Output

After obtaining the corner detection results for the original images, try different values for the `rot_angle` parameter defined in the code template (originally set to 0) to rotate the input image before applying the Harris corner detector you have implemented in the previous part. Observe the results on the rotated images. How does the Harris corner detector behave with regard to rotations of the input?

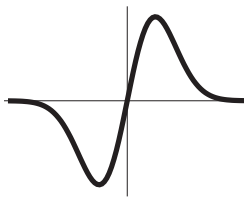
3 Canny Edge Detection

The Canny filter determines edges based on searching for local minima or maxima in the approximated first-order derivative of a signal.

3.1 Theoretical Exercises

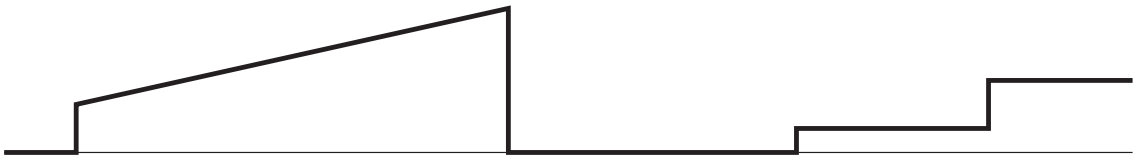
The final version of the Canny filter is as follows:

$$\frac{d}{dx}G(x) = \frac{d}{dx}\left(\frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{1}{2}\left(\frac{x}{\sigma}\right)^2}\right)$$



mirrored $\frac{d}{dx}G$-Filter

a) Apply the Canny-Filter to the following profile in order to detect the edges. What is the magnitude of the filter response and the polarity at the edges? (draw a sketch).

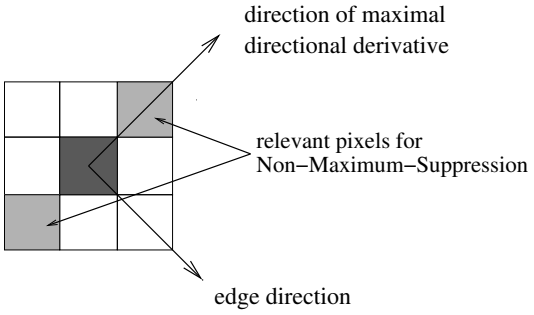


b) Apply the Canny-Filter with non-maximum suppression for different values for σ (width of the filter). For the following profiles, draw the positions of the detected edges as a function of σ.

3.2 Practical Exercise

Having seen how the Canny filter works in the 1-D case in the above theoretical exercises, we now move on to its practical application in the 2-D case. To complete this part, you may use either the Python script template `1_canny.py` or the Jupyter template `1_canny.ipynb`, which accompany this handout. **Before you start coding for this part, please read carefully the rest of the description below! It includes an exact definition of what your program is expected to do.**

In 2-D, the gradient magnitude is a measure for the strength of an edge, while direction of the gradient vector gives edge orientation. Edge detection can be done by thresholding the gradient magnitude, though this generally results in thick contours which are poorly localized. To thin the contours into single-pixel-wide lines, one should use non-maximum suppression. A pixel in the contour is kept as edge pixel only if its gradient magnitude is greater than that of its two neighboring pixels in the direction orthogonal to the edge, i.e. where the gradient attains a maximum.



Implement the Canny edge detector in 2-D. You should produce the following images:

- **Gaussian-smoothed image**: Check the hint in the code templates.
- **Gradient magnitude and direction image**: Apply the Sobel masks to the Gaussian-smoothed image to approximate partial intensity derivatives. Compute the magnitude and direction of the gradient based on these partial derivatives.
- **Orientation map**: At each pixel, the orientation of an edge can be obtained from the direction of the gradient. Note that directions α and $\alpha + \pi$ are equivalent with respect to induced edge orientation. The resulting directions can be coded as grey values.
- **Edge image according to the threshold criteria**: Each pixel at which the gradient magnitude exceeds a given threshold should be displayed. The value of this threshold should be adjusted according to the distribution of gradient magnitude values in the image.
- **Edge image after the non-maximum suppression**: Each pixel which fulfills the threshold criteria and has a larger gradient magnitude than its adjacent pixels in the direction orthogonal to the edge should be displayed. Note that this algorithm is different from the non-maximum suppression in Harris corner detection!

Extract edges from the provided images `zurlim.png`, `cube_left.pgm` and `cube_right.pgm` with your implementation. Appropriate parameters are suggested in the code templates.