Lab Session 1
Python introduction and Jupyter notebooks

Übungen zur Vorlesung "Methoden der Bioinformatik"
Teil Bildanalyse (Computer Vision), BMCV Group, PD Dr. K. Rohr
Wintersemester 2023/2024

Course web-page:
`http://www.bioquant.uni-heidelberg.de/research/groups/biomedical-computer-vision/teaching/compmeth/Labsessions`

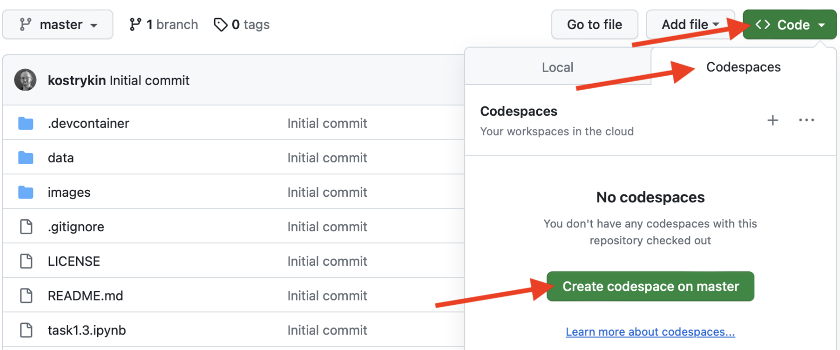
Overview of running GitHub Codespaces: `https://github.com/codespaces`

1 Setting up your GitHub repository

1. Open the course web-page in any web-browser of your choice.
2. Click on the link in "Create a new GitHub repository by using this link".
3. This will load another web-page entitled "Create a new repository". Leave everything on default and confirm the creation of the repository by clicking the "Create repository" button.
4. You should see a "Generating your repository" message for a few seconds and then be presented with an **overview of your repository**. You just created your first GitHub repository – congrats! :)

2 Firing up a GitHub Codespace

1. Open the **overview of your repository** on GitHub. This is the web-page that you landed on after completing Task 1.
2. Click on the green "Code'" button, then select the "Codespaces" tab.
3. Click the green "Create codespace on master" button. This will load **VS Code** (Visual Studio Code) inside of your web-browser. Wait until everything is loaded, it may take about one minute.



Note: If, at any time, **VS Code** behaves weirdly (e.g., complaining about missing extensions, not loading notebooks, or similar), try to simply reload the **VS Code** window. To do that, press Ctrl + ⇧ + P (or ⌘ + ⇧ + P if you are on macOS) and type "Reload window ↵". Your work progress will be preserved.

Lab Session 1

3 Your first Jupyter notebook

The left panel of **VS Code** shows an overview of **your local repository**. Right now it is identical to your GitHub repository (which is the **origin** of your local repository). The assignments of this course are organized into several Jupyter notebooks. These are the `task*.ipynb` files that you can see in your repository. By progressing from task to task, you will have to work with different notebooks.

1. Double click the file `task1.3.ipynb` inside of **VS Code** to open the notebook for this task, and follow the instructions inside the notebook. – **Note:** In Jupyter notebooks, code cells can be run in an *arbitrary order*. This is very helpful for experimenting and trying out new things. Nevertheless, an assignment in this course is *only* considered "finished" when the results can be reproduced by re-running all code cells *from top to bottom* by clicking the "Run all" button.

2. When finished, close your notebook. Changes are saved automatically within **your local repository**, but remember, that those changes will be lost when you close the codespace, unless you push them to your GitHub repository.

3. To do that, click on the "Terminal" tab at the bottom of **VS Code** and type the following Git commands. – **Hint:** The Git commands are given one per line, press ↵ at the end of each command:

   ```
   git commit --all
   git push origin master
   ```

   Pressing ↵ after the first command will open a text editor for typing a **commit message**. Type something like "Finish task 1.3" and close the editor.

   The **commit message** is arbitrary. If you wanted, you could revert to any previous commit at a later time, and choosing an expressive message is convenient for finding the commit which you will be looking for. Besides, there are some conventions for how a **commit message** should be formatted: It should tell in an "imperative mood" and as concisely as possible, what *the commit* should do[1].

4 Writing loops in Python

Now you already know how to edit, commit, and push a Jupyter notebook.

1. Open the notebook `task1.4.ipynb` and follow the instructions in the notebook.

2. Commit and push your changes, then close **VS Code**.

[1]From the official Git documentation: `https://git.kernel.org/pub/scm/git/git.git/tree/Documentation/SubmittingPatches?h=v2.36.1#n181`

Lab Session 2
Images, Histograms, Intensity clipping

Übungen zur Vorlesung “Methoden der Bioinformatik”
Teil Bildanalyse (Computer Vision), BMCV Group, PD Dr. K. Rohr
Wintersemester 2023/2024

1 Image IO (input/output)

1. Open VS Code in a GitHub Codespace (see Task 2 of Lab Session 1) using the repository which you created before (see Task 1 of Lab Session 1).

2. Open the notebook `task2.ipynb`.

3. Enter the following code into the *first* code cell of the notebook and run it:

```
import numpy
import matplotlib.pyplot as plt
```

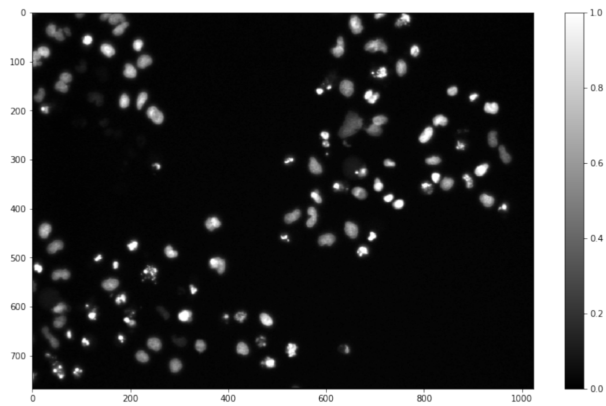
   Notes:

   - The first instruction should be clear from the lecture. The second instruction loads the module “`matplotlib.pyplot`” and makes it available by using the abbreviation “`plt`”. This module is useful for visualizing data.
   - Make sure this code cell *always* remains the very first code cell of your notebook, so it is run first when the Notebook is re-run from top to bottom.

4. Then, extend the notebook as follows:

   (a) Use `img = plt.imread('data/cells.png')` to load an image. **Note:** The type of the returned object (`img`) is `numpy.ndarray` (also see the introduction on page 2). Objects of this type represent images.

   (b) Use `plt.figure()` (or, e.g., `plt.figure(figsize=(15,8))` if you want to specify the size of the figure) to add a *figure* to a code cell of the notebook. Then, *within the same code cell*, use `plt.imshow(img, 'gray')`[1] to display the image within the figure. In addition, use `plt.colorbar()` after the `imshow`-instruction to include a legend of the gray-scale encoding. Run the code cell *with and without* the `colorbar`-instruction and observe the differences. Finally, you should obtain an output like this:



[1]The parameter “`'gray'`x’ in “`plt.imshow(img, 'gray')`” specifies the color map for a *monochromatic* image (i.e. a two-dimensional array of image *intensities*, non-RGB) for visualization (e.g., gray-scale)

Lab Session 2

Introduction: Working with `numpy.ndarray` objects

If `img` is an object of the type `numpy.ndarray`, then the object `img` has the following attributes and methods:

Attributes (data)	**Methods** (behaviours)
`img.ndim`: Corresponds to the dimension of `img`. **`img.shape`:** If `img` is an image, then the element `img.shape[0]` corresponds to the image height (number of rows) and `img.shape[1]` to the image width (number of columns).	**`img.copy()`:** Tells `img` to return a copy of itself. **`img.clip(t1, t2)`:** Tells `img` to return a copy of itself using intensity clipping (see Task 3). **`img.flatten()`:** Tells `img` to return a flat representation of itself (see Task 2). (Methods are like functions *which belong to objects*.)

Note that the above is *not* a complete list (there are many more attributes/methods[2] which are *not* relevant for this assignment).

Further hints regarding `numpy.ndarray` objects:

1. The pixel in the upper left corner of the image has the coordinates $(0, 0)$, and the pixel in the lower right corner has the coordinate $(\text{width} - 1, \text{height} - 1)$.
2. If `img` is an image (i.e. object of the type `numpy.ndarray`), then the intensity value of a pixel at position `p` corresponds to `img[p]`, where `p` has two elements (row, column). Alternatively, you can use `img[row, column]` instead of `img[p]`.

2 Histograms

Extend your notebook by a histogram of the previously loaded image – **Hints:**

1. The function `plt.hist(data)` produces a histogram of a *sequence of values* (`data`).
2. In Python, a *sequence* is, for example, a list, a *flat* array, . . .
3. The image `img` is a *two-dimensional* array and `img.flatten()` yields a flat representation (sequence of all pixel values).

[2] see `https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html`

Lab Session 2

3 Intensity clipping

Given two thresholds T_1 and T_2, each pixel x, y of the image with an intensity value $g(x, y)$ less than the threshold T_1 is assigned the value T_1 and each pixel with a value greater than T_2 is assigned the value T_2. Pixels with a value between the two thresholds T_1 and T_2 remain unchanged:

$$g_{\text{clip}}(x, y) = \begin{cases} T_1 & \text{if } g(x, y) < T_1, \\ T_2 & \text{if } g(x, y) > T_2, \\ g(x, y) & \text{otherwise} \end{cases}$$

If g is the image `img`, y is the image `row`, and x is the image `column`, then the mathematical expression $g(x, y)$ corresponds to `img[row, column]` in Python.

In this task, you will **perform intensity clipping** (i.e. compute g_{clip}, in *three* different ways) using the previously loaded image (Task 1) and visualize the result using the `imshow` function.

1. Perform intensity clipping using the method `clip` of `numpy.ndarray`.
2. Reproduce the behaviour of the method `clip`, i.e. perform intensity clipping *without* using the method `clip`!
   (a) Use *two* nested `for`-loops (one outer loop for the image rows or columns, one inner loop for the pixels of the current row/column) and `if`-conditions.
   (b) Use a *single* `for`-loop and `if`-conditions – **Hint:** `numpy.ndindex(img.shape)` yields an iterable. The items of this iterable correspond to *all* pixel coordinates of the image `img`. Each item is a pair of coordinates (i.e. row and column). Use "`[0]`" and "`[1]`" to access the corresponding row and column of an item.

Include a legend of the gray-scale encoding (using "`plt.colorbar()`") in *each* figure!

4 Writing re-usable code (BONUS)

In the previous task, you have used loops to reproduce the behaviour of the method `clip` of `numpy.ndarray`. Now, make this code *re-usable* by putting it into a *function* which you can use anytime later. For your convenience, a skeleton of the code you need to write to implement the function is already added to the notebook (see "`def clip_image...`").

Lab Session 3
Mean, median, and Gaussian filtering

Preparation. Open the notebook `task3.ipynb` in VS Code (see Task 2 of Lab Session 1). Enter the "`import numpy`" and "`import matplotlib.pyplot as plt`" instructions into the *first* code cell of the notebook and run it (cf. Lab Session 2).

1 Linear filtering by convolution (mean filter)

In this task you will implement and test a *mean filter* (box filter).

1. Use `imread` and `imshow` to load and show the image `data/lena.png`.
2. Finish the implementation of the re-usable function `meanfilter`. The function parameters are `img` (the image to be filtered) and `size` (filter size determining the filtering neighborhood). You may assume that `size` is an odd number. The function must return the filtered result image. Do *not* modify the input image `img`!
3. Test your solution by using the function `meanfilter` for the previously loaded image (e.g., set the filter size to `3`). If something does not work as expected, look for errors you have made and fix them. Repeat fixing and testing until everything works.
4. Compare your result for filter size `5` with the correct result image `data/lena_meanfilter5.png`. Use `imread` to load the image. For a *quantitative* comparison of two images, use the instruction "`assert numpy.allclose(img1, img2, atol=1/255)`" where "`img1`" and "`img2`" are the two images – **Notes:**
   (a) The instruction "`assert condition`" interrupts the code execution if `condition` is `False`, rising your attention to an error. Otherwise, nothing happens.
   (b) In contrast to a comparison using the mathematical equality operator ("`img1 == img2`"), a comparison using `allclose` tolerates *numerical inaccuracies*. The parameter "`atol=1/255`" specifies the maximum tolerated difference per pixel. Since the PNG file quantifies image intensities as multiples of $1/255$, errors lower than $1/255$ cannot be distinguished from numerical inaccuracies.

Hints:

1. To create a new `numpy.ndarray` object representing an image with height `shape[0]` and width `shape[1]`, initially filled with zeros, you can use `numpy.zeros(shape)`.
2. You can use two nested `for`-loops: An *outer* `for`-loop to iterate over all pixels of the image and an *inner* `for`-loop to iterate over all pixels of the filtering neighborhood.
3. To iterate over all pixels of the image or the filtering neighborhood using a `for`-loop, `ndindex` can be used (cf. Lab Session 2, Task 3.2(b)).
4. Bear in mind the border problem, i.e. you should not access pixels where the neighborhood is partially outside the image.
5. Also consider the schematic illustration in Figure 1.

Lab Session 3

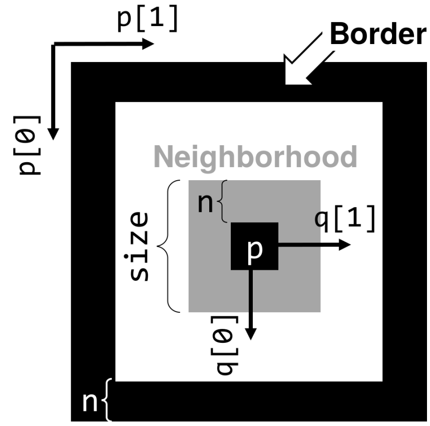


Figure 1: Schematic illustration of a `size` × `size` filtering *neighborhood* and the image *border* with `n` = ⌊(`size` − 1) /2⌋, assuming that `size` is an *odd* number ≥ 3. The image pixel `p` corresponds to the *center* of the filtering neighborhood (i.e. the *origin* of the coordinate system of the filtering neighborhood).

2 Non-linear filtering (median filter)

In this task you will implement and test a *median filter*.

1. Finish the implementation of the re-usable function `medianfilter`. The function parameters are `img` (the image to be filtered) and `size` (filter size). You may assume that `size` is an odd number. The function must return the filtered result image. Do *not* modify the input image `img`!

2. Analogously to Task 1, test your solution by using the function `medianfilter` and *quantitatively* compare your result for filter size `5` with the correct result image `data/lena_medianfilter5.png`.

Hints:

1. You can use a `list` to store the pixel values of the filtering neighborhood. An empty list is created by "`list()`" or the shorthand expression "`[]`".

2. The function `sorted(sequence)` returns a sorted `sequence` (e.g., a `list`, a *flat* `numpy.ndarray`). Alternatively, both `list` and `numpy.ndarray` objects provide the method `sort()`. Using this method tells the object to sort itself. – **Examples:**

Example 1	**Example 2**	**Output**

`data = [4, 3, 8, 2]`
`print(sorted(data))`
	`data = [4, 3, 8, 2]`
	`data.sort()`
	`print(data)`
		`[2, 3, 4, 8]`

If `data` is a `numpy.ndarray` object with `data.ndim == 2`, then invoking the method `data.sort()` sorts the values of each row of `data` independently from the other rows.

Lab Session 3

3 Using pre-implemented filters

1. Load the module `scipy.ndimage` using the instruction "`import scipy.ndimage`".
2. Use the following pre-implemented filters in this module and include the filtering results (via `imshow`) into your notebook:
   (a) Use `scipy.ndimage.uniform_filter(img, size)` for a mean filter.
   (b) Use `scipy.ndimage.median_filter(img, size)` for a median filter.
   (c) Use `scipy.ndimage.gaussian_filter(img, sigma)` for a Gaussian filter (where "`sigma`" is the standard deviation of the Gaussian function).
3. Compare the results obtained using the functions in 3.2(a) and 3.2(b) with those you obtained in Tasks 1 and 2. What are the main differences? Do you have an explanation? Use a *Markdown* cell to write your answer, i.e. change the cell type!

4 Slicing and benchmarking (BONUS)

In this task, you will implement a filtering function which is *faster* than those you implemented in Tasks 1 and 2. You will also learn how to *benchmark* the run time of code.

1. Decide which filtering method you want to accelerate (mean or median filter). If you are unsure, choose the one whose solution you are more confident with.
2. Finish the implementation of the re-usable function `fastfilter`. The function parameters are `img` (the image to be filtered) and `size` (filter size). The function must return the filtered result image. Do *not* modify the input image `img`!

   Important: Use the hints below to confine your code to only a single `for`-loop:

   (a) If `img` is an object of the type `numpy.ndarray` (e.g., an image), then `img[i0:i1, j0:j1]` corresponds to the rectangular subsection of the image (also called *slice*) ranging from row `i0` to `i1-1` and column `j0` to `j1-1` (all inclusive). The subsection itself also is of type `numpy.ndarray`.
   (b) **For mean filters:** The mean value of an `numpy.ndarray` object can be computed using its `mean()` method (e.g., `img.mean()` if `img` is an object of the type `numpy.ndarray`).
   (c) **For median filters:** The method `flatten()` of `numpy.ndarray` objects (cf. Lab Session 2) yields a flat representation (which can be sorted).
3. Test your solution by using the function `fastfilter` for the previously loaded image and *quantitatively* compare your result to that you obtain using `meanfilter` or `medianfilter`, respectively.
4. Use the instruction "`%timeit fastfilter(img, 9)`" to benchmark the run time of your `fastfilter` implementation. Use a similar instruction to benchmark the run time of `meanfilter` or `medianfilter`, respectively.
5. Document your observations and try to think of an explanation (use Markdown!)

Lab Session 4

Edge detection: Derivative operators

Übungen zur Vorlesung “Methoden der Bioinformatik”
Teil Bildanalyse (Computer Vision), BMCV Group, PD Dr. K. Rohr
Wintersemester 2023/2024

Preparation. Open the notebook `task4.ipynb` in VS Code (see Task 2 of Lab Session 1). Enter the “`import numpy`” and “`import matplotlib.pyplot as plt`” instructions into the *first* code cell of the notebook and run it (cf. Lab Session 2).

1 Prewitt filter

1. Use `imread` and `imshow` to load and show the image `data/lena.png`.

2. Compute the partial derivatives g_x and g_y of an image $g(x, y)$ by *convolving* the image with Prewitt derivative operators (see Figure 1). To this end, finish the implementation of the re-usable functions `prewitt_h` and `prewitt_v`. The functions are supposed to compute and return the *convolution* of the input image `img` with the horizontal and vertical Prewitt derivative operators, respectively. – **Hints:**

   (a) Do *not* modify the input image! Avoid the border problem by computing the partial derivatives g_x and g_y only for those pixels which have their neighborhood completely inside the image.

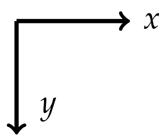
   (b) If you do not know where to start, start from your implementation of the mean filter (Lab Session 3, Task 1.2). The 3×3 mean filter performs a convolution using a uniform filter mask (weighting each image pixel equally, dividing by $3 \times 3 = 9$). You only need to change the weights based on the values of “`q`”!

3. Test your implementations for `prewitt_h` and `prewitt_v` by including images of the computed partial derivatives into your notebook. Use `colorbar()` after each `imshow`-instruction to also include a legend of the gray-scale encoding.

4. Quantitatively compare your results obtained using `prewitt_h` and `prewitt_v` with the correct result images `data/lena_prewitt_h.tiff` and `data/lena_prewitt_v.tiff`, respectively (for quantitative image comparison, recall Lab Session 3, Task 1).

Important hint: Use `skimage.io.imread` instead of `imread` to load a TIFF file. However, remember to first *load* the `skimage.io` module (use “`import skimage.io`”).

Rule of thumb:[1] There is no reason for not *always* using `skimage.io.imread` (and not using `imread`) except that you have to remember to load the module.



$$\frac{1}{6}\begin{array}{|c|c|c|}\hline +1 & 0 & -1 \\ \hline +1 & 0 & -1 \\ \hline +1 & 0 & -1 \\ \hline \end{array} \qquad \frac{1}{6}\begin{array}{|c|c|c|}\hline +1 & +1 & +1 \\ \hline 0 & 0 & 0 \\ \hline -1 & -1 & -1 \\ \hline \end{array}$$

Figure 1: Prewitt derivative operators

[1] e.g., for the exam ☺

Lab Session 4

2 Edge detection

1. Compute the *magnitude* of the image gradient

$$\left\|\nabla g\left(x,y\right)\right\| = \sqrt{g_x^2\left(x,y\right) + g_y^2\left(x,y\right)}$$

   and include the resulting image into the notebook. **Hints:**

   (a) When working with objects of the type `numpy.ndarray` (e.g., images), mathematical operations are *propagated* to the intensity values of the image. For example, if `img` is an image of the type `numpy.ndarray`, then the expression `img*2` yields an image with *doubled* intensities (in comparison to `img`).

   (b) The square root of an intensity value (or all values in an image that is a `numpy.ndarray` object) can be computed using the `numpy.sqrt` function (e.g., `numpy.sqrt(value)` or `numpy.sqrt(img)`).

2. Quantitatively compare your result with the correct result image `data/lena_prewitt_gradmag.tiff`.

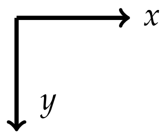
3 Sobel filter (BONUS)

Repeat Task 1 using the *Sobel* filter instead of the Prewitt filter:

1. Compute the partial derivatives g_x and g_y by *convolving* the image $g\left(x,y\right)$ with Sobel derivative operators (see Figure 2). To this end, implement the re-usable functions `sobel_h` and `sobel_v` analogously to `prewitt_h` and `prewitt_v`.

2. Test your implementations for `sobel_h` and `sobel_v` by including images of the computed partial derivatives into your notebook (also include color legends).

3. Quantitatively compare your results using `sobel_h` and `sobel_v` with the correct result images `data/lena_sobel_h.tiff` and `data/lena_sobel_v.tiff`, respectively.



$$\frac{1}{8}\begin{array}{|c|c|c|}\hline +1 & 0 & -1 \\ \hline +2 & 0 & -2 \\ \hline +1 & 0 & -1 \\ \hline\end{array} \qquad \frac{1}{8}\begin{array}{|c|c|c|}\hline +1 & +2 & +1 \\ \hline 0 & 0 & 0 \\ \hline -1 & -2 & -1 \\ \hline\end{array}$$

Figure 2: Sobel derivative operators