Image Analysis Fundamentals in Python

Aaron Ponti

Fall 2024

Contents

Figure 1: Python logo

1 Introduction

Python (https://www.python.org/) is an interpreted, high-level, dynamically-typed, garbage-collected, general purpose programming language designed and developed by Guido Van Rossum (the "benevolent dictator for life") at the Centrum Wiskunde & Informatica (Amsterdam) in the late 1980s and first released in 1991. Currently in its major version 3, it is used in many application domains, from web and internet development through desktop and mobile applications; in particular, and in line with our purposes, Python is becoming the language of choice of scientific and numerical applications (such as machine learning, data analysis and statistics, computer vision, and more). Python packages such a **NumPy** (http://www.numpy.org/), **SciPy**, (https://www.scipy.org/), **Pandas** (https://pandas.pydata.org/), **scikit-learn** (https://scikit-learn.org/), **scikit-image** (https://scikit-image.org/), **Matplotlib** (https://matplotlib.org/), **Tensorflow** (https://www.tensorflow.org/), and **PyTorch** (https://pytorch.org/) - just to mention a few - turn Python into a scientific computing powerhouse. Python runs on Windows, macOS, Linux, Raspberry Pi, iOS/iPadOS, and more.

There are several ways to download, install and maintain Python and its packages. The default way is to download Python itself from https://www.python.org/downloads/ and to use `pip` (the **P**ackage **I**nstaller for **P**ython) to manage the installed packages (https://pypi.org). An alternative, that has some advantages, makes use of the `conda` **package manager**.

1.1 Getting the course material

Before we learn how to set up and use Python and the scientific libraries that we will need in the course, we will download and extract the course material. The official course page is https://ia-res.ethz.ch/courses/iaf/. Please go ahead and download the full course material (as a zipped archive) from https://ia-res.ethz.ch/courses/iaf/iaf.zip; extract the archive and you will (almost) be ready to go. Paths to code and data used in the rest of this document will be relative to the `iaf` folder.

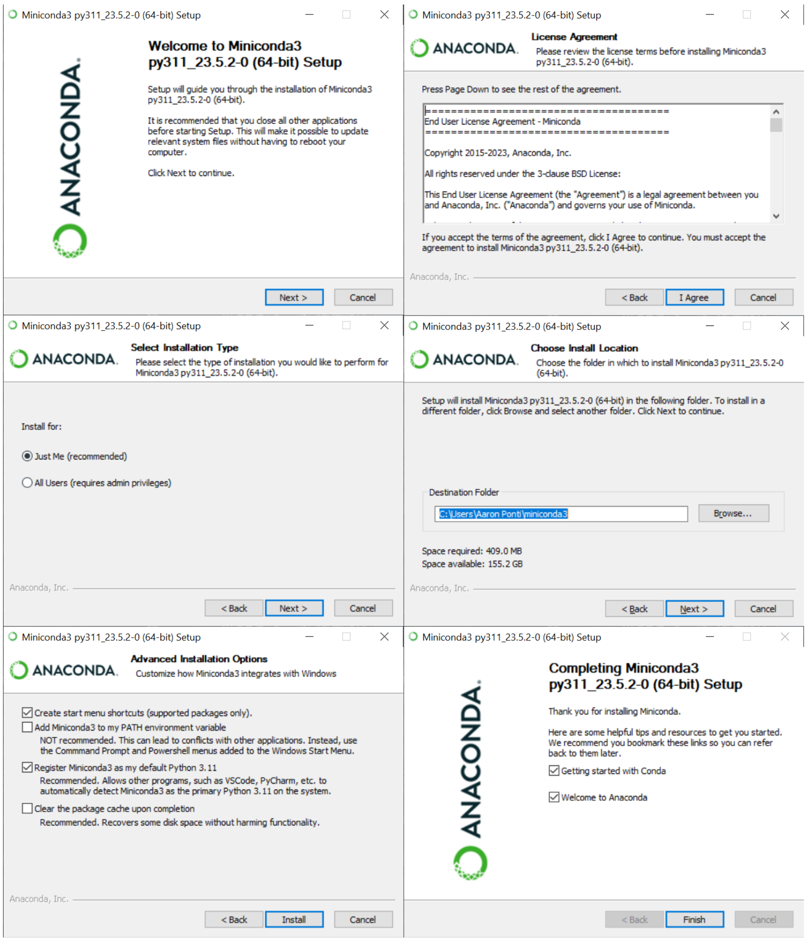
1.2 Downloading and installing Python (via Miniconda)

Anaconda (https://www.anaconda.com/) is a comprehensive data science platform that integrates Python and thousands of packages into a curated distribution focused on data sciences that can easily be maintained via the package manager `conda`. **Miniconda** (https://docs.conda.io/en/latest/miniconda.html), on the other hand, is a lightweight installer for `conda` that offers precise control over package installations, optionally confined in so-called *environments* (see Managing `conda` environments below), and facilitates seamless updates. While this is a functionality that it shares with `pip`, `conda` additionally enables effortless installation and management of parallel Python interpreters.

We can download Miniconda for **Python 3.11** for **Windows**, **Linux** or **macOS** 32 or 64bit from https://docs.conda.io/en/latest/miniconda.html. The following sections explain how to install Miniconda on Windows, macOS and Linux.

1.2.1 Installing Miniconda on Windows

Download **Miniconda3 Windows 64-bit** for **Python 3.11** to your Desktop. Double click the installer and follow the instructions. All default settings can be chosen.



Welcome to Miniconda3 py311_23.5.2-0 (64-bit) Setup

Setup will guide you through the installation of Miniconda3 py311_23.5.2-0 (64-bit).

It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.

Click Next to continue.

License Agreement

Please review the license terms before installing Miniconda3 py311_23.5.2-0 (64-bit).

Press Page Down to see the rest of the agreement.

```
========================================
End User License Agreement - Miniconda
========================================

Copyright 2015-2023, Anaconda, Inc.

All rights reserved under the 3-clause BSD License:

This End User License Agreement (the "Agreement") is a legal agreement between you
and Anaconda, Inc. ("Anaconda") and governs your use of Miniconda.
```

If you accept the terms of the agreement, click I Agree to continue. You must accept the agreement to install Miniconda3 py311_23.5.2-0 (64-bit).

Select Installation Type

Please select the type of installation you would like to perform for Miniconda3 py311_23.5.2-0 (64-bit).

Install for:

● Just Me (recommended)

○ All Users (requires admin privileges)

Choose Install Location

Choose the folder in which to install Miniconda3 py311_23.5.2-0 (64-bit).

Setup will install Miniconda3 py311_23.5.2-0 (64-bit) in the following folder. To install in a different folder, click Browse and select another folder. Click Next to continue.

Destination Folder: C:\Users\Aaron Ponti\miniconda3

Space required: 409.0 MB
Space available: 155.2 GB

Advanced Installation Options

Customize how Miniconda3 integrates with Windows

☑ Create start menu shortcuts (supported packages only).

☐ Add Miniconda3 to my PATH environment variable
NOT recommended. This can lead to conflicts with other applications. Instead, use the Commmand Prompt and Powershell menus added to the Windows Start Menu.

☑ Register Miniconda3 as my default Python 3.11
Recommended. Allows other programs, such as VSCode, PyCharm, etc. to automatically detect Miniconda3 as the primary Python 3.11 on the system.

☐ Clear the package cache upon completion
Recommended. Recovers some disk space without harming functionality.

Completing Miniconda3 py311_23.5.2-0 (64-bit) Setup

Thank you for installing Miniconda.

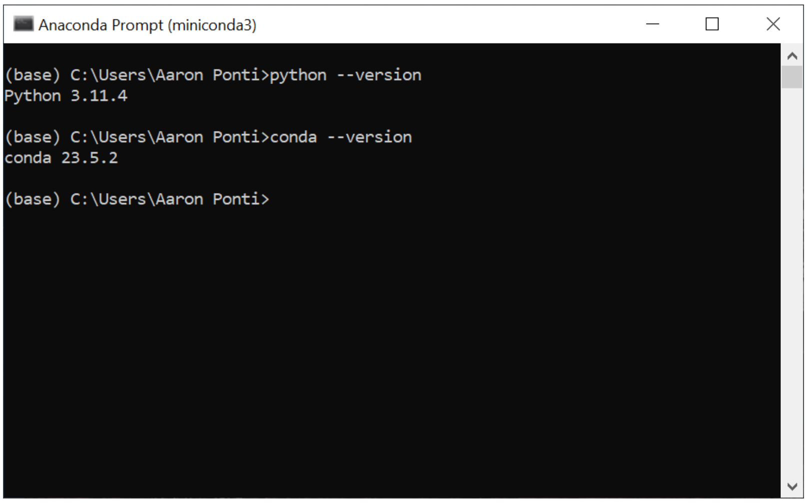
Here are some helpful tips and resources to get you started. We recommend you bookmark these links so you can refer back to them later.

☑ Getting started with Conda

☑ Welcome to Anaconda

Figure 2: Windows installation steps

From the Start Menu, choose **Anaconda Prompt (miniconda3)**. conda and Python are ready to be used.

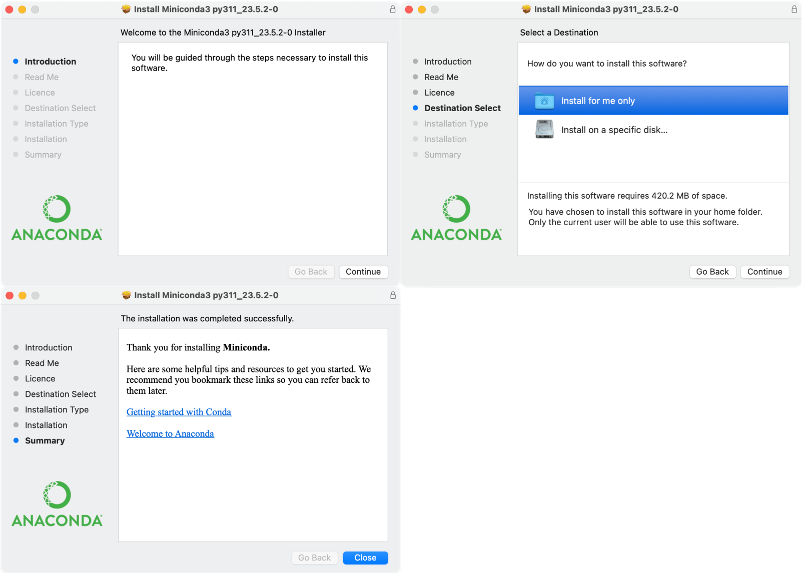


```
(base) C:\Users\Aaron Ponti>python --version
Python 3.11.4

(base) C:\Users\Aaron Ponti>conda --version
conda 23.5.2

(base) C:\Users\Aaron Ponti>
```

Figure 3: Python and conda on Windows

1.2.2 Installing Miniconda on macOS

Download **Miniconda3 macOS {Intel | M1} 64-bit pkg** (depending on your architecture) to your Desktop. Double click the installer and follow the instructions. All default settings can be chosen.



Welcome to the Miniconda3 py311_23.5.2-0 Installer

You will be guided through the steps necessary to install this software.

Select a Destination

How do you want to install this software?

Installing this software requires 420.2 MB of space.

You have chosen to install this software in your home folder. Only the current user will be able to use this software.

The installation was completed successfully.

Thank you for installing **Miniconda.**

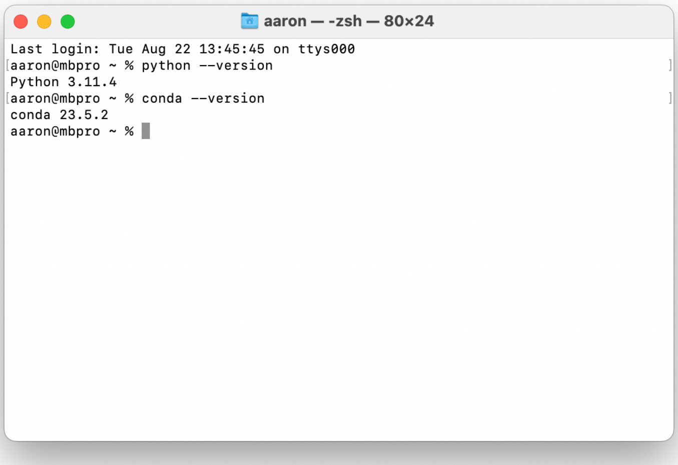
Here are some helpful tips and resources to get you started. We recommend you bookmark these links so you can refer back to them later.

Getting started with Conda

Welcome to Anaconda

Figure 4: macOS installation steps (a few screens are not shown)

From the **Applications**, start your terminal. conda and Python are ready to be used.



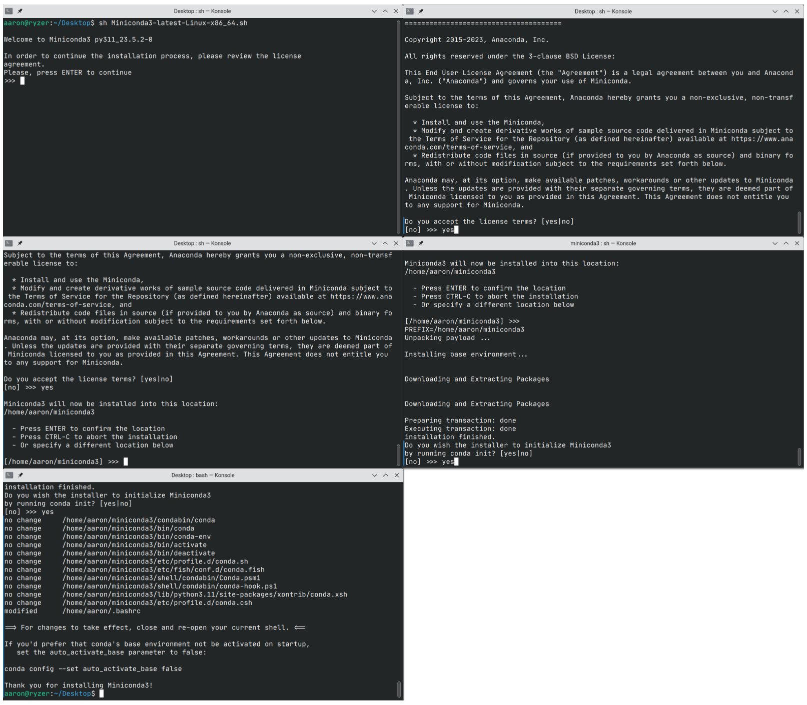
```
Last login: Tue Aug 22 13:45:45 on ttys000
aaron@mbpro ~ % python --version
Python 3.11.4
aaron@mbpro ~ % conda --version
conda 23.5.2
aaron@mbpro ~ %
```

Figure 5: Python and conda on macOS

1.2.3 Installing Miniconda on Linux

Download **Miniconda3 Linux 64-bit** to your Desktop. Open your terminal, execute the following (do not type the shell prompt $) and follow the instructions. It is recommended to answer yes to the question `Do you wish the installer to initialize Miniconda3 by running conda init?`

```
$ cd ~/Desktop
$ sh Miniconda3-latest-Linux-x86_64.sh
```



```
aaron@ryzer:~/Desktop$ sh Miniconda3-latest-Linux-x86_64.sh

Welcome to Miniconda3 py311_23.5.2-0

In order to continue the installation process, please review the license
agreement.
Please, press ENTER to continue
>>> 
```

```
======================================

Copyright 2015-2023, Anaconda, Inc.

All rights reserved under the 3-clause BSD License:

This End User License Agreement (the "Agreement") is a legal agreement between you and Anacond
a, Inc. ("Anaconda") and governs your use of Miniconda.

Subject to the terms of this Agreement, Anaconda hereby grants you a non-exclusive, non-transf
erable license to:

  * Install and use the Miniconda,
  * Modify and create derivative works of sample source code delivered in Miniconda subject to
 the Terms of Service for the Repository (as defined hereinafter) available at https://www.ana
conda.com/terms-of-service, and
  * Redistribute code files in source (if provided to you by Anaconda as source) and binary fo
rms, with or without modification subject to the requirements set forth below.

Anaconda may, at its option, make available patches, workarounds or other updates to Miniconda
. Unless the updates are provided with their separate governing terms, they are deemed part of
 Miniconda licensed to you as provided in this Agreement. This Agreement does not entitle you
to any support for Miniconda.

Do you accept the license terms? [yes|no]
[no] >>> yes
```

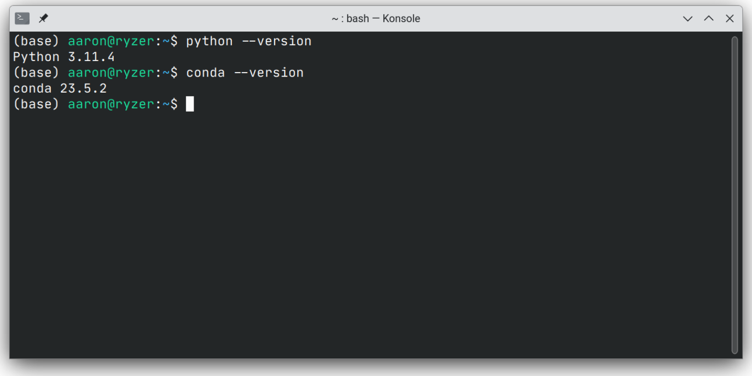
```
Subject to the terms of this Agreement, Anaconda hereby grants you a non-exclusive, non-transf
erable license to:

  * Install and use the Miniconda,
  * Modify and create derivative works of sample source code delivered in Miniconda subject to
 the Terms of Service for the Repository (as defined hereinafter) available at https://www.ana
conda.com/terms-of-service, and
  * Redistribute code files in source (if provided to you by Anaconda as source) and binary fo
rms, with or without modification subject to the requirements set forth below.

Anaconda may, at its option, make available patches, workarounds or other updates to Miniconda
. Unless the updates are provided with their separate governing terms, they are deemed part of
 Miniconda licensed to you as provided in this Agreement. This Agreement does not entitle you
to any support for Miniconda.

Do you accept the license terms? [yes|no]
[no] >>> yes

Miniconda3 will now be installed into this location:
/home/aaron/miniconda3

  - Press ENTER to confirm the location
  - Press CTRL-C to abort the installation
  - Or specify a different location below

[/home/aaron/miniconda3] >>> 
```

```

Miniconda3 will now be installed into this location:
/home/aaron/miniconda3

  - Press ENTER to confirm the location
  - Press CTRL-C to abort the installation
  - Or specify a different location below

[/home/aaron/miniconda3] >>>
PREFIX=/home/aaron/miniconda3
Unpacking payload ...

Installing base environment...


Downloading and Extracting Packages


Downloading and Extracting Packages

Preparing transaction: done
Executing transaction: done
installation finished.
Do you wish the installer to initialize Miniconda3
by running conda init? [yes|no]
[no] >>> yes
```

```
installation finished.
Do you wish the installer to initialize Miniconda3
by running conda init? [yes|no]
[no] >>> yes
no change     /home/aaron/miniconda3/condabin/conda
no change     /home/aaron/miniconda3/bin/conda
no change     /home/aaron/miniconda3/bin/conda-env
no change     /home/aaron/miniconda3/bin/activate
no change     /home/aaron/miniconda3/bin/deactivate
no change     /home/aaron/miniconda3/etc/profile.d/conda.sh
no change     /home/aaron/miniconda3/etc/fish/conf.d/conda.fish
no change     /home/aaron/miniconda3/shell/condabin/Conda.psm1
no change     /home/aaron/miniconda3/shell/condabin/conda-hook.ps1
no change     /home/aaron/miniconda3/lib/python3.11/site-packages/xontrib/conda.xsh
no change     /home/aaron/miniconda3/etc/profile.d/conda.csh
modified      /home/aaron/.bashrc

==> For changes to take effect, close and re-open your current shell. <==

If you'd prefer that conda's base environment not be activated on startup,
   set the auto_activate_base parameter to false:

conda config --set auto_activate_base false

Thank you for installing Miniconda3!
aaron@ryzer:~/Desktop$ 
```

Figure 6: Linux installation steps

After restarting the terminal, conda and Python are ready to be used.



```
(base) aaron@ryzer:~$ python --version
Python 3.11.4
(base) aaron@ryzer:~$ conda --version
conda 23.5.2
(base) aaron@ryzer:~$ 
```

Figure 7: Python and conda on Linux

1.3 Managing conda environments

Python is referred to as being *"with batteries included"*, because it comes with a very extensive **standard library** that implements a large number of functionalities. However, Python applications often use additional, third-party packages and modules that are not part of the standard library. Different applications may rely on different versions of these packages, and even different versions of Python itself, and a mechanism that allows any number of Python installations and package versions to coexist is needed. A **virtual environment**[1] is a separate space where specific versions of both Python and various packages can be installed, and be independent both from the base installation (what Miniconda created on installation) and other virtual environments.

Extensive documentation on how to manage conda environments can be found at https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html. Here we will discuss the fundamentals.

[1] There are several similar but distinct types of Python virtual environments. In our course, we will only discuss conda environments.

A conda environment is created with conda create, activated with conda activate and populated with packages with conda install as shown below.

To create[2] and activate an environment test_env using Python 3.11 use the following:

```
$ conda create -n test_env python=3.11 -y
$ conda activate test_env
```

To search for a specific package (*e.g.*, numpy), use conda search:

```
$ conda search numpy
...
numpy                          1.25.2 py310heeff2f4_0  pkgs/main
numpy                          1.25.2 py311h08b1b3b_0  pkgs/main
numpy                          1.25.2 py311h24aa872_0  pkgs/main
numpy                          1.25.2  py39h5f9d8c6_0  pkgs/main
numpy                          1.25.2  py39heeff2f4_0  pkgs/main
```

To install the latest version of a package, just specify the package name. Installing a specific version or a range of versions is also possible:

```
$ conda install numpy               # Latest version
$ conda install numpy==1.25.2       # Specific version ("==")
$ conda install numpy=1.25          # Latest patch of 1.25 (notice the "=")
$ conda install numpy">=1.0,<1.20" # Any version in the range 1.0 - 1.20 (excluded)
```

Specifying point versions or range of versions can become useful when many packages are installed in parallel that rely on different versions of the same dependencies. conda can try to *resolve* the environment by finding the version of each package that is compatible to all other packages requiring it, but for large environments this becomes difficult and error prone. Explicitly specifying the required versions can speed up installation and guarantee compatibility.

Please notice that the official conda repositories only host a subset[3] of all Python packages that can be found in the official **Python Package Index** on https://pypi.org/. Fortunately, pip installations work (mostly) fine in conda environments, and packages not found in the conda repositories can be added with a pip install call, for instance:

```
$ pip install pIceImarisConnector
```

Analogously to conda, pip also allows to install specific versions or ranges of versions for a package (albeit with a slightly different syntax):

```
$ pip install numpy               # Latest version
$ pip install numpy==1.25.2       # Specific version
$ pip install numpy==1.25.*       # Latest patch of 1.20
$ pip install numpy">=1.0,<1.20" # Any version in the range 1.0 - 1.20 (excluded)
```

In some rare occasions, packages installed via pip may show incompatibilities with packages installed with conda (NumPy can sometimes be an issue). In the extreme case, one can use conda to create the virtual environment with the desired Python version, and then use pip exclusively to populate it. **This is what we will do for our course.**

[2] You can also direct creation of the environment to a custom location, *e.g.* conda create --prefix D:\envs\my_env python. To activate it, pass the full path: conda activate D:\envs\my_env.

[3] A larger, community-driven alternative is https://conda-forge.org/.

Feature	conda	pip
Package Management	Manages Python and non-Python packages.	Manages Python packages only.
Environment Management	Can create isolated environments, install packages and any Python interpreter.	Can create isolated environments and install packages but cannot install a different Python interpreter.
Dependency Resolution	Offer robust version conflict resolution across packages.	Offers less robust and flexible dependency resolution.
Platform Compatibility	Provides platform-agnostic (binary) package installations.	Provides binary (*wheels*) and source packages that may require a C/C++ compiler on the client machine.
Package Sources	Sources packages from the conda package repository as well as other channels (*e. g.*, *conda-forge*).	Primarily sources packages from the Python Package Index (PyPI), which however hosts many more packages.
Scope	Specializes in data science applications.	Is better for generic Python development.

Table 1: ‘conda‘ vs. ‘pip‘ comparison.

1.4 Setting up the iaf-env conda environment

In this course, we will make use of quite a few libraries. To make their installations easier, we will install a single library iaf (for **I**mage **A**nalysis **F**undamentals) from a custom pypi repository that pulls all the other packages as dependencies. All needed **source** distributions from https://pypi.org/ have already been compiled into binary packages for Windows, Linux and macOS (both on Intel and M1 processors) and no local compilation *should* be needed[4]. In case the installation fails on your machine, please consult **Appendix A**.

The following three calls should set up everything we need for the course.

```
$ conda create -n iaf-env python=3.11
$ conda activate iaf-env
$ pip install --extra-index-url https://ia-res.ethz.ch/pypi iaf
```

Alternatively, you can use the code/python/iaf_environment.yml file:

```
$ conda env create -f iaf_environment.yml
```

You can test that the installation succeeded by running the following code:

```
$ conda activate iaf-env
$ python -c "import iaf; print(iaf.__version__)"
```

```
0.5.0
```

Everything needed for our course is installed and ready to use!

1.5 Editors / Environments

> **Note** Everything we will do from here on assumes that you have started the **Anaconda Prompt** (in Windows) or the **terminal** (macOS and Linux) and have activated the iaf-env environment using conda activate iaf-env.

How do we write and execute Python commands and programs?

[4] For packages that contain exclusively Python code, installation from a **source distribution** (fundamentally, an archive) just copies the extracted code to the location that the Python interpreter scans for packages and modules. Some third-party packages, however, rely on *extensions* (*i.e.*, dynamic libraries) written in C, C++, or Cython (https://cython.org), that often but not always come packaged in compiled form as a **binary distribution** (*wheel*). In the case these extensions are not pre-compiled, a development environment must be installed and configured on the target machine to build them from source code during package installation.

1.5.1 Python itself

The simplest (and probably least recommended) way to run Python code is to start the Python interpreter and type commands. In the console, type `python`:

```
$ python
Python 3.11.4 (main, Jul  5 2023, 13:45:01) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The interpreter is ready to take commands. You can quit by typing `quit()`.

1.5.2 IPython

The next best option is `IPython` (https://ipython.org/), which is a more flexible and more interactive version of the Python interpreter. In the console, type `ipython`:

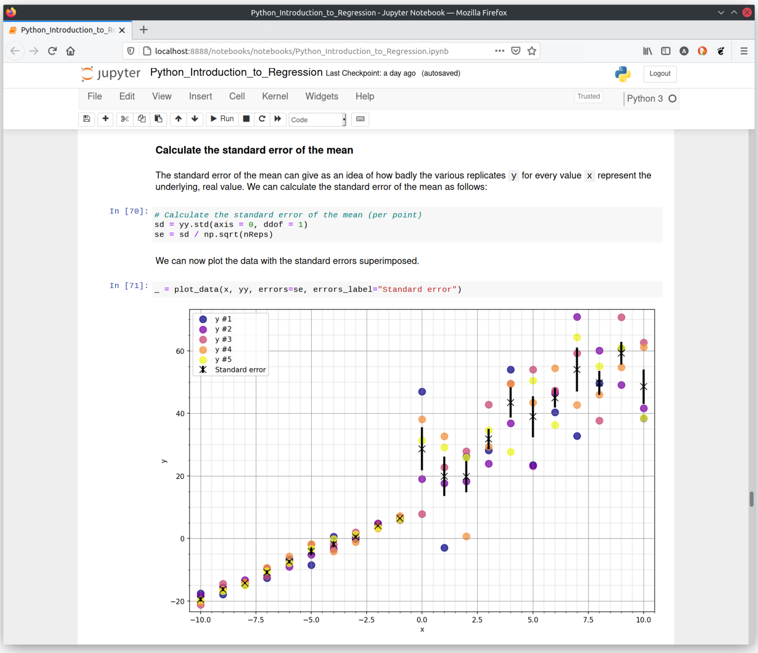
```
$ ipython
Python 3.11.4 (main, Jul  5 2023, 13:45:01) [GCC 11.2.0]
Type 'copyright', 'credits' or 'license' for more information
IPython 8.14.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

The IPython interpreter is ready to take commands. You can quit by typing `exit`.

1.5.3 Jupyter Notebook and JupyterLab

In our course we will predominantly use IPython and Jupyter Notebooks (https://jupyter.org/). A Jupyter notebook is a Python interpreter than can be run from inside a web browser, and allows for mixing code, documentation, results, graphics, all in the same interactive document!

You can run jupyter by changing to the folder where you want to store your notebooks, and then executing `jupyter notebook` or `jupyter lab` (JupyterLab is the next generation of the Jupyter environment). We will see how to use Jupyter more in detail below.


Python_Introduction_to_Regression - Jupyter Notebook — Mozilla Firefox
Python_Introduction_to_Re
localhost:8888/notebooks/notebooks/Python_Introduction_to_Regression.ipynb
jupyter
Python_Introduction_to_Regression Last Checkpoint: a day ago (autosaved)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Trusted
Python 3
Run
Code
y #1
y #2
y #3
y #4
y #5
Standard error
60
40
20
0
−20
y
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x

Calculate the standard error of the mean

The standard error of the mean can give as an idea of how badly the various replicates `y` for every value `x` represent the underlying, real value. We can calculate the standard error of the mean as follows:

```
In [70]: # Calculate the standard error of the mean (per point)
         sd = yy.std(axis = 0, ddof = 1)
         se = sd / np.sqrt(nReps)
```

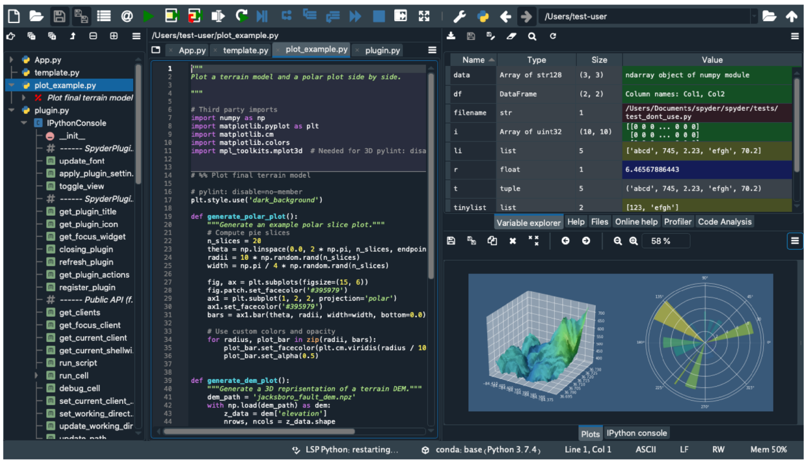
We can now plot the data with the standard errors superimposed.

```
In [71]: _ = plot_data(x, yy, errors=se, errors_label="Standard error")
```

Figure 8: Jupyter Notebook

1.5.4 Spyder IDE

Finally, a powerful **integrated development environment** (IDE) very reminescent of MATLAB is **spyder** (https://www.spyder-ide.org/). We won't use spyder in this course, but feel free to install it with `conda install spyder` and run it with `spyder`.



```
"""
Plot a terrain model and a polar plot side by side.

"""

# Third party imports
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm
import matplotlib.colors
import mpl_toolkits.mplot3d  # Needed for 3D pylint: disa


# %% Plot final terrain model

# pylint: disable=no-member
plt.style.use('dark_background')

def generate_polar_plot():
    """Generate an example polar slice plot."""
    # Compute pie slices
    n_slices = 20
    theta = np.linspace(0.0, 2 * np.pi, n_slices, endpoin
    radii = 10 * np.random.rand(n_slices)
    width = np.pi / 4 * np.random.rand(n_slices)

    fig, ax = plt.subplots(figsize=(15, 6))
    fig.patch.set_facecolor('#395979')
    ax1 = plt.subplot(1, 2, 2, projection='polar')
    ax1.set_facecolor('#395979')
    bars = ax1.bar(theta, radii, width=width, bottom=0.0)

    # Use custom colors and opacity
    for radius, plot_bar in zip(radii, bars):
        plot_bar.set_facecolor(plt.cm.viridis(radius / 10
        plot_bar.set_alpha(0.5)


def generate_dem_plot():
    """Generate a 3D reprisentation of a terrain DEM."""
    dem_path = 'jacksboro_fault_dem.npz'
    with np.load(dem_path) as dem:
        z_data = dem['elevation']
        nrows, ncols = z_data.shape
```

Name	Type	Size	Value
data	Array of str128	(3, 3)	ndarray object of numpy module
df	DataFrame	(2, 2)	Column names: Col1, Col2
filename	str	1	/Users/Documents/spyder/spyder/tests/test_dont_use.py
i	Array of uint32	(10, 10)	[[0 0 0 ... 0 0 0] [0 0 0 ... 0 0 0]
li	list	5	['abcd', 745, 2.23, 'efgh', 70.2]
r	float	1	6.46567886443
t	tuple	5	('abcd', 745, 2.23, 'efgh', 70.2)
tinylist	list	2	[123, 'efgh']

Figure 9: Spyder IDE

1.5.5 Others

Good Python **IDEs** are **JetBrains PyCharm** (https://www.jetbrains.com/pycharm/: the community edition is for free), and **Microsoft Visual Studio Code** (https://code.visualstudio.com/download) with the **Python extension** (https://marketplace.visualstudio.com/items?itemName=ms-python.python). Both integrate scientific tools (such as Jupyter notebooks) with broader Python development tools. We won't use these tools in the course.

2 Python language basics

To follow along with the basics of the Python programming language, start either `python` or `ipython` from the command line after having activated the `iaf-env` environment.

```
$ conda activate iaf-env
$ ipython    # or $ python; or $ jupyter notebook; or $ jupyter lab
```

For this introductory section course, we will use `ipython`.

2.1 Getting help

The **official Python documentation** (https://docs.python.org/3/) is very extensive, and covers all aspects of Python installation, usage, estension, and distribution. The best entry point for new users is the **Tutorial** (https://docs.python.org/3/tutorial/index.html), that slowly and informally introduces the new *pythonista* to the basic concepts and features of the Python language and system. Once the basics are digested, the complete **reference to the Python programming language** (https://docs.python.org/3/reference/index.html) and its **standard library** (https://docs.python.org/3/library/index.html#library-index) document every single aspect of Python in all its glorious details.

The following is an even simpler version of the tutorial to get us started with Python.

2.2 Hello World!

The first program in any programming language is always the **Hello, World!** program (https://en.wikipedia.org/wiki/%22Hello,_World!%22_program). It is a simple program that outputs the text `Hello, World!`

on the screen[5]:

```
In [1]: print("Hello, World!")
Hello, World!
```

`print()` is a Python **function**, and `"Hello, World!"` is a **string**. We will discuss them in detail below, but let's first start with some more fundamental concepts.

In IPython, prompts of the form `In [1]:` indicate the expressions and commands that we type at the Python prompt, whereas `Out[1]:` shows the corresponding output:

```
In [1]: car = { "brand": "Ford", "model": "Mustang", "year": 1964 }
In [2]: car    # Display car
```

```
Out[2]: {'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

For more clarity, we will render input code with syntax coloring and outputs in black.

2.3 Python as a calculator

The Python interpreter directly executes **expressions** typed at the prompt, and can therefore be used as a simple calculator.

```
In [1]: 2+2
```

```
Out[1]: 4
```

The basic arithmetic operators are `+`, `-`, `*`, `/`, `**`, `%` and these are used in conjunction with brackets: `()`. The symbol `**` is used to get exponents (powers): `2**4=16`. The symbol `%` is the modulo operator. **Operator precedence** in Python follows standard arithmetic order and is defined as follows:

1. quantities in brackets
2. powers ($**$)
3. $*$, / working left to right
4. $+$, $-$ working left to right

For example,

```
In [2]: (2+3)*4/5**2
```

is performed in this sequence:

1. (2+3) $=$ 5 $\rightarrow$ **5** * 4 / 5**2
2. 5**2 $=$ 25 $\rightarrow$ 5 * 4 / **25**
3. 5*4 $=$ 20 $\rightarrow$ **20** / 25

and the output is:

```
Out[2]: 0.8
```

2.4 Assignments (to variables)

When performing calculations at the Python prompt, the results are displayed on the console. But what if we want to use that result in a subsequent calculation? Manually retyping the output isn't efficient, especially for more complex code development. The solution is to *capture* the output for future use. This can be efficiently achieved by assigning the result to a **variable**:

```
In [1]: a = 3 - 2**4    # No output!
```

The result (-13) is *assigned* to the *variable* a. We can now use this variable for a subsequent operation:

```
In [2]: a * 5
```

```
Out[2]: -65
```

[5] You can also type `print("Hello, World!")` into your text editor of choice, save it as `hello_world.py`, and run it from your Anaconda Prompt or terminal with `python hello_world.py`. This is anyway how you would run larger applications. **Careful**: please make sure to use an editor that saves as **plain text** (Visual Studio Code, Notepad, Notepad++, vim, ...): applications like TextEdit, Wordpad, or Microsoft Word, that save in a Rich Text format, will create files that the Python interpreter cannot understand.

The value of `a * 5` is output to the console. If we want to store this result as well, we need to use another variable:

```
In [3]: b = a * 5    # Again, no ouput!
```

We can now look at what variables we have:

```
In [4]: a
```

```
Out[4]: -13
```

```
In [5]: b
```

```
Out[5]: -65
```

In IPython, we can use the **magic commands** `who` and `whos` (or `%who` or `%whos`) to see what variables we have defined so far.

```
In [6]: whos
```

```
Variable   Type    Data/Info
----------------------------
a          int     -13
b          int     -65
```

We will discuss the `Type` of variables next.

2.5 Variable (or data) types

We say that Python is a *dynamically-typed* language, because even though **it is** *typed*, we (mostly) do not need to explicitly state the type of a variable, as long as Python can infer it.

A (data) type determines:

- the possible **values** for that type (integers, floating points, strings, booleans, ...);
- the **operations** that can be done on values of that type (*e.g.*, arithmetic operations on numbers, concatenation for strings, ...);
- the way values of that type can be **stored** (32-bit vs. 64-bit floating points, ...).

Most programming languages also allow the programmer to define *additional* data types (or *classes*), usually by combining multiple elements of other types and defining the valid operations of the new data type. We will discuss them later in the course.

2.5.1 Numbers

In C++ (and C, Java, C#, and many others *statically-typed* languages), the following code would not compile:

```
a = 3;
```

Depending on the compiler, the error message would be something like this:

```
variables.cpp:3:2: error: 'a' was not declared in this scope
    3 |   a = 3;
      |   ^
```

Indeed, `a = 3` is an **assignment**, and in statically-typed languages a variable can be assigned a value only **after** it has been **declared**.

```
int a;     # Declare a variable of type int(eger)
a = 3;     # Assign 3 to it

int b = 5; # This combines the two steps
```

In Python, this is legitimate:

```
In [1]: a = 3
```

The Python interpreter infers that 3 is an integer number, and creates a variable `a` that is of type `int`. We can test this a follows:

```
In [2]: type(a)

Out[2]: int
```

Integers are not the only type of numbers known to Python. **Floating point numbers** are **numbers** that contain floating decimal points[6]. For example, 5.5, -2.0, $1e-3$:

```
In [3]: b = 1.3
In [4]: type(b)

Out[4]: float
```

The division operator / has an alternative version // that forces the result of the division to be an integer (for partial back-compatibility with Python 2):

```
In [5]: 16 / 3

Out[5]: 5.333333333333333

In [6]: 16 // 3

Out[6]: 5
```

2.5.2 Strings

Strings represents sequences of characters (*i.e.*, text). A string can be defined in using single quotes `'...'` or double quotes `"..."`, and can even be mixed:

```
In [1]: 'Do you like Python?'  # Single quotes

Out[1]: 'Do you like Python?'

In [2]: "Of course!"  # Double quotes

Out[1]: 'Of course!'

In [3]: "It's the best"  # Using double quotes, we can have single quotes in the string!

Out[3]: "It's the best"

In [4]: 'It\'s the best'  # We can also 'escape' the single quote

Out[4]: "It's the best"
```

Strings can be **concatenated** with the + operator:

```
In [5]: "Hello, " + "World!"

Out[5]: 'Hello, World!'
```

To concatenate strings and numbers, the number must be explicitly converted to string first:

```
In [6]: "I am " + str(27) + " years old!"

Out[6]: 'I am 27 years old!'
```

In recent versions of Python, **f-strings** were introduced to simplify and optimize this concatenations:

```
In [7]: age = 27
In [8]: f"I am {age} years old!"   # Notice the leading 'f'

Out[8]: 'I am 27 years old!'
```

String literals can span multiple lines and they are delimited by `"""..."""` or `'''...'''`. We will see examples of string literals when we discuss functions and their documentation.

[6] Python also knows other type of numbers, such as `Decimal` (https://docs.python.org/3/library/decimal.html#decimal.Decimal), `Fraction` (https://docs.python.org/3/library/fractions.html#fractions.Fraction, and `complex` (https://docs.python.org/3/library/stdtypes.html#typesnumeric) that we won't further discuss here.

2.5.3 Lists

Lists are one of Python's *compound* data types, that are used to group values together. A list is a sequence of comma-separated values (or items) between square brackets. These items may be of different type, though in practice this is rarely the case.

```
In [1]: squares = [1, 4, 9, 16, 25]
```

Lists (as all other **sequence** type, such as strings and tuples) can be **indexed** and **sliced**.

A list is **indexed** by passing an integer that represents the position of the value inside the list, with the first element of a list of `n` elements having `index = 0` and the last element having index `n - 1`. A list can also be indexed from the end, with the last index being `-1`.

```
In [2]: squares[0]
```

```
Out[2]: 1
```

```
In [3]: squares[4]
```

```
Out[3]: 25
```

```
In [4]: squares[-1]
```

```
Out[4]: 25
```

A list is sliced by defining a **start** and an **end index** and optionally a **step**. Be aware that the start index is included, but the end index is excluded!

```
In [5]: squares[1:3]   # Elements at indices 1 and 2
```

```
Out[5]: [4, 9]
```

```
In [6]: squares[1:5:2] # 1 to 5 (excluded) with a step of 2
```

```
Out[67]: [4, 16]
```

If indices are omitted, they will fall back to their default values: `start = 0`, `stop = n`, `step = 1`.

```
In [7]: squares[::2]  # From 0 to 4 (included) with step 2
```

```
Out[7]: [1, 9, 25]
```

Lists can be modifies in several ways. Individual and slices of numbers can be replaced:

```
In [8]: squares[1] = 10             # -> [1, 10, 9, 16, 25]
In [9]: squares[2:5] = [50, 60, 70] # -> [1, 10, 50, 60, 70]
```

New numbers can be added to the list. The `append(element)` method adds the new element at the end of the list; the `insert(position, element)` adds the new element at the specified position.

```
In [10]: squares.append(100)    # -> [1, 10, 50, 60, 70, 100]
In [11]: squares.insert(0, -1) # -> [-1, 1, 10, 50, 60, 70, 100]
In [12]: squares.insert(3, 25) # -> [-1, 1, 10, 25, 50, 60, 70, 100]
```

Lists can be concatenated with the + operator:

```
In [13]: a = [1, 2, 3]
In [14]: b = [4, 5, 6]
In [15]: a + b
```

```
Out[15]: [1, 2, 3, 4, 5, 6]
```

The built-in function `len()` returns the number of elements in the list.

```
In [16]: len(squares)
```

```
Out[16]: 8
```

2.5.4 Tuples

Tuples are similar to lists, but are delimited by round brackets `( ... )` instead of square brackets. A tuple is an ordered and **unchangeable** collection of items. Once a tuple is defined, it cannot be changed anymore.

```
In [1]: fruits = ("apple", "pear", "orange", "banana")
In [2]: len(fruits)
```

```
Out[2]: 4
```

```
In [3]: fruits[2]
```

```
Out[3]: 'orange'
```

```
In [4]: fruits[2] = "grapes"
```

```
TypeError: 'tuple' object does not support item assignment
```

One common usage of tuples is for **functions** (see below) to return more than one output.

2.5.5 Dictionaries

Dictionaries are used to store data values in `key:value` pairs. A dictionary is an ordered[7] collection of items (of any data type), that can be changed but that does not allow duplicates.

```
[1]: car = {
    ...:     "brand": "Ford",
    ...:     "model": "Mustang",
    ...:     "year": 1964
    ...: }
    ...:              # Hit the enter key one more time to execute!
In [2]: print(car)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

```
In [3]: car["year"] = 1967
In [4]: car["year"]
```

```
Out[4]: 1967
```

New keys can be added at any time:

```
In [5]: car["color"] = "red"
In [6]: car
```

```
Out[6]: {'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'red'}
```

`key:value` pairs can be removed from the dictionary with the `del` command.

```
In [7]: del car["brand"]
In [8]: car
```

```
Out[8]: {'model': 'Mustang', 'year': 1964, 'color': 'red'}
```

Keys and values can be queried with the `keys()` and `values()` methods, respectively. Later, we will see how to use these (and the `items()` method) to iterate over a dictionary:

```
In [9]: print(car.keys())
```

```
dict_keys(['model', 'year', 'color'])
```

```
In [10]: print(car.values())
```

```
dict_values(['Mustang', 1964, 'red'])
```

```
In [11]: print(car.items())
```

```
dict_items([('model', 'Mustang'), ('year', 1964), ('color', 'red')])
```

[7] Python dictionaries are ordered since Python 3.7. In earlier versions, the order was not preserved.

2.5.6 User types

We will discuss user types (**classes**) later in the course, after we will have introduced a few additional important concepts!

2.6 Code indentation

Before moving on to more complex topics that require *combinations of statements*, we need to discuss a fundamental distinction between Python and most other programming languages. In Python, code formatting is not merely aesthetic, but it dictates the code's structure itself! Consider this short C++ (sub)program:

```
for (int i=0; i<10; i++)
{
    if (i == 5)
    {
        break;
    }
}
```

Here, a **block** of code is encapsulated by curly braces `{ ... }`. This code is completely equivalent to:

```
for(int i=0; i<10; i++){if(i == 5) {break;}}
```

Contrast this with the equivalent Python code:

```
for i in range(10):
    if i == 5:
        break
```

In Python, **blocks** are determined by the level of **indentation**! Lines indented by an equal number of spaces are part of the same block! Any deviation in formatting could disrupt the code's functionality. While this makes Python's code very clean, we have to be especially careful: altering the formatting could yield unpredictable (wrong!) behavior.

The two code snippets below have only one difference: the indentation of the `delete_dir(dirname)` line. Which of the two snippets behave as intuitively expected (**A** or **B**)? What does the other one do?

Notice: for your safety (and the safety of your data), the `delete_dir()` function does not exist.

```
# A
r = input(f"Delete directory '{dirname}' ?")
if r == "y":
    print(f"Deleting directory '{dirname}'...")
    delete_dir(dirname)

# B
r = input(f"Delete directory '{dirname}' ?")
if r == "y":
    print(f"Deleting directory '{dirname}'...")
delete_dir(dirname)
```

2.7 Relational and logical operators

A **relational operator** is a language construct or operator that tests or defines some kind of relation between two entities. These include numerical equality (*e.g.*, `5 == 5`) and inequality (*e.g.*, `4 >= 3`): the result of such a test is either `True` or `False` [8].

An expression created using a relational operator forms what is known as a **relational expression** or a **condition**. In Python, relational operators are `==` (*equal to*), `!=` (*not equal to*), `<` (*less than*), `>` (*greater than*), `<=` (l*ess than or equal to*), `>=` (*greater than or equal to*).

```
In [1]: a = 5
In [2]: a > 3
```

[8] Please mind the case: **True** (not: **true**); **False** (not: **false**).

```
Out[2]: True
```

The result of a relational expression is a variable of type `bool` (**boolean**) and its value can be either `True` or `False` (and nothing else).

A **logical operator** combines relational expressions in a way that again results in either a `True` or `False` result. Here we will consider the `and`, `or` and `not` operators, that operate on *scalars*.

```
In [3]: a > 3 and a % 2 == 0
```

```
Out[3]: False
```

While `a` (*i.e.*, 5) is indeed larger than 3, it is not an even number and therefore the combined expression:

```
In [4]: a > 3 and a % 2 == 0
```

evaluates to `False`. The `and` and `or` operators are so-called **short-circuit operators**: they evaluate their second operand only when the result is not fully determined by the first operand. Short-circuit operators are handy in situations like the following:

```
In [5]: b = 0
In [6]: b != 0 and a / b > 0
```

```
Out[6]: False
```

If `b` is 0, `a / b` (*i.e.*, `a / 0`) is never calculated[9].

A logical expression

```
A and B and C and ... and Z
```

is true only if **all** statements `A`, `B`, `C`, …, `Z` are true[10].

A logical expression

```
A or B or C or ... or Z
```

is true if **at least one** of its operands is true[11].

The `not` operator, produces a value of `True` if its operand is `False` and a value of `False` if its operand is `True`[12].

2.8 Conditional flow control: if, else

Conditional statements enable selecting at run time which block of code to execute. The code path to run is chosen depending on the result of one or more logical operations. The simplest conditional statement is an `if` statement. For example:

```
In [1]: from random import randint # We will discuss modules later
In [2]: a = randint(1, 100) # Random integer between 1 and 100
In [3]: if a % 2 == 0:
    ...:     print("a is even!")
```

If `a` is even, this code will output:

```
a is even!
```

Unfortunately, if `a` is odd nothing will be output. The keyword `else` comes to rescue:

```
In [4]: if a % 2 == 0:
    ...:     print("a is even!")
    ...: else:
    ...:     print("a is odd!")
```

[9] Without the short-circuiting of the first check, `a/b` would be evaluated and would result in a `ZeroDivisionError: division by zero`.

[10] See the section on *logical conjunction* on http://en.wikipedia.org/wiki/Truth_table.

[11] See the section on *logical disjunction* on http://en.wikipedia.org/wiki/Truth_table.

[12] See the section on *logical negation* on http://en.wikipedia.org/wiki/Truth_table.

Now all cases are covered. If statements can include any number of alternate choices using the keyword elif (*else if*):

```
In [5]: if a < 30:
   ...:     print("small")
   ...: elif a < 80:
   ...:     print("medium")
   ...: else:
   ...:     print("large")
```

Notice that a = 20 would satisfy both the first and the second test, but Python exits the if block as soon as the first True condition is met.

2.9 Loop control: for, while, continue, break

2.9.1 for loop

A for loop is used for iterating over the items of any sequence (*e.g.*, a list, a tuple, a dictionary, a set, a string, or a range):

```
for i in sequence:
    statements
```

For example:

```
In [1]: fruits = ["apple", "banana", "cherry"]
In [2]: for f in fruits:
   ...:     print(f)

apple
banana
cherry
```

To iterate over a sequence of numbers, the range() function can be used:

```
In [3]: for i in range(0, 5, 2): # From 0 (included) to 5 (excluded) with step 2
   ...:     print(i)

0
2
4
```

for loops can be nested:

```
In [4]: for x in range(1, 4):       # 1, 2, 3
   ...:     for y in range(1, 3):   # 1, 2
   ...:         print(x + y)
```

The two nested loops above result in the following sequence of operations:

```
(x = 1) + (y = 1) = 2
(x = 1) + (y = 2) = 3
(x = 2) + (y = 1) = 3
(x = 2) + (y = 2) = 4
(x = 3) + (y = 1) = 4
(x = 3) + (y = 2) = 5
```

For each iteration of the external for loop, all iterations of the internal loop are executed.

Dictionaries can be iterated over in a series of ways:

```
In [5]: car = {
   ...:     "brand": "Ford",
   ...:     "model": "Mustang",
   ...:     "year": 1964
   ...: }
```

```
In [6]: for key in car:                 # Iterates over the keys
    ...:     print(key)
```

```
brand
model
year
```

```
In [7]: for key in car.keys():          # Again, iterates over the keys
    ...:     print(key)
```

```
brand
model
year
```

```
In [8]: for value in car.values():      # Iterates over the values
    ...:     print(value)
```

```
Ford
Mustang
1964
```

```
In [9]: for key, value in car.items(): # Iterates over the items (tuple!)
    ...:     print(key, value)
```

```
brand Ford
model Mustang
year 1964
```

2.9.2 while loop

The while loop repeats a group of statements an indefinite number of times under control of a logical condition. A while statement has following syntax:

```
while expression:
    statements
```

As long as expression is True, the statements will be executed.

Example If we sum all integer numbers starting from n = 1, what is the first value of n for which the sum of all integers 1 + 2 + 3 + ... + n exceeds 100?

```
In [1]: n = 0
    ...: sum = 0
    ...: while sum <= 100:    # As soon as this turns False, the block is
    ...:     n = n + 1        # no longer executed!
    ...:     sum = sum + n
    ...: print(n)
```

```
14
```

With n = 13, the sum is 91. With n = 14, the sum is 105, and is larger than 100 for the first time: the sum <= 100 expression returns False and the while loop is exited.

2.9.3 continue and break

The continue operator passes control to the next iteration of the for or while loop in which it appears, skipping any remaining statements in the body of the loop. The same holds true for continue statements in **nested** loops. That is, execution continues at the beginning of the loop in which the continue statement was encountered.

```
In [1]: for x in range(6):
    ...:     if x == 3:
    ...:         continue
    ...:     print(x)
```

```
0
1
2
4
5
```

The `break` statement lets you exit early from a `for` loop or `while` loop. In nested loops, break exits from the innermost loop only.

```
In [2]: for x in range(6):
   ...:     if x == 3:
   ...:         break
   ...:     print(x)

0
1
2
```

Exercise Let's revisit the previous example: if we sum all integer numbers starting from `n = 1`, what is the first value of `n` for which the sum of all integers `1 + 2 + 3 + ... + n` exceeds 100? **This time, initialize `n` to `1` and use `break` to exit the `while` loop at the first value of `n` for which the sum exceeds 100.**

2.10 Functions

Functions are program routines that optionally accept input arguments, perform some computation, and optionally return output arguments.

Let's define a simple function that returns the maximum of two numbers.

```
In [1]: def max(a, b):
   ...:     """Returns the max of two numbers."""
   ...:     if a >= b:
   ...:         return a
   ...:     else:
   ...:         return b
```

The **function definition** is introduced by the keyword `def`, followed by the name of the function (`max`) and an optional list of input arguments (here `a` and `b`). Parentheses are mandatory, even if there are no input arguments. All the statements that belong to the function must be indented. The first statement of the function body can optionally be a **string literal** (delimited by `""" ... """`; this string literal is the function's documentation string, or *docstring* [13].

A function can return one or more output arguments using the keyword `return`.

```
In [2]: c = max(5, 7)
In [3]: c

Out[3]: 7
```

2.10.1 Types of function arguments

In Python, the arguments of a function can be divided into several categories: **required**, **positional**, **optional with default values**, and **keyword arguments**. The slightly confusing aspect of this argument classification is that the type of an argument is determined not only by its definition in the function signature, but (mostly) by how it is used in the function call[14].

For instance, consider the following `fun_with_args` function:

```
In [4]: def fun_with_args(a, b=10, c=20):
   ...:     """Function to explain the possible argument types."""
   ...:     return a + b + c
```

[13] There are several conventions and standards on how doctrings should be formatted. See for example: https://www.python.org/dev/peps/pep-0257/.

[14] There is even more flexibility in how parameters can be handled, but we do not need to discuss it for our purposes. More information can be found here: https://docs.python.org/3/tutorial/controlflow.html#special-parameters.

2.10.1.1 Positional arguments

If the function `fun_with_args` is called without specifying the names of the arguments, the parameters are **positional**.

```
In [5]: fun_with_args(1, 2, 3) # a=1, b=2, c=3
```

All parameters are determined by their respective **positions** in the function call.

2.10.1.2 Keyword arguments

If we explicitly specify the argument **names**, the parameters become **keyword parameters**.

```
In [6]: fun_with_args(a=1, b=2, c=3) # a=1, b=2, c=3
```

2.10.1.3 Arguments with default values

We can omit arguments that have a **default values** assigned to them.

```
In [7]: fun_with_args(1) # a=1, b=10 (default), c=20 (default)
```

2.10.1.4 Mixed

Finally, we can combine everything.

```
In [8]: fun_with_args(1, c=3) # a=1, b=10 (default), c=3
```

Here, a is a required argument filled positionally, c is a keyword argument, and b is left at its default value. Note that when using both positional and keyword arguments in a function call, all positional arguments must come before any keyword arguments.

```
In [9]: fun_with_args(c=3, b=2, 1) # ERROR!

    fun_with_args(c=3, b=2, 1)
                             ^

SyntaxError: positional argument follows keyword argument
```

In the previous call, it is not clear to which parameter the value 1 should be assigned: since it is not explicitly named, it is expected to be filled positionally, but it is passed in the wrong place.

In summary, the classification of an argument as required, positional, having a default value, or being a keyword argument is largely determined by how the function is called, not solely by how its parameters are defined.

2.10.2 Multiple return values and tuple unpacking

Functions can return multiple values packed into tuples. Consider the function `fun`:

```
In [10]: def fun(a, b, c):
    ...:     """Function with two output arguments."""
    ...:     d = a + c
    ...:     e = b * c
    ...:     return (d, e)    # This notation is also valid: return d, e
```

Here, **tuple unpacking** allows us to directly assign returned values to variables o and p:

```
In [11]: o, p = fun(3, 5, 7)
In [12]: o

Out[12]: 10

In [13]: p

Out[13]: 35
```

2.10.3 A note on single-element tuples

To define a tuple with a single element, we use a trailing comma:

```
In [19]: u = (1, )
In [20]: type(u)
```

```
Out[20]: tuple
```

```
In [21]: v = (1)  # No comma!
In [22]: type(v)
```

```
Out[22]: int      # The parentheses were ignored!
```

2.11 Introduction to Object-Oriented Programming

In OOP, a **class** serves as a blueprint for **objects**. These objects encapsulate data (**attributes**) and behavior (**methods**). Essentially, classes allow us to create Python custom types, or **instances**, that can model real-world entities.

2.11.0.1 Defining a class

Consider the Car class with default attributes make and year:

```
In [1]: class Car:
    ...:     """A simple class representing a Car."""
    ...:     def __init__(self, make="VW", year=2021):
    ...:         """Constructor."""
    ...:         self.make = make
    ...:         self.year = year
```

Once defined, Car is a new type that Python recognizes. Instances can be created as follows:

```
In [2]: car = Car()
```

We can query the Car attributes with the dot notation:

```
In [3]: car.make
```

```
Out[3]: 'VW'
```

```
In [4]: car.year
```

```
Out[4]: 2021
```

2.11.1 The constructor method

The special function __init__() initializes the object's attributes. It takes a self-referential argument, conventionally named self, and any additional arguments necessary for initialization.

```
In [5]: vw = Car(make="VW", year=2021)
In [6]: ferrari = Car(make="Ferrari", year=1983)
```

2.11.2 Adding functionality

We can enhance Car with methods that model its behavior and alter its state:

```
In [7]:
class Car:
    """A simple class representing a Car."""
    def __init__(self, make="VW", year=2021):
        """Constructor."""
        self.make = make
        self.year = year
        self.is_on = False
        self.speed = 0.0
    def __repr__(self):
        """String representation of the object."""
        if self.is_on:
            return f"{self.make} ({self.year}): engine is on and speed is {self.speed}."
        else:
            return f"{self.make} ({self.year}): engine is off."

    def turn_on(self):
        """Turn on the Car."""
        self.is_on = True

    def turn_off(self):
        """Turn off the Car."""

        # We can only turn off the car, if the speed is 0!
        if self.speed > 0.0:
            print("Stop the car first!")
        else:
            self.is_on = False

    def accelerate(self, new_speed):
        """Accelerate to the new speed."""

        # We can only accelerate if the Car is on
        if not self.is_on:
            print("Turn on the car first!")
        else:
            self.speed = new_speed
            if self.speed > 250.0:
                print("Maximum car speed (250.0) reached!")
                self.speed = 250.0

    def brake(self):
        """Brake."""

        # We can only brake if the car is on and moving!
        if not self.is_on:
            print("The car is off!")
            return

        if self.speed == 0.0:
            print("The car is not moving!")
            return

        # Okay, we can brake
        self.speed = 0.0
```

2.11.3 Testing the class

Finally, let's test our class to ensure that it behaves as expected:

```
In [8]: car = Car()
In [9]: car    # This calls car.__repr__()
```

```
Out[9]: VW (2021): engine is off.
```

```
In [10]: car.accelerate(100.0)
```

```
Out[10]: Turn on the car first!
```

```
In [11]: car.turn_on()
In [12]: car

Out[12]: VW (2021): engine is on and speed is 0.0.

In [13]: car.brake()

Out[13]: The car is not moving!

In [14]: car.accelerate(100.0)
In [15]: car

Out[15]: VW (2021): engine is on and speed is 100.0.

In [16]: car.accelerate(300.0)

Out[16]: Maximum car speed (250.0) reached!

In [17]: car

Out[17]: VW (2021): engine is on and speed is 250.0.

In [18]: car.turn_off()

Out[18]: Stop the car first!

In [19]: car.brake()
In [20]: car

Out[20]: VW (2021): engine is on and speed is 0.0.

In [21]: car.turn_off()
In [22]: car

Out[22]: VW (2021): engine is off.
```

2.11.4 Summary

In essence, a class allows us to model real-world entities in code. It maintains an internal state, provides methods to alter that state, and ensures valid states through programmed behaviors.

2.12 Python modules and packages

Python offers **modules** and **packages** to break down large programs into manageable, reusable parts.

2.12.1 Modules

In Python, a **module** serves as a container for code that can be reused across various programs. A module can be either built-in, part of Python's standard library, or user-defined, that is created by individual developers. Regardless of its origin, a module becomes accessible in your program through the `import` statement. For instance, we can import a `my_module` module as follows:

```
import my_module
```

2.12.1.1 Module search path

When we import a module, Python looks for it in several locations:

- The directory where the script runs or the current directory in an interactive session (*e.g.*, Python or IPython).
- Directories listed in the `PYTHONPATH` environment variable.
- Installation-dependent directories or (conda) virtual environments.

2.12.1.2 Interacting with a module

Assuming we wrote a module named `my_module.py` with various definitions:

```
# my_module.py
s = "This is a string from my_module."
a = [0, 1, 2, 3]

def my_fun(arg):
    print(arg)

class My_Class:
    def __init__(self, id=1):
        self.id = id
        print(self.id)
```

we can interact with it as follows:

```
# IPython started from /home/aaron
In [1]: import my_module
In [2]: my_module

Out[2]: <module 'my_module' from '/home/aaron/my_module.py'>

In [3]: my_module.s

Out[3]: 'This is a string from my_module.'

In [4]: my_module.a

Out[4]: [0, 1, 2, 3]

In [5]: my_module.my_fun(10)

Out[5]: 10

In [6]: m = my_module.My_Class()

Out[6]: 1
```

Notice how we prepend the module name to the objects that it contains. Indeed, the following call fails:

```
In [7]: m = My_Class()

NameError: name 'My_Class' is not defined
```

2.12.1.3 Namespace isolation

When imported, the objects from a module reside in their own **namespace**, thus avoiding any naming conflicts. It is indeed a good (defensive) programming strategy to import modules and target their contents with the `module.attribute` syntax. However, it is possible to import objects directly into the caller's namespace as follows:

```
In [8]: from my_module import my_fun, My_Class
In [9]: my_fun(12)

Out[9]: 12
```

2.12.1.4 Caution: overwriting namespace

Be cautious, as importing objects directly into the caller's namespace can overwrite existing objects.

```
In [10]: s = "This is the variable s in the base namespace."
In [11]: s

Out[11]: 'This is the variable s in the base namespace.'

In [12]: from my_module import s
In [13]: s
```

```
Out[13]: 'This is a string from my_module.'
```

To import everything from a module, use:

```
In [14]: from my_module import *
```

Use this with care. Some modules are quite extensive and will add a lot of names into the namespace (with the risk of *name collision*). Also, importing many unneeded objects will take time and use up memory unnecessarily.

2.12.1.5 Aliasing

It is also possible to import objects with alternate names (aliases):

```
In [15]: from my_module import My_Class as MC
In [16]: import my_module as mmod
```

2.12.1.6 Conditional execution in modules

Python modules can incorporate testing or initialization code. This is commonly managed by the `__name__` attribute within the module. For instance, consider the last three lines added to `my_module`:

```python
# my_module.py
s = "This is a string from my_module."
a = [0, 1, 2, 3]

def my_fun(arg):
    print(arg)

class My_Class:
    def __init__(self, id=1):
        self.id = id
        print(self.id)

# These are the three new lines we added to my_module.py
print("Let's test our code:")
m = My_Class(id=42)
assert(m.id == 42)
```

Upon importing this module, the print statement and the following lines will execute:

```
In [17]: import my_module
```

```
Let's test our code:
42
```

This behavior is helpful for initialization routines, but sometimes you may want this code to execute only when the module is run as a standalone file, not when it is imported as a module. This is where the `__name__` attribute (double underscore[15] before and after the string `name`) comes into play.

In Python, the `__name__` attribute is set to the module's name when we import it. However, when we run the module directly as a script, `__name__` is set to `"__main__"`. We can leverage this to conditionally execute code:

```python
# my_module.py
# ... (previous code)
if __name__ == "__main__":
    print("Let's test our code:")
    m = My_Class(id=42)
    assert(m.id == 42)
```

Now, the code inside the `if __name__ == "__main__":` block will only run when the module is executed as a script, not when imported.

[15] This is pronounced *dunder name*, for double underscore name.

```
In [18]: import my_module  # No output, since __name__ is "my_module"
```

If you run the module directly:

```
$ python my_module.py  # __name__ is now "__main__"
```

```
Let's test our code:
42
```

This allows us to segregate testing or initialization code from the functional elements of a module, providing more control over its behavior.

2.12.2 Packages

When multiple modules share a thematic or functional connection, Python packages offer a hierarchical way to organize them. For instance, we might have various modules for image processing and statistical analysis. While they can co-exist in a single directory, separating them into distinct packages improves code organization.

2.12.2.1 Creating packages

A Python package is essentially a directory that may contain a special `__init__.py` file along with one or more module files[16]. The folder hierarchy represents the package and subpackage structure.

```
images                       # Package images
|    |__ consts.py           # Module consts
|    |__ filter              # Subpackage filter
|    |    |__ frequency.py   # Module frequency
|    |    |__ linear.py      # Module linear
|    |    |__ nonlinear.py   # Module nonlinear
|    |__ register            # Subpackage register
|    |    |__ register.py    # Module register
|    |__ segment             # Subpackage segment
|         |__ segment.py     # Module segment
stats                        # Package stats
|__ tests.py                 # Module tests
|__ visual.py                # Module visual
```

2.12.2.2 Importing from packages

Packages allow for structured import statements. For instance:

```
In [1]: from stats import visual     # `visual` is in the namespace
In [2]: import images.filter.linear # `images.filter.linear` is in the namespace
```

2.12.2.3 Package initialization: `__init__.py`

The optional `__init__.py` file is executed when a package is imported. This file often contains package-level variables or initializations. It's common to define a package's version here.

```
|__ images
|    |__ __init__.py
```

with the following content:

```
__version__ = "0.0.1"
```

We can now access the package version as follows:

```
In [3]: import images
In [4]: images.__version__
```

```
Out[4]: '0.0.1'
```

[16] The `__init__.py` file was mandatory Python 2 bit is optional in Python 3.

2.12.2.4 Importing within subpackages

Modules in subpackages can use either absolute or relative imports to access objects in other subpackages or parent packages.

```
from images.consts import UINT16_MAX  # Absolute import
from ..consts import UINT16_MAX       # Relative import
```

In relative imports, `..` refers to the parent package, and `.` refers to the current package. Note that the use of path separators like / is not allowed in relative imports (*e.g.*, `../consts` is incorrect).

3 Essential third-party libraries

3.1 NumPy

NumPy (https://numpy.org) adds support for high-performance multi-dimensional arrays to Python, along with a large collection of high-level mathematical functions to operate on these arrays. NumPy is open-source software and has many contributors. Almost all scientific libraries for Python use NumPy arrays as their back-end data structure, rendering them all fully compatible with each other. The QuickStart tutorial (https://numpy.org/doc/stable/user/quickstart.html) gently introduces the fundamental aspects of NumPy. Here, we will look at some of those basics.

First of all, NumPy must be imported into Python. It is convention to import is with an alias, `np`.

```
In [1]: import numpy as np
```

There are many ways to create NumPy[17] arrays.

```
# From Python lists
In [2]: a = np.array([1, 2, 3, 4, 5, 6])  # 1D array
In [3]: a

Out[3]: array([1, 2, 3, 4, 5, 6])

In [4]: b = np.array([[1, 2, 3], [4, 5, 6]])  # 2D array
In [5]: b

Out[5]:
array([[1, 2, 3],
       [4, 5, 6]])

# Specifying the final dimensions (shape), type and filling value
In [6]: c = np.zeros((3, 2, 4), dtype=np.float64)  # 3D array (z, y, x)
In [7]: c

Out[7]:
array([[[0., 0., 0., 0.],
        [0., 0., 0., 0.]],
       [[0., 0., 0., 0.],
        [0., 0., 0., 0.]],
       [[0., 0., 0., 0.],
        [0., 0., 0., 0.]]])

In [8]: c = np.ones((1, 2), dtype=int) # 1 row, 2 columns
In [9]: c

Out[9]:
array([[1, 1]])

# Using specific distributions
In [10]: n = np.random.randn(1000)  # Normal distribution N(0, 1)
```

NumPy arrays have many operations associated to them. For instance, we can calculate mean and standard deviation of the `n` array above like this:

[17] See https://numpy.org/doc/stable/reference/arrays.ndarray.html for the full API.

```
In [11]: n.mean()

Out[11]: -0.007093885346150171  # Close to the expected value 0.0

In [12]: n.std()

Out[12]: 0.9869874098207414     # Close to the expected value 1.0
```

Indexing and slicing NumPy arrays works as with standard Python arrays:

```
In [13]: a = np.arange(5, 11)
In [14]: a

Out[14]: array([5, 6, 7, 8, 9, 10])

In [15]: a[0]

Out[15]: 5

In [16]: a[-1]     # Last element in the array

Out[16]: 10

In [17]: a[::2]    # From beginning to end with step 2

Out[17]: array([5, 7, 9])

In [18]: a[::-1]    # Cool way to reverse an array

Out[18]: array([10,  9,  8,  7,  6,  5])
```

Arithmetic operations on NumPy array are applied element-wise:

```
In [19]: a = np.array([1, 2, 3])
In [20]: b = np.array([4, 5, 6])
In [21]: a * b

Out[21]: array([ 4, 10, 18])
```

Matrix multiplications are possible using the operator `@`. For a matrix multiplication we need a to be of shape `(3, 1)` and b to be `(1, 3)`: `a @ b` results in `3, 3` matrix (outer product). Alternatively, if a is `(1, 3)` and b is `(3, 1)`, the result is a scalar (`(1, 1)`) and the operation reduces to the dot product of the two vectors.

```
In [22]: a.reshape((3, 1)) @ b.reshape((1, 3))

Out[22]:
array([[ 4,  5,  6],
       [ 8, 10, 12],
       [12, 15, 18]])

In [23]: a.reshape((1, 3)) @ b.reshape((3, 1))

Out[23]: array([[32]])
```

Finding elements in arrays is possible using the `where()` function. Let's create a small 1D vector a to work with:

```
In [24]: a = np.arange(11, 20)
In [25]: a

Out[25]: array([11, 12, 13, 14, 15, 16, 17, 18, 19])
```

Let's find the indices of the elements of a that satisfiy some simple condition:

```
In [26]: indices, = np.where(a > 15)  # Returns a tuple x, y
In [27]: indices

Out[27]: array([5, 6, 7, 8])

# Let's check
In [28]: a[indices]

Out[28]: array([16, 17, 18, 19]) # Indeed, all > 15!
```

Notice that np.where returns a tuple of indices for each dimension of the array. If the array is 1D, only one vector if indices is returned, but still packed in a tuple!

We can also combine logical expressions[18]:

```
In [29]: indices, = np.where((a > 13) & (a <= 15))
In [30]: indices

Out[30]: array([3, 4])

# Let's check
In [31]: a[indices]

Out[31]: array([14, 15]) # Indeed, 13<a[indices]<=15!
```

np.where also works in 2D and beyond.

```
In [32]: b = np.arange(1, 17).reshape(4, 4)
In [33]: b

Out[33]:
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16]])

# Find some values
In [34]: indices = np.where((b > 12) & (b < 15))
In [35]: indices

Out[35]: (array([3, 3]), array([0, 1]))

# The values that satisfy the conditions are (3, 0) and (3, 1)
In [36]: b[indices]

Out[36]: array([13, 14]) # Indeed, 12<b[indices]<15!
```

Logical indexing can also be used to select subsets of NumPy arrays:

```
In [37]: a >= 16

Out[37]: array([False, False, False, False, False,  True,  True,  True,  True])
```

We can directly use a boolean array to select elements from another array. Implicitly, the values of the array at the indices that correspond to the indices of the True values in the boolean array are used:

```
In [38]: a[a >= 16]

Out [38]: array([16, 17, 18, 19])
```

3.2 Pandas

pandas (https://pandas.pydata.org) is a library for Python for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series.

First of all, pandas must be imported into Python. It is convention to import is with an alias, pd.

```
In [1]: import pandas as pd
```

Important data structures in pandas are DataFrame and Series. A DataFrame consists of one or more Series.

```
In [2]: s = pd.Series([1, 3, 5, np.nan, 6, 8])
In [3]: s

Out[3]:
0    1.0
1    3.0
2    5.0
```

[18] Notice that the **vector** equivalents of and and or are & and |, respectively.

```
3    NaN
4    6.0
5    8.0
dtype: float64
```

Every value in a Series has an **index** associated to it (the numbers 0 to 5 on the left of the Series values above). `NaN` indicates a missing value.

A DataFrame consists of one or more Series organized in **columns**, with an index associated to them (not necessarily numeric).

```
In [4]: dates = pd.date_range("20210901", periods=6)
In [5]: dates

Out[5]:
DatetimeIndex(['2021-09-01', '2021-09-02', '2021-09-03', '2021-09-04',
               '2021-09-05', '2021-09-06'],
              dtype='datetime64[ns]', freq='D')

In [6]: df = pd.DataFrame(np.random.randn(6, 4), index=dates, columns=["A", "B", "C", "D"])
In [7]: df

Out[7]:
                   A         B         C         D
2021-09-01 -1.896984 -0.784408 -0.176049  0.678751
2021-09-02  0.036235  0.934565 -0.929419  0.511440
2021-09-03  0.308496  1.357357  0.422399 -1.228895
2021-09-04 -0.978040  0.555728 -1.982682 -0.289128
2021-09-05 -0.599076  1.144799 -1.416163  0.469280
2021-09-06 -0.462621  2.194998  0.596607 -1.163095
```

Often, a DataFrame is imported into pandas from a text file, an Excel datasheet, a MySQL database, or an URL.

```
In [8]: df = pd.read_csv("iris.csv", sep=",")
In [9]: df.columns

Out[9]:
Index(['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',
       'petal width (cm)', 'species'],
      dtype='object')

In [10]: df.columns = ['SL', 'SW', 'PL', 'PW', 'S']  # Save some space
In [11]: df.head()   # Show the first 5 rows

Out[11]:
    SL   SW   PL   PW       S
0  5.1  3.5  1.4  0.2  setosa
1  4.9  3.0  1.4  0.2  setosa
2  4.7  3.2  1.3  0.2  setosa
3  4.6  3.1  1.5  0.2  setosa
4  5.0  3.6  1.4  0.2  setosa

In [12]: df.describe()  # Summmarize the dataframe (only numeric fields)

Out[12]:
               SL          SW          PL          PW
count  150.000000  150.000000  150.000000  150.000000
mean     5.843333    3.057333    3.758000    1.199333
std      0.828066    0.435866    1.765298    0.762238
min      4.300000    2.000000    1.000000    0.100000
25%      5.100000    2.800000    1.600000    0.300000
50%      5.800000    3.000000    4.350000    1.300000
75%      6.400000    3.300000    5.100000    1.800000
max      7.900000    4.400000    6.900000    2.500000
```

We can access subsets of a DataFrames in a series of ways.

```
In [13]: df["SL"]   # Get the full column (Series) "SL"
```

```
Out[13]:
0      5.1
1      4.9
2      4.7
3      4.6
4      5.0
      ...
145    6.7
146    6.3
147    6.5
148    6.2
149    5.9
Name: SL, Length: 150, dtype: float64
```

```
df["SL"][:5]   # Get the first 5 rows of "SL"
```

```
Out[11]:
0    5.1
1    4.9
2    4.7
3    4.6
4    5.0
Name: SL, dtype: float64
```

```
In [14]: df[df["S"] == "versicolor"][["SL", "SW"]][:5] # First 5 versicolors, SL and SW
```

```
Out[14]:
     SL   SW
50  7.0  3.2
51  6.4  3.2
52  6.9  3.1
53  5.5  2.3
54  6.5  2.8
```

To update cells in place, one can use the `loc` method.

```
In [15]: df.loc[df["S"] == "setosa", "S"] = "iris setosa"
In [16]: df.head()
```

```
Out[16]:
    SL   SW   PL   PW            S
0  5.1  3.5  1.4  0.2  iris setosa
1  4.9  3.0  1.4  0.2  iris setosa
2  4.7  3.2  1.3  0.2  iris setosa
3  4.6  3.1  1.5  0.2  iris setosa
4  5.0  3.6  1.4  0.2  iris setosa
```

In contrast, the method `iloc` uses the index value to target the cells.

```
In [17]: df.iloc[1, 0] = 100
In [18]: df.head()
```

```
Out[18]:
      SL   SW   PL   PW            S
0    5.1  3.5  1.4  0.2  iris setosa
1  100.0  3.0  1.4  0.2  iris setosa
2    4.7  3.2  1.3  0.2  iris setosa
3    4.6  3.1  1.5  0.2  iris setosa
4    5.0  3.6  1.4  0.2  iris setosa
```

To convert a Series object to a NumPy array, one can use the `values()` method of the Series object.

```
In [19]: n = df["PL"].values
In [20]: n
```

```
Out[20]: array([1.4, 1.4, 1.3, 1.5, 1.4, 1.7, ...]) # Cut to save space
```

```
In [21]: type(n)
```

```
Out[21]: numpy.ndarray
```

A very handy functionality of Pandas is the possibility to group elements of a DataFrame by one or more columns. Before we do that, let's restore the original value of the `iris setosa` element we modified above:

```
In [22]: df.iloc[1, 0] = 4.9  # Let's restore the original value
```

As an example, we can now use the DataFrame method `groupby` to group all entries by species, using the `S` column:

```
In [23]: df_by_species = df.groupby("S")
```

Now our queries will not relate to individual entries, but to the groups:

```
In [24]: df_by_species.describe()
```

```
Out[24]:
               SL                                                    SW
            count   mean       std  min    25%  50%  75%  max count ...
S
iris setosa  50.0  5.006  0.352490  4.3  4.800  5.0  5.2  5.8  50.0 ...
verginica    50.0  6.588  0.635880  4.9  6.225  6.5  6.9  7.9  50.0 ...
versicolor   50.0  5.936  0.516171  4.9  5.600  5.9  6.3  7.0  50.0 ...

[3 rows x 32 columns]
```

```
In [25]: df_by_species["SL"].mean()
```

```
Out[25]:
S
iris setosa    5.006
verginica      6.588
versicolor     5.936
Name: SL, dtype: float64
```

We can also iterate over the subsets of the DataFrame that belong to each group:

```
In [26]: for species, frame in df.groupby("S"):
    ...:     print(f"{species}\n{frame.head(3)}")
```

```
iris setosa
    SL   SW   PL   PW       S
0  5.1  3.5  1.4  0.2  setosa
1  4.9  3.0  1.4  0.2  setosa
2  4.7  3.2  1.3  0.2  setosa
verginica
      SL   SW   PL   PW          S
100  6.3  3.3  6.0  2.5  verginica
101  5.8  2.7  5.1  1.9  verginica
102  7.1  3.0  5.9  2.1  verginica
versicolor
     SL   SW   PL   PW           S
50  7.0  3.2  4.7  1.4  versicolor
51  6.4  3.2  4.5  1.5  versicolor
52  6.9  3.1  4.9  1.5  versicolor
```

Finally, to write a DataFrame object to a `.csv` file, we can use the `DataFrame.to_csv()` method:

```
In [25]: df.to_csv("filename.csv", index=False)
```

By default, to_csv() will save the DataFrame's index as an additional column to the .csv file. In most cases, however, this behavior is not necessary and can be disabled by passing the argument index=False to the function. Please note that DataFrameGroupBy objects created by the .groupby() method (such as df_by_species above) cannot be directly saved using to_csv().

3.3 Matplotlib

Matplotlib (https://matplotlib.org/) is a (large) plotting library for Python that can create publication-quality figures. The gallery (https://matplotlib.org/stable/gallery/index.html) gives a nice overview of the types of plots that can be generated with Matplotlib.

We usually import Matplotlib as follows:

```
In [1]: import matplotlib.pyplot as plt
```

The simplest way to create a plot is as follows:

```
# plot_simply.py

# Prepare the data
x = np.linspace(-10, 10, 100)    # import numpy as np, first
y = np.cos(x)

# Plot
plt.plot(x, y, label='y(x)')

# Add axes labels
plt.xlabel("x")
plt.ylabel("y")

# Add title
plt.title("y = cos(x)")

# Add legend
plt.legend(loc="best")

# Show (this is not necessary in a Jupyter notebook)
plt.show()
```

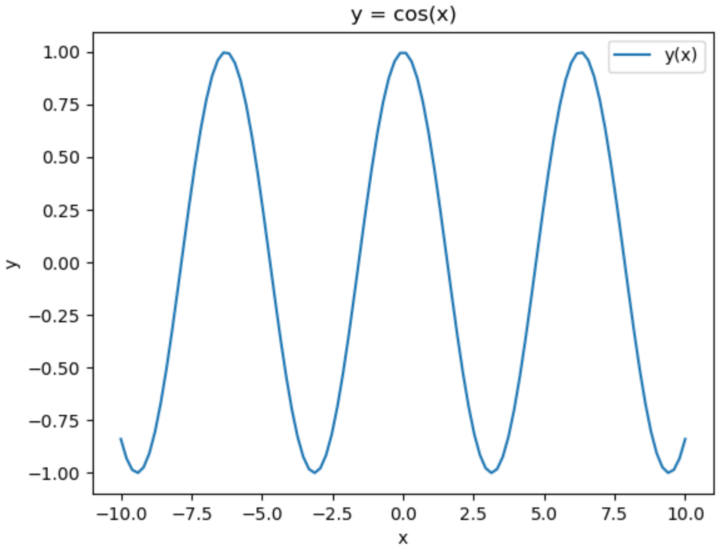


Figure 10: Our first plot

If we want better control on the elements of our plots, we can switch to a more object-oriented use of Matplotlib. In particular, we can create a **Figure** with one or more sets of **axes** and then explicitly target them.

```
fig = plt.figure(figsize=(20,10))
ax1 = fig.add_subplot(1, 2, 1)  # 1 row, 2 columns, 1st subplot
ax2 = fig.add_subplot(1, 2, 2)  # 1 row, 2 columns, 2nd subplot
```

Alternatively, we can combine the calls:

```
fig, ax = plt.subplots(1, 2, figsize=(20,10))
ax1, ax2 = ax    # Extract the axes
```

Now we can plot data and change the axes individually:

```
# plot_complex.py

import matplotlib

fig, ax = plt.subplots(1, 2, figsize=(10,5))
ax1, ax2 = ax

ax1.bar([1990, 1995, 2000, 2005, 2010],
        [103, 127, 118, 134, 118],
        width=3)
ax1.set_ylim((100, 135))
ax1.set_title("Average yearly production")
ax1.grid(True, which='major', color='gray',
         linestyle='-', linewidth=0.5)
ax1.minorticks_on()
ax1.grid(True, which='minor', color='gray',
         linestyle='--', linewidth=0.2)
ax1.set_xlabel("Year")
ax1.set_ylabel("Sales [millions US$]")

ax2.plot([0, 1, 2, 3, 4, 5], [3, 5, 3, 1, 4, 6],
         color="r", linestyle='--', linewidth=2,
         marker='.', markersize=18)
ax2.add_line(
    matplotlib.lines.Line2D((0, 5), (4, 4),
                            linestyle='-.',
                            linewidth=0.5,
                            color="black"))
ax2.add_line(
    matplotlib.lines.Line2D((0, 5), (2, 2),
                            linestyle='-.',
                            linewidth=0.5,
                            color="black"))
ax2.set_xlabel("x")
ax2.set_ylabel("y")
ax2.annotate("Some cool data point",
             xy=(1.0, 5.0),
             xytext=(1.5, 5.2),
             arrowprops={"arrowstyle": "->"},
             fontsize=12)
```

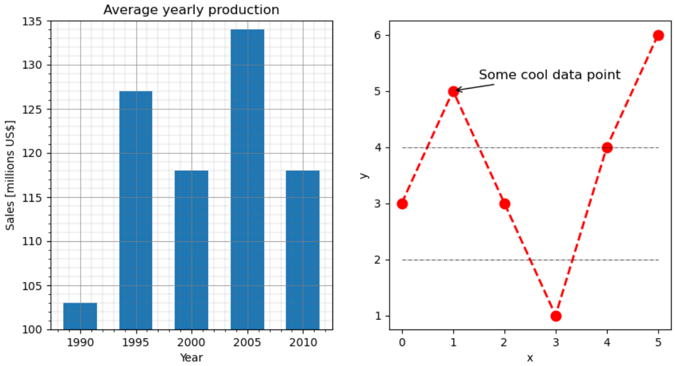


Figure 11: A more complex plot

In the second plot of the previous figure, we added three distinct lines to the same set of axes. We can do this by first creating a new `matplotlib.lines.Line2D` object `line`, and then add it to the axes with `ax.add_line(line)`.

Finally, to save a figure to file, one can use:

```
# Current, active figure
plt.savefig("/path/to/figure.png", dpi=100)

# Specific figure
fig.savefig("/path/to/figure.png", dpi=100)
```

The function `savefig` takes many parameters, as explained on https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.savefig.html. In particular, `dpi` allows to increase the resolution of the plot for higher-quality figures that can be submitted for publication. Also, either specifying the `format` argument or just by setting the correct extension to the file name `fname`, one can switch from image formats (such as `png` or `jpg`) to vector graphics formats such as `svg` (that can be edited in publishing software such as Adobe Illustrator).

3.3.1 Backends

Matplotlib will render its plots using different backends depending on where it is called from. In Jupyter notebooks, plots are rendered directly in the notebook whenever a cell containing Matplotlib calls is executed. The backend in Jupyter notebooks can also be set explicitly with the `%matplotlib` magic word: `%matlotlib inline` displays the plots right after the executed cells.

If other backends are used, an explicit `plt.show()` call may be needed to display the generated plot (*e.g.*, if a window must be opened). A complete list of backends can be found here: https://matplotlib.org/stable/api/index_backend_api.html. Depending on where Python is installed, a proper backend is usually configured; changing backends usually requires third-party libraries to be installed. The backend is changed by calling:

```
import matplotlib
matplotlib.use('TkAgg') # Or any of the other backends
```

before any plots are created.

4 Image analysis libraries used

Among Python's most comprehensive, high-performance, and high-quality image processing libraries, we can count **scikit-image**, **scipy.ndimage**, **OpenCV**, **SimpleITK**, **Mahotas**, and **Pillow**. These libraries have a significant overlap in functionality but, depending on the specific application, each may be more performant or offer more and better algorithms than the others. In our course, we will predominantly use **scikit-image**, since it implements almost all algorithms we need to solve our image processing tasks and has excellent documentation, making it an ideal learning tool. However, we will complement it with scipy.ndimage and the small **iaf** library, which implements various commodity functions that simplify tasks that may be more complex to perform in scikit-image (and other libraries). In this section, we will give some information about scikit-image, scipy.ndimage and iaf. You can find more information on the other libraries in Appendix B.

4.1 scikit-image

Figure 12: scikit-image logo

scikit-image is a collection of algorithms for image processing. It aims to be the reference library for scientific image analysis in Python and is available free of charge and without restrictions. One of the essential pillars of scikit-image's mission is to promote education in image processing by offering extensive pedagogical documentation.

4.1.1 License

scikit-image is available free of charge under a very permissive license: https://scikit-image.org/docs/dev/license.html.

4.1.2 References

- Stéfan van der Walt, Johannes L. Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, Tony Yu and the scikit-image contributors. scikit-image: Image processing in Python. PeerJ 2:e453 (2014) https://dx.doi.org/10.7717/peerj.453

4.1.3 Documentation

Website: https://scikit-image.org/

Current documentation: https://scikit-image.org/docs/0.21.x/

4.1.4 Installation

To install `scikit-image` using `conda`, use:

```
$ conda install scikit-image
```

To install `scikit-image` using `pip`, use:

```
$ pip install scikit-image
```

4.2 scipy.ndimage

Figure 13: SciPy logo

SciPy is an extensive library for mathematics, science, and engineering; **scipy.ndimage** is a package from SciPy that focuses on multidimensional image processing. Many functions in scikit-image are wrappers around the lower-level scipy.ndimage algorithms.

4.2.1 License

SciPy is available free of charge under the **BSD License (BSD)**: https://github.com/scipy/scipy/blob/main/LICENSE.txt

4.2.2 Documentation

Website: https://scipy.org/

Current documentation: https://docs.scipy.org/doc/scipy/index.html

4.2.3 Installation

To install SciPy using conda, use:

```
$ conda install scipy
```

To install SciPy using pip, use:

```
$ pip install scipy
```

4.3 iaf

Figure 14: iaf logo

iaf is a small helper library that simplify some operations that would otherwise be more verbose or complex in scikit-image, Matplotlib, and other libraries.

4.3.1 License

iaf is released under the BSD-3 Clause license.

4.3.2 Code and documentation

Code: https://git.bsse.ethz.ch/pontia/iaf

Current documentation: https://ia-res.ethz.ch/docs/iaf/index.html

5 Basic concepts of image analysis

In the following sections, we will look at fundamental concepts and tools of image processing and analysis using the libraries presented in the previous section. You can follow along on the Jupyter notebook code/python/notebooks/basic_concepts.ipynb.

5.1 Data types

Images are addressed in computer memory as two- or three-dimensional arrays of pixel intensities. The third dimension may have different interpretations depending on the context. For example, for standard **RGB** images, the third dimension encodes the image's **R**(ed), **G**(reen), and **B**(lue) channels. However, in a microscopy experiment, the third dimension could store the emission intensity of fluorophores, the various planes of a three-dimensional acquisition, or a series of acquisitions in a time series.[19].

scikit-image uses high-performance NumPy arrays to store and process images. Since image intensities are a quantized version of the original analog signal, the data type of the Numpy array is chosen to accommodate the number of gray values used to represent the **dynamic range** of the discretized signal. The following table

[19] A multi-channel, multi-plane, multi-time point acquisition would need to be stored in a five-dimensional array!

summarizes a few commonly used detector dynamic ranges used in microscopy, the corresponding number of bits used by the computer to represent them, and the NumPy data type. Here, we only consider positive integer values[20], since they are most commonly found in digital images.

Detector dynamic range	Computer dynamic range	NumPy data type
0, ..., 255 (8 bits)	0, ..., 255 (8 bits)	`numpy.uint8`
0, ..., 4095 (12 bits)	0, ..., 65535 (16 bit)	`numpy.uint16`
0, ..., 65535 (16 bits)	0, ..., 65535 (16 bits)	`numpy.uint16`

When performing quantification, floating point values are required. Here, `NumPy` offers the types `numpy.float32` and `numpy.float64` (that corresponds to the native Python type `float`). Since real numbers can only be approximated by a finite number of bits, the possible range and the precision of the floating point data types differ. In both cases, small numbers can be approximated more precisely than large numbers.

	`np.float32`	`np.float64`
Min value	-3.4028235e+38	-1.7976931348623157e+308
Max value	3.4028235e+38	1.7976931348623157e+308
Precision at 1.0 (eps)	1.1920929e-07	2.220446049250313e-16
Precision at 1000.0	6.1035156e-05	1.1368683772161603e-13

The precision (or spacing) shows the minimum distance between two real numbers that will result in different binary representations. Smaller differences will collapse into the same bit sequence. For instance, if `a = 1000.0` is of type `float32`, the following holds true:

```
a = np.float32(1000)
b = a + np.float32(1e-4)  # 1e-4 is larger than precision
b == a
```

```
False
```

```
c = a + np.float32(1e-5)  # 1e-5 is smaller than precision
c == a
```

```
True
```

5.1.1 Limitations of bit representations

Operations on data types that cause values to overflow the boundaries of the data type that encodes them can be surprising and quite disruptive. Consider the following:

```
a = np.uint8(10)
b = np.uint8(20)
a - b
```

```
246   # Ops!
```

A value of `-10` cannot be stored in a `numpy.uint8` data type. Hence, the negative value **wraps around** the dynamic range to give a value of 246 instead! Modern Python interpreters will warn us with a message like this:

```
<ipython-input-85-09bd029d0285>:1: RuntimeWarning: overflow encountered in ubyte_scalars
  a - b
```

but the overflow may go unnoticed in a production environment and cause serious trouble. When applying operations on integer values, then, it is a good defensive strategy to **cast** the data to floating point and, if necessary, cast it back to integer in the end.

```
c = np.float32(a) - b    # Casting one is enough
c                        # c is of type np.float32
```

```
-10.0
```

[20] The `u` in `numpy.uint8` stands for `unsigned`. The corresponding `signed` data types are `numpy.int8` and `numpy.int16`, with the respective ranges `-128 .. 127` , and `-32768 .. 32767`.

When applying operations to images, we often want to clip the values that fall outside the dynamic range[21].

```
e = np.uint8([10, 20, 30])
f = np.uint8([20, 10, 15])
g = e.astype(np.float32) - f
g
```

```
array([-10.,  10.,  15.], dtype=float32)  # One negative value
```

```
g[g < 0] = 0             # Set negative values to 0
g = g.astype(np.uint8)  # Cast back
```

```
array([ 0, 10, 15], dtype=uint8)
```

5.2 Image reading and writing

For standard image file formats like .tif and .png, we can use the imread() function from scikit-image, as in the following example:

```
from skimage.io import imread

img = imread("image.tif")
print(img.shape, img.dtype)
```

```
(1068, 1212) uint16
```

To write an image back to disk, we can use:

```
from skimage.io import imsave

imsave("out_image.tif", img)
```

For proprietary file formats like Nikon .nd2, there are a few Python libraries that we can use. Here, we will use the iaf library, that provides the NikonND2Reader:

```
from iaf.io.readers import NikonND2Reader

reader = NikonND2Reader("file.nd2")
reader
```

```
NikonND2Reader("file.nd2")
  - Dimensions: (v=7, t=1, c=2, z=6, y=1024, x=1024)
  - Voxel size: (x=0.519600, y=0.519600, z=4.955000)
  - Series geometry: "czyx"
```

This example ND2 file contains v=7 series (*e.g.*, stage positions). We can load the first series (v=0) with:

```
stack = reader[0]
stack.shape
```

```
(2, 6, 1024, 1024)  # c, z, y, x
```

To iterate over all series in the file, we can use:

```
for img in reader:
    print(img.shape, img.mean())
```

```
(2, 6, 1024, 1024) 82.52957439422607
(2, 6, 1024, 1024) 112.75271670023601
(2, 6, 1024, 1024) 117.6698609193166
(2, 6, 1024, 1024) 117.19015145301819
(2, 6, 1024, 1024) 151.8092711766561
(2, 6, 1024, 1024) 138.3780381679535
(2, 6, 1024, 1024) 155.3497955004374
```

[21] One can query the min and max value of integer types with np.iinfo(np.uint8).min and np.iinfo(np.uint8).max. The equivalent for floating point values is np.finfo(). The iaf library has two commodity functions iaf.io.cast.safe_to_uint8 and iaf.io.cast.safe_to_uint16 that perform safe casting for us.

The reader objects exposes metadata information via properties:

Property	Explanation
reader.channel_names	Tuple of names for each of the acquisition channels
reader.filename	Full file name of the opened file
reader.geometry	Geometry for each series
reader.iter_axis	Axis over which the iteration occurs (one of "v" or "t")
reader.metadata	Processed file metadata (dictionary)
reader.num_channels	Number of channels
reader.num_planes	Number of planes (z levels)
reader.num_series	Number of series (acquisitions) in the file
reader.num_timepoints	Number of time points
reader.voxel_sizes	Voxel sizes in units (μm) (x, y, z)

Please notice that if a file contains more than one series, the reader will iterate over series, and iter_axis will be "v". Otherwise, the reader will iterate over time points, and iter_axis will be "t". "v" and "t" are the only iteration axes supported. In both cases, the geometry property will indicate the dimensionality of the **array** returned by the iterator (*e.g.*, if geometry is "czyx", the retuned data will be a multi-channel 3D stack of images, with the first dimension being the channel c and the second the plane z).

5.3 Colors

In this course, we will focus on gray-value images that represent the intensity of some signal (for instance, fluorescence emission). The only information stored in such an image is the intensity of the original signal at each pixel position. By default, intensity images are displayed as gray-scale images with black pixels mapped to the intensity 0 and white pixels mapped to 255 (for 8-bit images) or 65535 (for 16-bit images).

Even with gray-value images, we may sometimes use color to assign specific meaning to pixels or emphasize spatial relationships between different images. For example, in the panel below, mCherry, GFP, and DIC are three intensity images corresponding to two **fluorescence** and one **differential-interference-contrast** microscopy acquisition. The intensity value at each location is proportional to the strength of the emitted signal collected at that pixel and therefore carries quantitative value. Since mCherry, GFP, and DIC are three views of the same underlying sample (a neuron), we can increase their information content by assigning different colors and combining them into a **composite image** that helps visualize the spatial relation of the three channels. The iaf library offers the iaf.color.to_composite() to easily compose images. If mcherry, gfp and dic are three NumPy arrays containing the corresponding gray-value images, we can create a composite as follows:

```
from iaf.color import to_composite

cmp = to_composite(
    images=(mcherry, gfp, dic),
    colors=(Color.Red, Color.Green, Color.Gray)
)
```

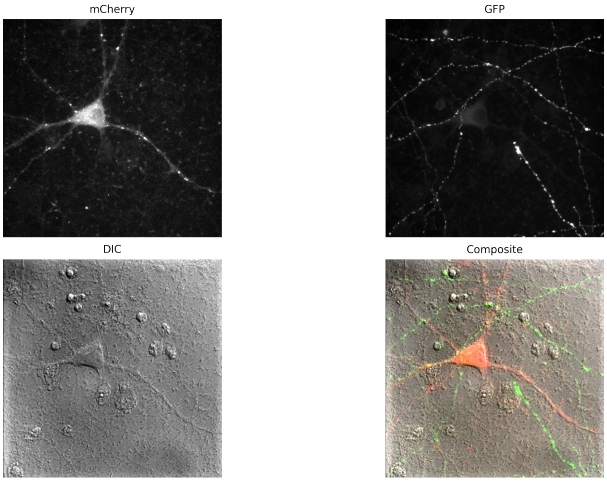


Figure 15: Three intensity channels and the corresponding composite image

In contrast to composite images, **RGB images** use a weighted sum of the three components **R**(ed), **G**(reen), and **B**(lue) to represent up to $256^3 = 16,777,216$ different colors. The intensity of each of the three channels does not represent any physical quantity; it is just one of the three components of the (r, g, b) tuple that represents a specific color for each pixel location.

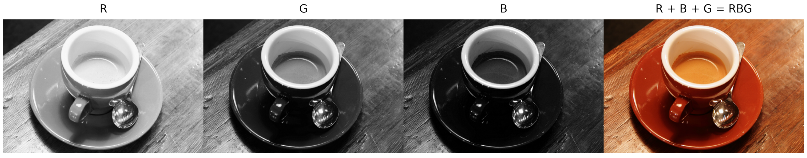


Figure 16: A color (*RGB*) image is the sum of three components (channels): Red, Green, and Blue

5.4 Histogram and histogram operations

Many fluorescence microscopy images tend to be predominantly background. The signal of interest is confined to smaller patches of brighter pixels. In the `FITC` image below, only cell membranes are fluorescently labeled, the signal is of low contrast, and details are difficult to appreciate.

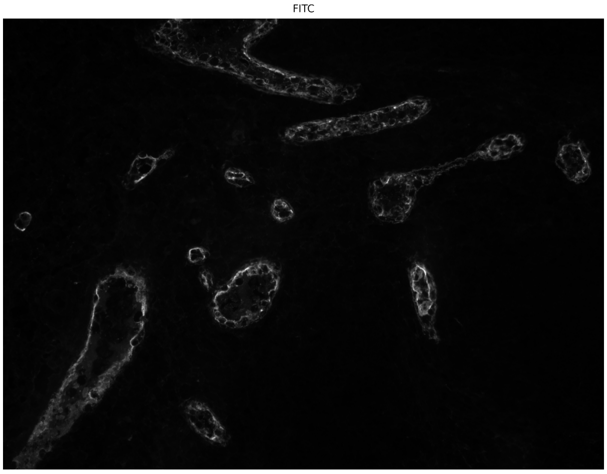


Figure 17: The FITC.tif image

The histogram of the `FITC` image below reveals that the vast majority of pixels have a very low intensity as expected since they are part of the image's background. The actual signal intensities are few and scattered over the whole range of the dynamic range. We can calculate the histogram of an image in `scikit-image` using the `histogram()` function from `skimage.exposure`.

```
from skimage.exposure import histogram

# Calculate the histogram over the data type source range
n, b = histogram(img, source_range="dtype")

# Calculate the log of the counts
norm_log_n = np.log(1.0 + n)
```

In the plot below, the solid red area shows the raw pixel intensity counts, while the superimposed semi-transparent area shows the logarithm of those counts. The logarithmic plot is handy for studying the distribution of the actual signal intensities in images that are dominated by the background. In particular, it helps spot cases of **signal saturation**. In an adequately quantized signal, the intensity counts should decay and reach zero well before reaching the maximum of the dynamic range. The dynamic range of the physical detector (*such as* the camera pixel) and the data type used to store the discretized signal (such as 8- or 16-bit integers) define the maximum intensity range of the final image. If the (log) histogram shows residual counts for the highest bin or even a suspiciously high peak, as in the plot below, we know that some of the highest signal intensities have been **saturated**.

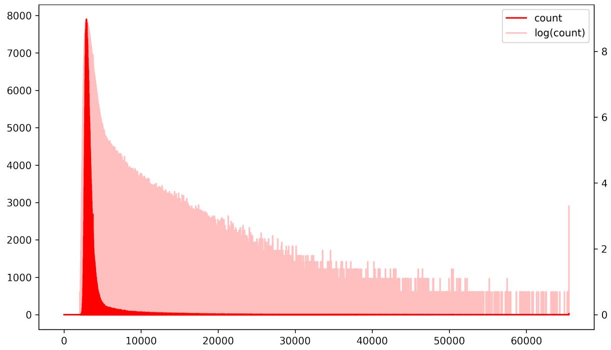


Figure 18: Histogram of FITC.tif

It is common to apply a **linear stretch** to the image intensities (a point operation) to increase the contrast in the image. Importantly, this is usually done only to enhance the visual clarity of the image and to accommodate the poor gray-value resolution of the human eyes: the original pixel intensities are **not permanently modified**! This fact is fundamental if we plan to use the image for quantitative analysis. In some (rare) cases, a linear stretch across a series of images is safe and justified. For example, many cameras have a dynamic range of 12 bits ($0 \dots 4095$), but the acquired images are stored with 16-bit dynamic range[22]. A linear stretch from 12 to 16 bits will not cause any loss of information and will still allow quantitative comparison across images (down to a constant multiplicative factor).

The simplest variant of linear stretch shifts the image towards zero by subtracting the minimum value and then stretches the histogram to cover the whole extent of the image's dynamic range. This operation can be done without loss of information if the range used to rescale the image contains all intensities in the starting image. For a 16-bit image, this would be:

$$I_s = 65535 \cdot \frac{I - \min(I)}{\max(I) - \min(I)}$$

In `scikit-image`, this can be easily done as follows[23]:

```
from skimage.exposure import rescale_intensity
```

[22] 12 bits correspond to 1.5 bytes. That is 4 bits of every second byte must be split among the neighbor bytes to calculate the correct intensity of the pixel they encode it. While this is certainly possible, it adds computational complexity and is avoided by *wasting* 4 bits for every pixel.

[23] As an exercise, try implementing it using `NumPy` operations only.

```
# Rescale between min(img) and max(img)
img_s = rescale_intensity(img)
```

It is common to use some percentile of the intensities to drop outliers (such as *dead* or *hot* pixels). We can pass an additional argument to rescale_intensity() that explicitly sets the lower and upper bound for normalization:

```
# Use the 5th and 95th intensity percentile
vmin, vmax = np.percentile(img, (5, 95))
img_s = rescale_intensity(img, (vmin, vmax))
```

The stretched image now looks like this:

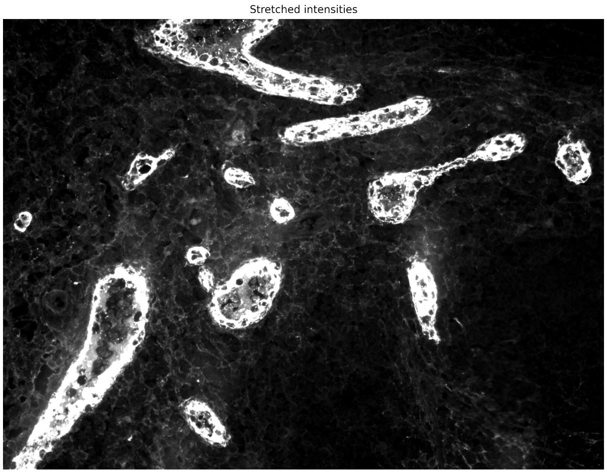


Figure 19: FITC.tif after linear stretching

The imshow() function from iaf.plot displays images with control on the intensity stretching via its auto_stretch and clip_percentile arguments, and without modifying the original image:

```
imshow(img, auto_stretch=True, clip_percentile=5.0)
```

The contrast in the cell membrane has definitely increased. However, if we plot the histogram and the logarithm of the histogram of the stretched image, we see that both low and high intensities are now heavily saturated.

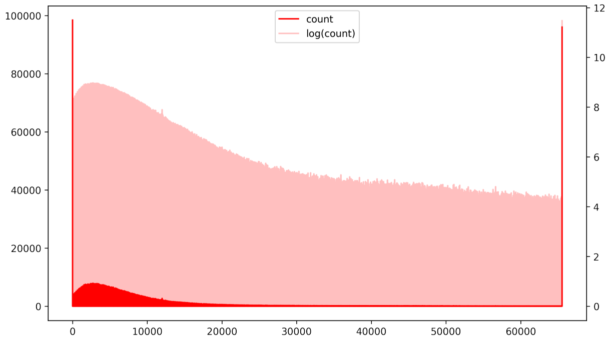


Figure 20: Histogram of FITC.tif after linear stretching

A very helpful tool to visualize the saturated pixels in the image is the **Hi-Lo** look-up table. Using iaf.color.get_hilo_cmap(), we can easily spot the saturated pixels as follows:

```
from iaf.color import get_hilo_cmap

imshow(img_s, cmap=get_hilo_cmap(65535))
```

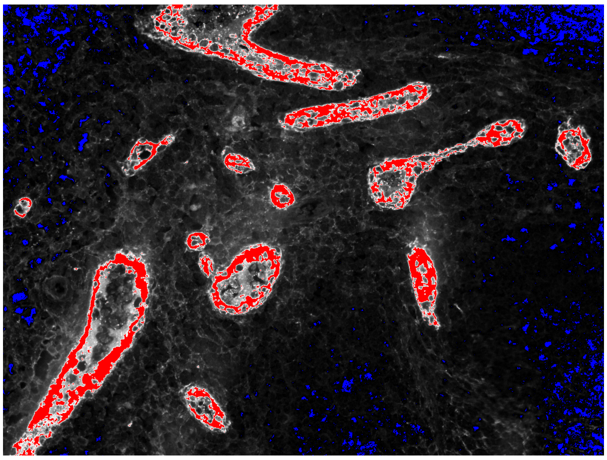


Figure 21: Linearly stretched FITC image with hi-lo look-up table.

The blue pixels saturate the lower bound of the dynamic range (that is, they have 0 intensity), while the red pixels saturate the upper bound (and have, in this case, an intensity of 65535). Beware of using such images for any quantitative analysis!

5.5 Filters

Images are filtered in the spatial domain by the convolution operation with a kernel whose weights are carefully selected to perform a specific transformation on the image's pixel. We will look at some examples using the `actin.tif` image below.

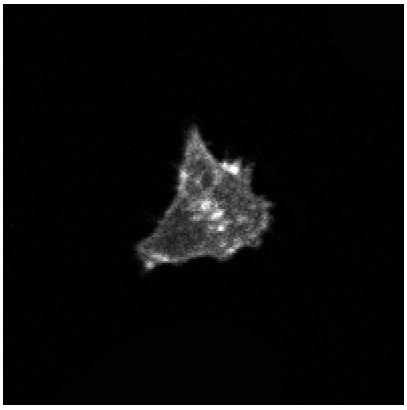


Figure 22: The actin.tif image

5.5.1 Average filter

We will start by running an average filter on the `actin.tif` image by defining our own $5x5$ kernel. We need a filter with homogeneous weights that sum up to 1.0.

$$h = \frac{1}{25} \cdot \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} \\ \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} \\ \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} \\ \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} \\ \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} \end{bmatrix}$$

In Python (using `NumPy`), we can define the kernel as follows:

```
k_5x5 = 1/25 * np.array([
    [ 1.0, 1.0, 1.0, 1.0, 1.0],
    [ 1.0, 1.0, 1.0, 1.0, 1.0],
    [ 1.0, 1.0, 1.0, 1.0, 1.0],
    [ 1.0, 1.0, 1.0, 1.0, 1.0],
    [ 1.0, 1.0, 1.0, 1.0, 1.0]
], dtype=np.float32)
```

or, more concisely:

```
k_5x5 = 1/25 * np.ones((5, 5), dtype=np.float32)
```

For comparison, we also create a filter with larger support:

```
k_9x9 = 1/81 * np.ones((9, 9), dtype=np.float32)
```

We apply the average filter by using the convolve() function from scipy.ndimage.

```
from scipy.ndimage import convolve

img_f_5x5 = convolve(img, k_5x5)
img_f_9x9 = convolve(img, k_9x9)
```

The results are shown in the figure below. Notice how kernels with larger support tend to blur the image stronger.

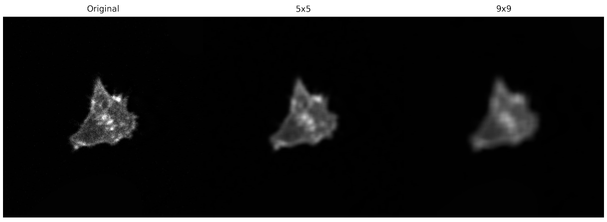


Figure 23: The support of the average filter

5.5.2 Gaussian filter

In contrast to the average filter, the Gaussian filter is better at preserving features of a given scale (or size). The support of the Gaussian kernel smoothly decays with the distance from the center pixel. The weights of the Gaussian kernel are calculated as follows:

$$G = e^{-\frac{x^2+y^2}{2\sigma^2}}$$

We can discretize the Gaussian filter explicitely:

```
In [1]: def gauss_2d(size, sigma):
    cy = size[0] // 2
    cx = size[1] // 2
    out = np.zeros(size, dtype=np.float32)
    for y in range(-cy, cy + 1):
        for x in range(-cx, cx + 1):
            r = y + cy
            c = x + cx
            out[r, c] = np.exp(-1.0 * ((x ** 2 + y ** 2) / (2 * sigma ** 2)))
    return out
```

and then apply it with the convolve() function as shown above. In practice, however, we will use the gaussian() function from scikit.filters:

```
from skimage.filters import gaussian

img_f_gs1 = gaussian(img, sigma=1.0)
img_f_gs3 = gaussian(img, sigma=3.0)
```

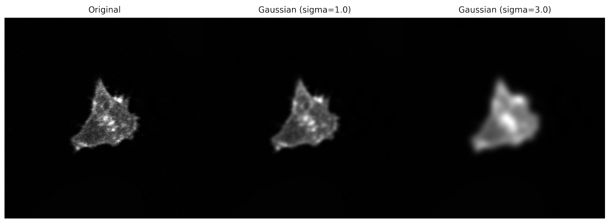


Figure 24: Gaussian filter

Higher values of `sigma` result in stronger smoothing of the image. Ideally, `sigma` should be chosen to match the size of the features of interest in the image or to suppress features (that is, noise) that is smaller than the support of the kernel.

5.5.3 Image derivatives as filters

Image derivatives can be approximated in a series of ways. Here, we will calculate the horizontal and vertical derivative by convolving the image with the `sx` and `sy` kernels defined below[24].

```
k_sx = np.array([
    [1, 0, -1],
    [1, 0, -1],
    [1, 0, -1]
], dtype=np.float32)

# k_sy is the transpose of k_sx
k_sy = k_sx.T

img_f_sx = convolve(img.astype(np.float32), k_sx)
img_f_sy = convolve(img.astype(np.float32), k_sy)
```

Vertical edges are preserved predominantly by `sx` and horizontal edges are preserved predominantly by `sy`. Edges at angles close to 45º have similar responses with both filters.

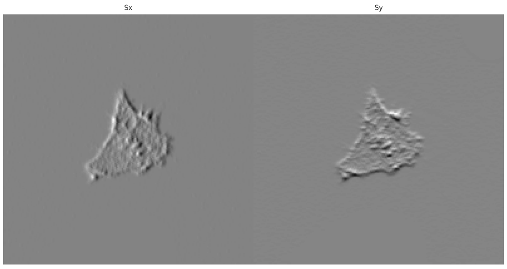


Figure 25: First derivative in x and y direction

The Laplacian is a discretization of the second-order derivative of the image and highlights regions of rapid intensity change. Two commonly used kernels are $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$ and $\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$. Notice that these kernels are the negative of the correct discretization of the Laplacian, to avoid flipping the image intensities. Similarly to the first derivative filters above, the Laplacian is often used for edge, but also for blob detection. Since it is highly sensitive to noise, it is usually combined with a smoothing step by a Gaussian kernel. The

[24] A slightly better (but still rather crude) approach is the Sobel operator, with kernels $G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$ and its transpose $G_x = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$.

combined Laplacian of Gaussian kernel can be pre-calculated as follows:

$$\mathrm{LoG} = -\frac{1}{\pi\sigma^4}\left[1 - \frac{x^2+y^2}{2\sigma^2}\right] e^{-\frac{x^2+y^2}{2\sigma^2}}$$

To discretize and apply the LoG we make use of the `gaussian_laplace()` function from `scipy.ndimage` (again, notice below the `-1.0` to flip the sign of the Laplacian).

```
from scipy.ndimage import gaussian_laplace

img_f_log = -1.0 * gaussian_laplace(img.astype(np.float32), sigma=2.0)
```

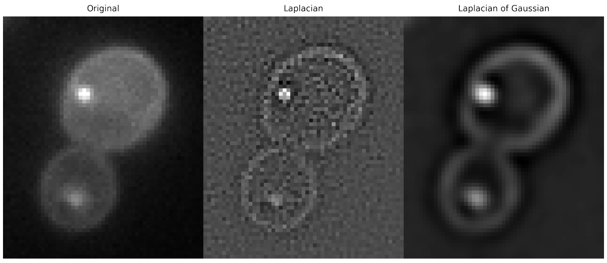


Figure 26: Laplacian and LoG

The Laplacian of Gaussian is more robust to noise that the simple Laplacian and enhances features at the scale `sigma` of the Gaussian kernel.

5.5.4 Linear vs. non-linear filters

Non-linear filters cannot be implemented by the convolution operator since, as their name implies, they do not perform any linear operation (*i.e.*, additions and multiplications) on the pixel neighborhood. For specific applications, however, non-linear filters may display higher performance than linear filters. To investigate this, we will use two versions of a reasonably high-SNR image that we perturb with either **salt & pepper** noise (to simulate a camera with *dead* or *hot pixels*) or **Gaussian** noise (to simulate a camera with important dark or read-out noise). We use the **peak SNR** value (in dB) as a single number to represent the difference in quality between the original image and either the noisy or filtered version.

The pSNR is defined as:

$$\mathrm{pSNR} = 20 \cdot \log_{10} \mathrm{MAX}_I - 10 \cdot \log_{10} \mathrm{MSE}$$

where:

$$\mathrm{MSE} = \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\left[I(i, j - K(i,j)\right]^2$$

and MAX_I is the maximum possible value of the data range of image I (255 for 8-bit and 65535 for 16-bit images).

We use the pSNR implementation from the `peak_signal_noise_ratio()` function defined in `skimage.metrics`.

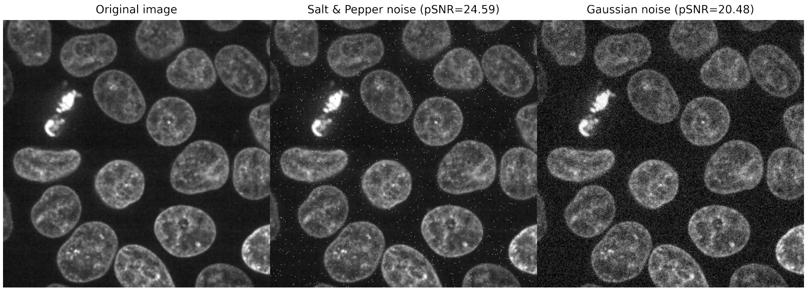


Figure 27: Different types of noise

The salt & pepper noise image has a higher pSNR because even though the noisy pixels are very different in intensity from the original image, they are much fewer than in the Gaussian noise image.

Salt & pepper noise is best suppressed using a median filter. The median filter replaces the intensity at the center of the neighborhood with the median of their intensities. This approach is perfect for removing individual outliers (such as hot pixels). Indeed, the pSNR of the median-filtered version of the salt & pepper noise image is 41.40 against 33.58 for the Gaussian filter. The Gaussian filter spreads the intensities of the noisy pixels to the neighboring pixels, reducing but not suppressing the noise (and reducing the signal strength).

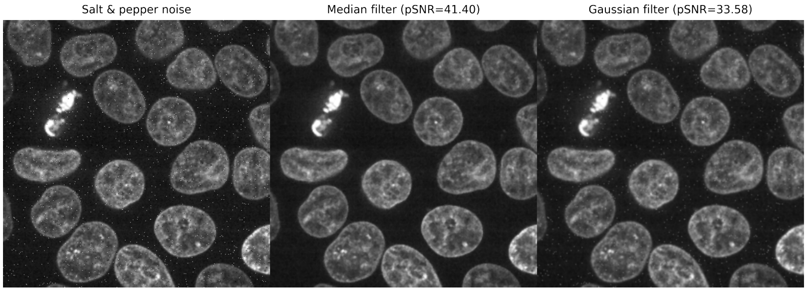


Figure 28: Salt & pepper noise filtered by median and Gaussian filter

If the starting image is perturbed by Gaussian noise, the Gaussian filter is the preferred approach. The median filter is inferior in quality (27.45 against 30.46) and has the additional inconvenience of being much more computationally expensive. For small 2D images, this difference may be negligible, but for large 3D datasets, the computation time may become prohibitive. Faster approximations of the median filter have been developed, but the performance of the Gaussian filter is still vastly superior.

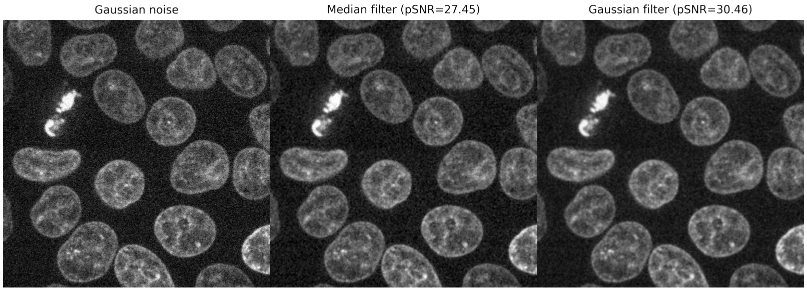


Figure 29: Gaussian noise filtered by median and Gaussian filter

5.6 Segmentation

When performing quantitative image analysis, we want to extract objects from images to measure several features we can use to test some hypotheses. It is then crucial to preserve their shapes as faithfully as possible. For a robust segmentation, the objects of interest would ideally have an intensity significantly higher than their local background. In the example image below, the cell body and the neuron axons have a stronger intensity than the image background. However, the axons are weaker than the cell body.

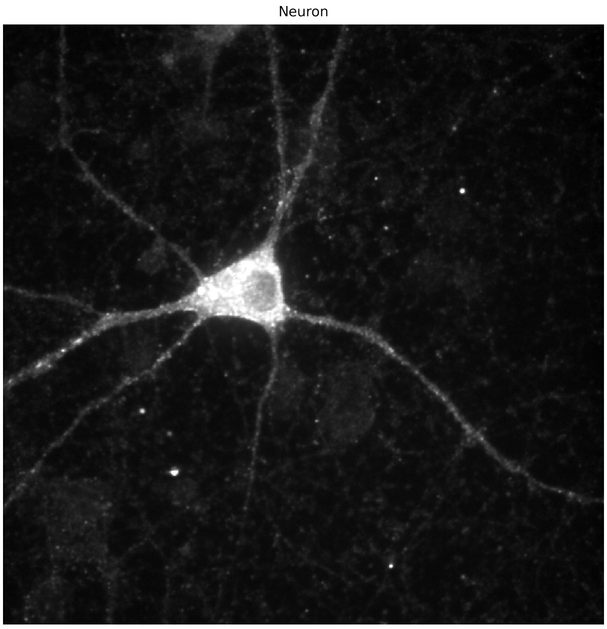


Figure 30: The neuron.tif image

Different segmentation algorithms may give different segmentations of the same signal. As an example, the plot below shows the histogram of the image intensities in red, and the threshold value estimated by the **Otsu** (`threshold_otsu()`), **Li** (`threshold_li()`), and **Triangle** (`threshold_triangle()`) algorithms.

The algorithms are implemented in the `skimage.filters` package, and can be used as follows:

```
from skimage.filters import threshold_otsu, threshold_li, threshold_triangle

th_ot = threshold_otsu(img)
th_li = threshold_li(img)
th_tr = threshold_triangle(img)
```

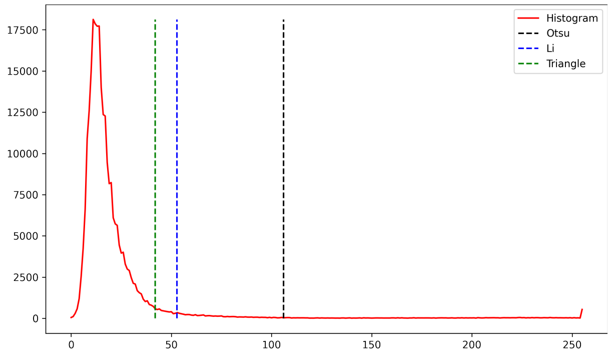


Figure 31: Threshold values obtained by different algorithms

The black and white segmentation masks can then be obtained by simple relational expressions:

```
bw_ot = img > th_ot
bw_li = img > th_li
bw_tr = img > th_tr
```

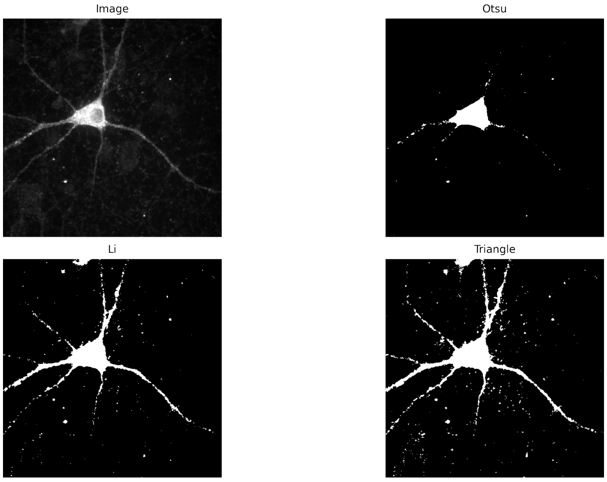


Figure 32: Segmentation mask obtained by the threshold values estimated above

If the intensity of the objects tends to vary a lot, the Otsu algorithm might perform sub-optimally. Compare the Otsu segmentation with the one obtained by the Li or the Triangle algorithms. These algorithms are particularly robust to segmenting objects with a wide range of intensities.

5.7 Background subtraction

Sometimes, segmentation based on simple thresholding fails spectacularly. For example, in the image below, the Otsu algorithm is applied to segment the individual cells that cover the whole field of view. Unfortunately, the Otsu-based segmentation fuses the whole center of the image into one gigantic blob.

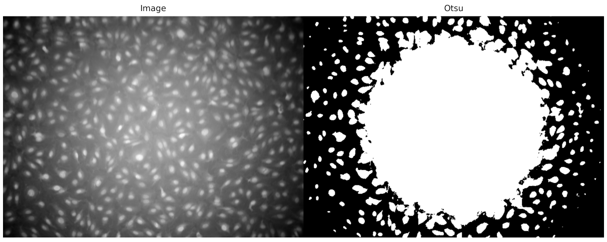


Figure 33: Segmentation with non-homogeneous background

Often, cameras that cover large fields of view in microscopy acquisitions display a more intense average signal in the center of the view that slightly fades toward the borders. This *shading* effect can make object segmentation more difficult because no single threshold value will be optimal throughout the image.

To investigate the reason for the failure of the Otsu algorithm on this image, we plot an intensity profile across the image. The plot shows that even though single cells are recognizable as sharp peaks across the profile, they lie on a highly curved background. The Otsu threshold cuts the hill across the middle, resulting in the whole image's central region being classified as foreground. Only a few cells at the periphery of the image are bright enough to be at least partially above the Otsu threshold.

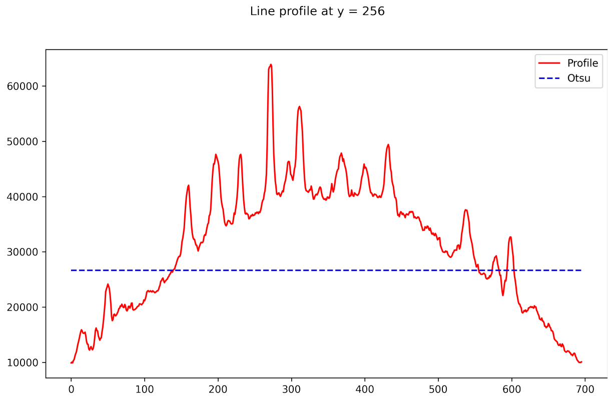


Figure 34: Intensity profile across the image

Our quick investigation shows a strong shading effect that significantly distorts the background level of the image. This effect can be corrected (or at least reduced) in several ways. The best method is to acquire a second image at the microscope containing only the background pixel intensities (that is, after removing the sample) and subtract this dark image from the image to be corrected. If this background image is not available, an approximation of the background can be estimated from the image itself. For this to work reliably, the signal must be ideally concentrated in small blobs surrounded by large background areas.

There are several algorithm that can be used to estimate the background of an image. The `iaf` library implements three distinct algorithms in the `subtract_background()` function from the `iaf.process` package: `"rolling ball"`, `"morphological_opening"`, and `"gaussian"`[25]. While the morphological opening algorithm may be the most accurate, it is computationally expensive, and the rolling ball algorithm is the default for the function. To speed up computation at the expense of some potential reduction in the accuracy, the `subtract_background()` function provides an additional parameter `down_size_factor` that will scale down the image for processing. The value of `radius` is scaled accordingly. The end result of will be returned at the original image size.

We can run background subtraction as follows.

```
img_corr, bkg = subtract_background(
    img, algorithm="morphological_opening",
    radius=25, return_background=True)
```

The `radius` parameter should be larger than the size of the objects in the image so that most pixels in the sampled neighborhood belong to the background. At the same time, the radius cannot be too large, or it will fail to estimate the *local* background.

We can use the `plot_background_subtraction_control()` function from the `iaf.plot.validation` package to check the result.

```
plot_background_subtraction_control(img, bkg)
```

[25] More information about the actual algorithms can be found in the `iaf` documentation: https://ia-res.ethz.ch/docs/iaf/process/index.html#iaf.process.subtract_background.

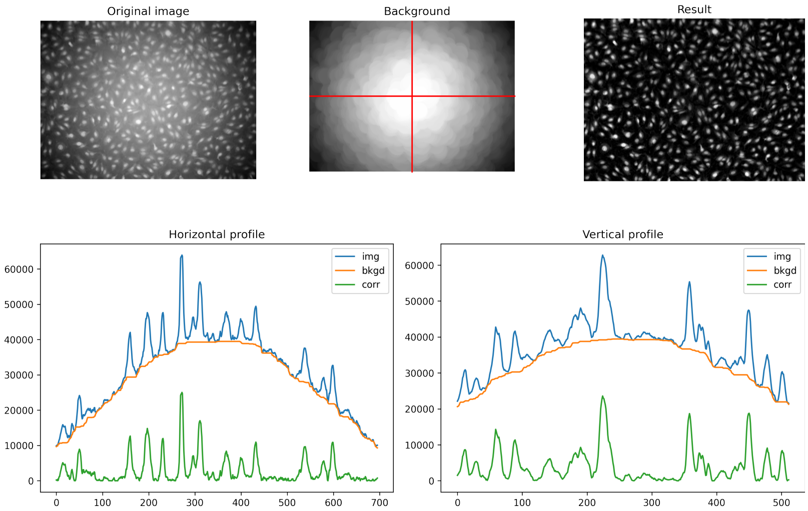


Figure 35: Background subtraction approach and results control

The function shows the original image, the estimated background, and the corrected image. One horizontal and one vertical profile across the center are plotted for all images. The blue `img` line shows the original image intensity, the orange `bkgd` line is the local background estimation, and the green `corr` line is the background-corrected intensity. The baseline of the corrected background should be as close to 0 as possible for a good result. If it is not, lower or higher values of `radius` should be tested. After background correction, the Otsu segmentation of the image works much better.

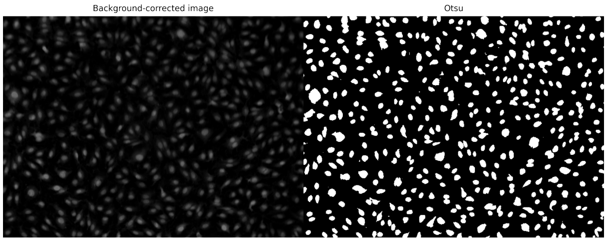


Figure 36: Segmentation after background subtraction

Please notice that background subtraction is not only crucial for correcting images with strong background shading. Another important application is the **ratiometric analysis** of two fluorescence channels. The ratio of the fluorescence channels could be used as a proxy for the relative concentration of two labeled proteins in the cell. However, the ratio of the signal intensities could be strongly affected if the image contains a significant background level. If the background signal is a significant fraction of the local intensity, it may dominate the calculation, and the ratio of the signals may become the ratio of the backgrounds!

5.8 Connected components

The result of segmentation is usually a binary image with all pixels belonging to the background having value 0 and those belonging to the foreground having value 1. Human visual perception can easily recognize distinct objects in a binary image. However, the same binary image for the computer is just a large matrix of numbers; some entries are 0, while others are 1.

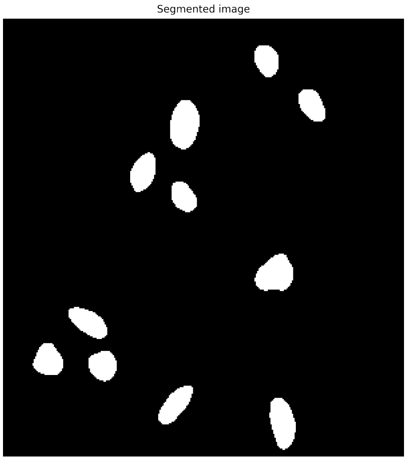

Segmented image

Figure 37: Segmentation mask

For the computer to make head or tail of it, we need to transform the binary mask into separate sets of foreground pixels to which we assign individual **labels**. These labels are unique integers starting from 1; the background pixels are, by convention, assigned to the set with label 0.

The algorithm that implements this operation is called **connected component analysis** (or connected component **labeling**). The algorithm scans the binary image, groups all physically connected foreground pixels, and assigns them to the same connected component. Each connected component is separated from the others by intervening background pixels. In the resulting label image, all pixels inside each connected component are set to the same integer value.

Connected components now map to individual objects that are spatially separate from each other and for which we can extract features for further analysis.

We can use the `label()` function from the `skimage.measure` package to perform connected component analysis (later we will see how to extract measurements):

```
from skimage.measure import label

labels, num = label(bw, background=0, return_num=True, connectivity=1)
```

To visualize the labels we can use the `show_labels()` function from `iaf.plot`.

```
from iaf.plot import show_labels

show_labels(labels, plot_labels=True, title="Connected components")
```

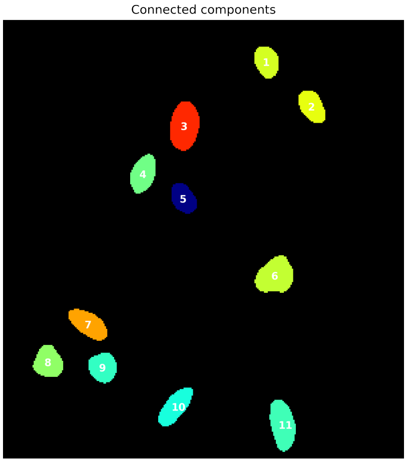

Connected components

Figure 38: Segmentation mask

The `show_label()` function assigns different colors to the different objects. The `plot_labels=True` argument instructs `show_labels()` to display the component labels on top of each object. In addition, it can plot the center-of-mass location (see example below).

5.9 Watershed segmentation

Sometimes, even if the segmentation works mostly fine, we still find that some of the objects are fused.

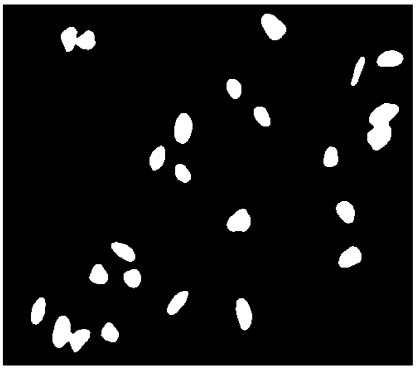


Figure 39: Segmentation mask with fused objects

```
labels, num = label(bw, background=0, return_num=True, connectivity=1)
print(f"Found {num} objects.")
```

```
Found 22 objects.
```

In the result of the connected component analysis, we see that some objects share the same label and therefore the same color, as shown in the figure below (yellow, red and blue objects). They also share one center of mass that is clearly between the two segments.

```
show_labels(labels, plot_centroids=True)
```

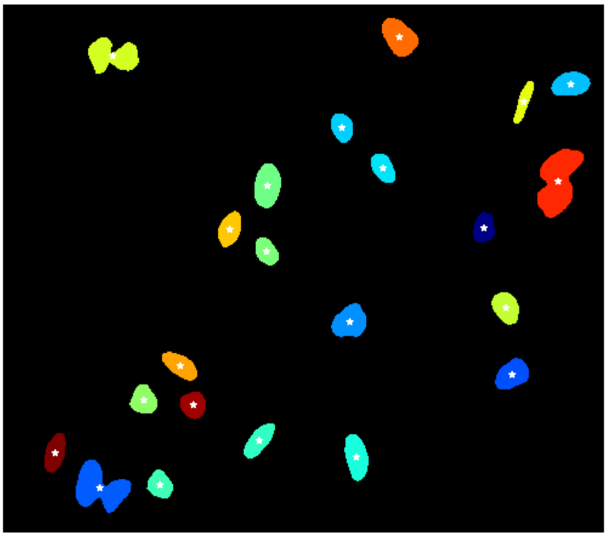


Figure 40: Connected-component labelling of mask with fused objects

The watershed transform can help us separate the fused objects. In practice, we can use the `separate_neighboring_objects()` from the `iaf.morph.watershed` package[26].

[26] The `separate_neighboring_objects` method is extracted, simplified and adapted from CellProfiler's `IdentifyPrimaryObjects` module (https://github.com/CellProfiler/CellProfiler/blob/master/cellprofiler/modules/identifyprimaryobjects.py).

The estimate_object_sizes() function from iaf.morph.watershed can help us set some of the arguments of separate_neighboring_objects[27]:

```
area, min_axis, max_axis, equiv_diam = estimate_object_sizes(labels)

labels_sep, num_sep, _ = separate_neighboring_objects(bw, labels, min_size=min_axis)
print(f"Found {num_sep} objects.")
```

```
Found 25 objects.
```

We can check that the fused objects have now all been separated successfully.

```
show_labels(labels_sep, plot_centroids=True)
```

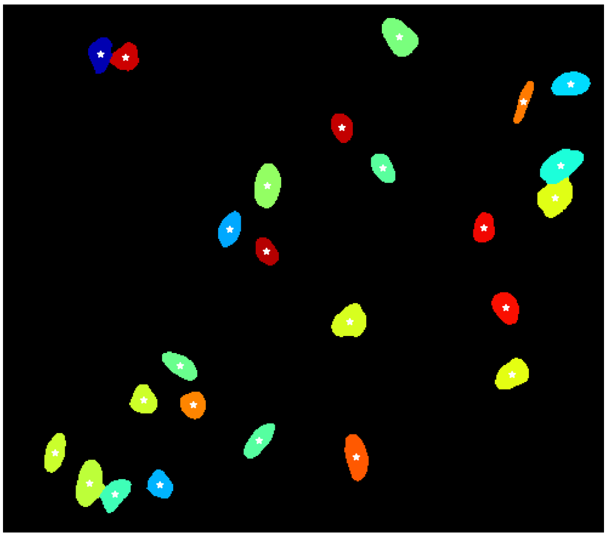


Figure 41: Connected-component labelling of mask after watershed segmentation

The separate_neighboring_objects function takes many arguments (please consult the documentation), but should have reasonable defaults. The min_size and maxima_suppression_size are probably the most sensitive ones.

5.10 Morphological operations

Another problem of bad segmentation is the fragmentation of objects into smaller components. In the example below, a large fraction of the cytoplasm of the cell has very low signal, and a naïve thresholding assigns most of it to the image background.

[27] As a starting point, the min_axis value returned by estimate_object_sizes() is a reasonable value for the min_size argument of separate_neighboring_objects. Since min_axis is the median of all min axes, though, it often helps to reduce it a bit, *e.g.*, ...min_size = 0.8 * min_axis).

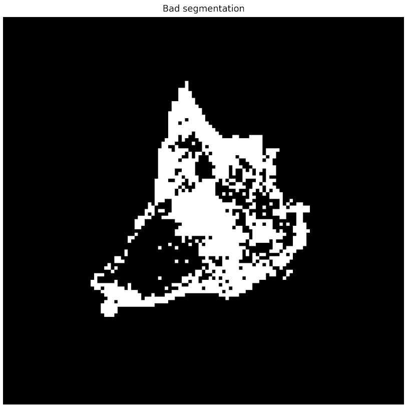


Figure 42: A bad segmentation result

The subsequent connected component analysis will then create a large number of separate objects and their respective labels.

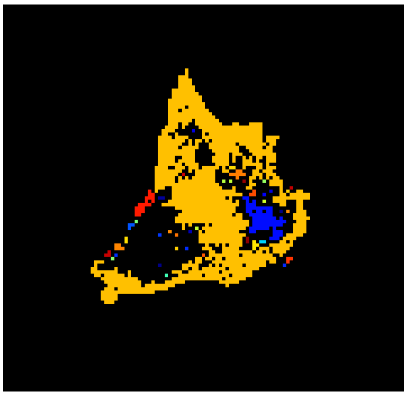


Figure 43: Connected-component labelling of the bad segmentation result

Morphological operations may help fixing this kind of problems[28]. If we run a round of dilation() (with a `disk` of radius 1 as structuring element), we can increase the size of small fragments enough to have them merge with their immediate neighbors.

```
from skimage.morphology import dilation, disk

dilated = dilation(bw, disk(1))
```

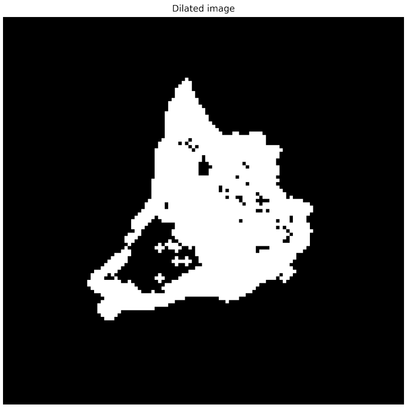


Figure 44: Dilation step

[28] Care however must be taken not to incur into the previous problem, where separate objects get erroneously fused.

Then, we can run the morphological binary_fill_holes() operation (from scipy.ndimage) to turn all background pixels that are completely surrounded by foreground to foreground pixels:

```
from scipy.ndimage import binary_fill_holes

filled = binary_fill_holes(dilated)
```

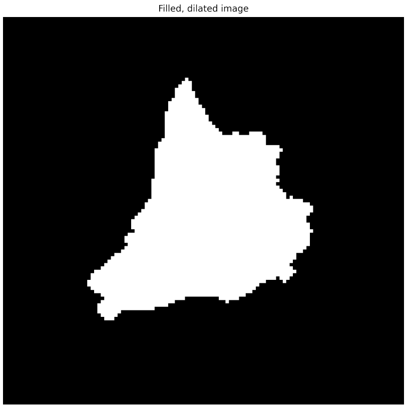


Figure 45: Fill-holes step

Finally, we can reverse the *fattening* effect of the original dilation with an erosion() step with the same structuring element.

```
eroded = erosion(filled, disk(1))
```

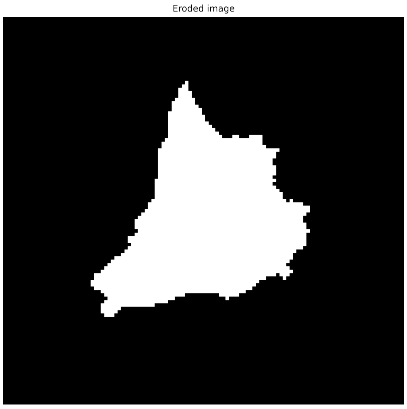


Figure 46: Erosion step

The erosion cannot carve holes into a homogeneous foreground field and we now have the complete object that the initial thresholding algorithm failed to segment.

5.11 Measurements

The last step in our analysis workflow is the extraction of quantitative measurements from the segmented and post-processed objects.

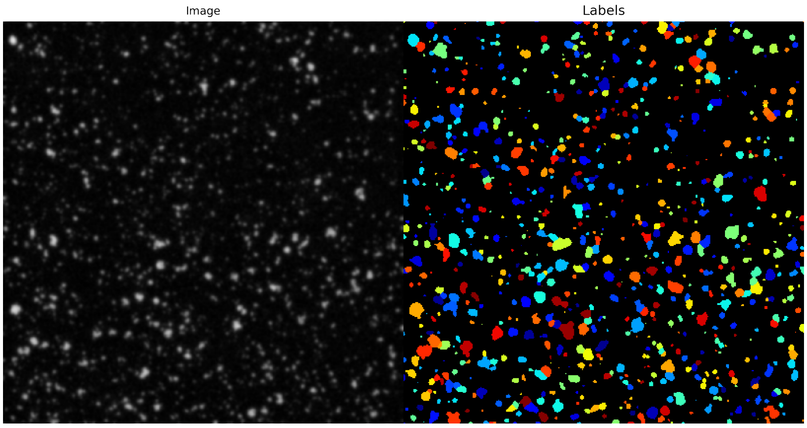


Figure 47: Image and corresponding connected-component labelling

For this, we can use the regionprops() function (or its sibling regionprops_table()) from skimage.measure. In the figure above and in the following, img is an intensity (gray-value) image and labels is the result of connected component analysis.

```
from skimage.measure import regionprops, regionprops_table

props = regionprops(labels, intensity_image=img)
```

The results of regionprops() is a long list of measurements for each of the labels, such as area, centroid, eccentricity, intensity_mean, perimeter, solidity, … (for the complete list of features, please see the documentation of regionprops()). Intensity-based measurements (such as intensity_mean) are only performed if a second argument intensity_image is passed to regionprops; if no intensity image is passed, only measurements that can be extracted from the labels (such as area and perimeter) will be returned.

We can iterate over all meaurements as follows:

```
for prop in props:
    print(f"label={prop.label:2}: area={prop.area:3}")
```

```
label= 1: area=  3
label= 2: area=  2
label= 3: area= 37
label= 4: area= 14
label= 5: area=  4
...
```

We can work with whole sets of measurements as follows:

```
n, b = histogram(props["intensity_mean"])

plt.bar(b, n)
plt.suptitle("Intensity mean");
```

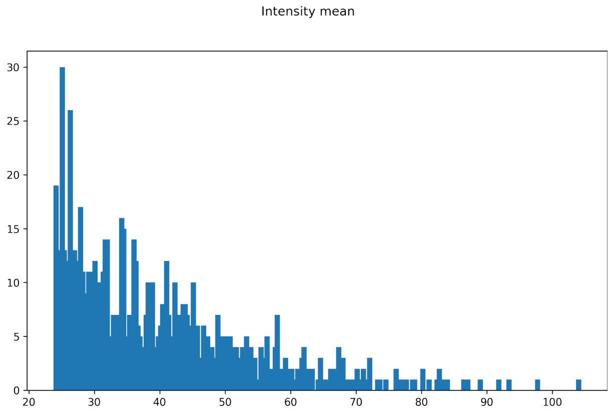


Figure 48: Histogram of mean intensities

Additional measurements can be added to `regionprops()` by specifying functions that are passed to the `extra_properties` argument. Those functions will take a region mask as its first argument, and an optional intensity image as the second argument.

```
def intensity_median(regionmask, intensity_image):
    return np.median(intensity_image[regionmask])

props = regionprops(labels, intensity_image=img, extra_properties=(intensity_median,))

props[0].intensity_median
```

```
35.0
```

6 Image processing and analysis examples

In the following, we will combine the notions we have discussed so far and study a series of image analysis examples of varying complexity.

6.1 Example 1: basic image import, processing, and export

This example shows how to read an image, adjust the contrast in the image, and then write the adjusted image to a file. You can follow along on the Jupyter notebook `code/python/notebooks/simple_processing.ipynb`. To run it, start the **Anaconda Prompt** in Windows or your terminal in macOS or Linux, and type the following:

```
$ cd code/python/notebooks/
$ jupyter notebook
```

(Alternatively, you can run `jupyter lab`.)

In the browser, select the `simple_processing.ipynb` notebook to start.

> **Note**
>
> While contrast stretching is implemented in all image analysis and processing packages, its main value lies in allowing us to see details that our eyes would otherwise miss. Be very careful in **replacing the original values with the modified ones**, since this may **compromise** your analysis!

6.1.1 Import all modules and functions

It is customary to import all needed modules and functions at the beginning of a module or a notebook. These are the ones we will need in this example:

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

```
from skimage.io import imread, imsave
from skimage.exposure import histogram, rescale_intensity
```

6.1.2 Read and display an image

We read an example image using the imread function from scikit-image. The chosen image of a young girl is in a file named pout.tif that is in current folder, and can therefore be opened without specifying the full path. The opened image is stored in a NumPy array named I .

```
I = imread('pout.tif')
```

We display the image, using the imshow() function from Matplotlib (we specify the arguments vmin=0 and vmax=255 to prevent imshow() from stretching the intensities for us)[29]:

```
plt.imshow(I, cmap="gray", vmin=0, vmax=255);
```

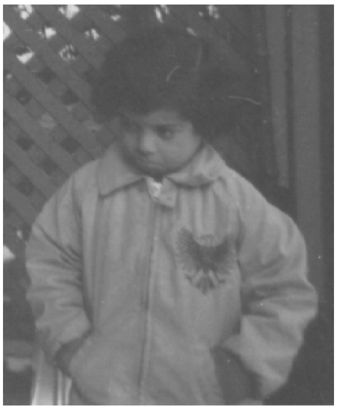


Figure 49: The pout.tif image

6.1.3 Inspect the image

We can query some information about the image (or, more precisely, the NumPy array that contains it) as follow:

```
print(f"Image size = {I.shape}, data type = {I.dtype}, size in memory = {I.size} bytes.")
print(f"Type of the Image variable = {type(I)}")
```

This gives us the following output:

```
Image size = (291, 240), data type = uint8, size in memory = 69840 bytes.
Type of the Image variable = <class 'numpy.ndarray'>
```

In IPython, we can also use the magic command %whos (with an optional filter on the type of variables), to get a summary information in the console:

```
%whos ndarray
```

```
Variable   Type       Data/Info
-------------------------------
I          ndarray    291x240: 69840 elems, type `uint8`, 69840 bytes
```

In both cases, we see that the imread() function returns the image data in the NumPy array I , which is a 291-by-240 2D array of uint8 data (and indeed $291 \cdot 240 = 69840$, the size of the array in bytes).

6.1.4 Improve image contrast

Let's look at the distribution of image pixel intensities. The image pout.tif is a somewhat low contrast image: it's not too dark, and it's not too bright, but it is somewhat dull. To see the distribution of intensities

[29] We can achieve the same results with the imshow() function from iaf.plot without passing any explicit arguments: imshow(I).

in the image, we create a histogram by calling the `histogram()` function from `skimage.exposure`. Notice how the histogram indicates that the intensity range of the image is rather narrow. The range does not cover the potential range of [0, 255], and is missing the high and low values that would (potentially) result in good contrast.

Note

> Stretching the contrast of images is acceptable for visualization purposes, and addresses the poor gray-value resolution of the human eye, but altering the original intensities can introduce biases in quantitative analysis!

To return the histogram over the full range of the `uint8` data type, we pass the argument `source_rage="dtype"` to the `histogram()` function.

```
hist = histogram(I, source_range="dtype")
```

The `hist` variable is a `tuple`, that contains two 1D arrays. The pixel frequencies (in `hist[0]`), and the bin center values in (`hist[1]`). We can plot the histogram as follows:

```
plt.bar(x=hist[1], height=hist[0])
```

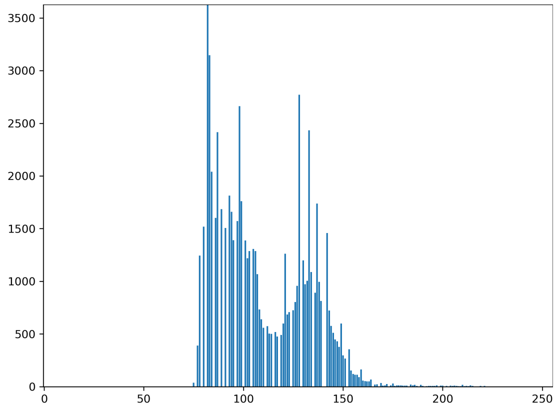


Figure 50: Histogram of pout.tif

We can improve the contrast of an `uint8` image using the `rescale_intensity()` function from `scikit-image` to stretch its intensities to cover the full `[0, 255]` dynamic range. By default, `rescale_intensity()` stretches the intensities between the minimum and maximum intensity values of the image.

```
I_out = rescale_intensity(I)
```

How does the stretched image `I_out` look like?

```
plt.imshow(I_out, cmap="gray", vmin=0, vmax=255);
```

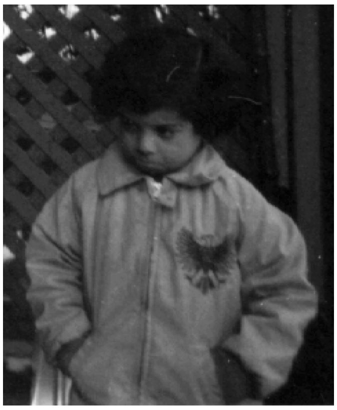


Figure 51: The stretched pout.tif image

The image became darker, but the contrast didn't change much. How is that? Let's look at the updated histogram.

```
hist_out = histogram(I_out, source_range="dtype")
plt.bar(hist_out[1], hist_out[0])
```

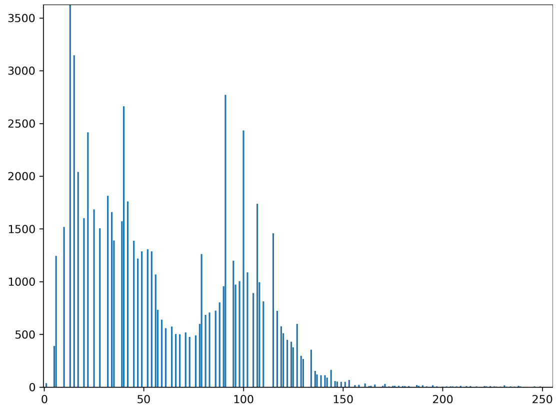


Figure 52: Histogram of pout.tif

Even though the left part of the histogram has shifted towards 0 (*i.e.*, dark pixels have become even darker), there does not seem to be a significant stretch of the histogram: instead of extending to cover the whole range, the right part of the histogram just shifted to the left, leaving a big gap in the intensities higher than about 150.

We said that, by default, `rescale_intensity()` stretches the intensities between the minimum and maximum intensity values of the image. Let's have a look at what those values are in the original image. Since `I` is a `NumPy` array, we can call the associated `min()` and `max()` methods on the array itself.

```
mn = I.min()
mx = I.max()
print(f"Minimum image intensity is {mn}; maximum image intensity is {mx}.")
```

```
Minimum image intensity is 74; maximum image intensity is 224.
```

Now we see that while the minimum could be stretched all the way from 74 down to 0, a very small number of bright pixels covered almost all the range up to 255 (precisely, 224). Those few pixels prevented the histogram from being stretched more significantly, and the little stretch we obtained was mostly in the lower end of the spectrum.

To prevent this kind of surprises, it is common to stretch the histogram of an image by considering a few percent of its lowest and highest intensities as potential **outliers**. We can consider the central 95% of all (sorted) pixel intensities as really *belonging* to the image, and discard the lowest and highest 2.5% as outliers. What pixel intensities in the image correspond to the 2.5^{th} and 97.5^{th} **percentile**? We can use the `percentile()` function from `NumPy` to find out:

```
p_low, p_high = np.percentile(I, (2.5, 97.5))
print(f"2.5th percentile is {p_low}; 97.5th percentile is {p_high}.")
```

```
2.5th percentile is 80.0; 97.5th percentile is 153.0.
```

We can now use these values as new range limits in `rescale_intensity()`:

```
I_out_pc = rescale_intensity(I, in_range=(p_low, p_high))
```

Let's have a look at the new image:

```
plt.imshow(I_out_pc, cmap="gray", vmin=0, vmax=255);
```

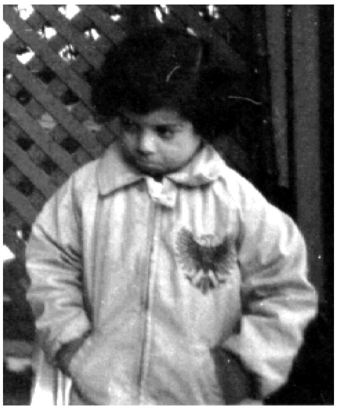


Figure 53: Histogram-equalized image

and at its histogram:

```
hist_out_pc = histogram(I_out_pc, source_range="dtype")
plt.bar(hist_out_pc[1], hist_out_pc[0])
```

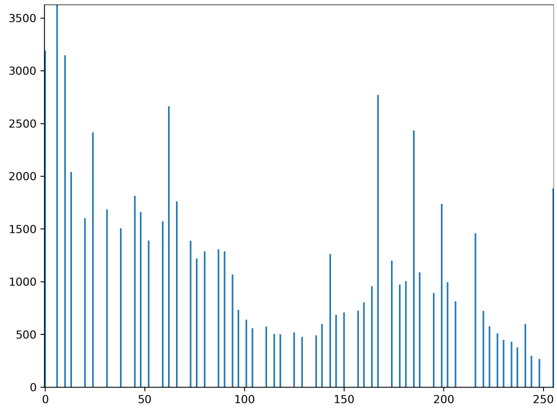


Figure 54: Histogram after equalization

The histogram is now nicely spread over the whole `uint8` range.

6.1.5 Write the adjusted image to a file

To conclude, we write the newly adjusted image `I_out_pc` to a **.tif** file, using the `imsave` function from `skimage.io`.

```
imsave("pout_stretched.tif", I_out_pc)
```

6.2 Example 2: correct nonuniform background illumination and analyze foreground objects

This example shows how to enhance an image as a preprocessing step before analysis. In this example, we correct the nonuniform background illumination and convert the image into a binary image so that we can perform analysis of the image foreground objects. You can follow along on the Jupyter notebook `code/python/notebooks/analysis.ipynb`.

6.2.1 Import all modules and functions

Again, we start by importing all modules and functions that we will use in this example.

```
import matplotlib.pyplot as plt
import numpy as np
import scipy
from skimage.exposure import rescale_intensity, histogram
from skimage.filters import threshold_otsu
from skimage.io import imread
from skimage.measure import label, regionprops
from skimage.morphology import area_opening, opening, disk, reconstruction
from iaf.morph.watershed import separate_neighboring_objects
from iaf.plot import get_labels_cmap
```

6.2.2 Read the Image into the Workspace

We read and display the gray-scale image `rice.png`.

```
I = imread("rice.png")
plt.imshow(I, cmap="gray", vmin=0, vmax=255)
```

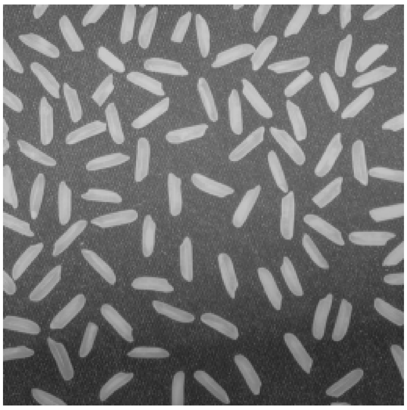


Figure 55: The rice.png image

6.2.3 Preprocess the image to enable analysis

In the sample image, the background illumination is brighter in the center and top of the image than at the bottom. As a preprocessing step before analysis, we will make the background uniform and then convert the image into a binary image. To make the background illumination more uniform, we create an approximation of the background as a separate image and then subtract this approximation from the original image[30].

As a first step to creating the background approximation image, we will remove all the foreground (rice grains) using **morphological opening** (the morphological opening operation is an erosion followed by a dilation). The opening operation has the effect of removing objects that cannot completely contain the structuring element. To remove the rice grains from the image, the structuring element must be sized so that it cannot fit entirely inside a single grain of rice. We use the `disk` function from `skimage.morphology` to create a disk-shaped structuring element with a radius of 15.

```
selem = disk(15)
```

And we run the morphological opening using the `opening` function from `skimage.morphology`.

```
background = opening(I, selem)  # Very slow for large images! See below.
```

Let's have a look at the extracted background.

```
plt.imshow(background, cmap="gray", vmin=0, vmax=255)
```

[30] We could also use the `subtract_background()` function from the `iaf.process` package, but here we show the morphological opening algorithm implemented in `subtract_background()`, albeit in slightly simplified form.

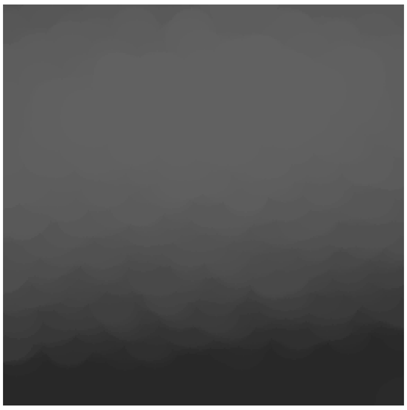


Figure 56: The estimated image background

In the generated background image we can see how and where the illumination varies.

We can then subtract the background approximation image, `background`, from the original image, `I`, and view the resulting image. The resulting image has a uniform background but is now a bit dark for analysis.

```
I2 = I - background
plt.imshow(I2, cmap="gray", vmin=0, vmax=255)
```

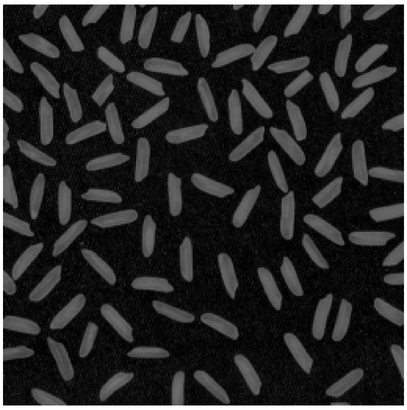


Figure 57: Background-subtracted rice image

Please note that for large images, morphological opening can be quite computationally intensive: iaf.process.subtract-backround implements a series of algorithms and optimizations to speed up the background subtraction.

Using `rescale_intensity`, we can increase the contrast of the processed image `I2` by saturating 1% of the data at both low and high intensities and by stretching the intensity values to fill the `uint8` dynamic range[31].

```
p_low, p_high = np.percentile(I2, (1, 99))
I3 = rescale_intensity(I2, in_range=(p_low, p_high))
```

Let's look at the resulting image:

```
plt.imshow(I3, cmap="gray", vmin=0, vmax=255)
```

[31] This step is not strictly necessary for the analysis, and we would drop it altogether if we wanted to extract intensity information from our segmented objects.

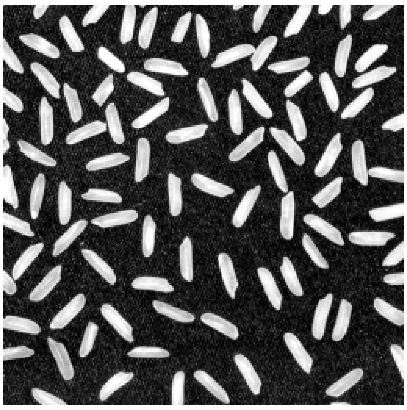


Figure 58: Contrast-enhanced rice image

To create a binary version of the processed image, that we can use for analysis, we use the **Otsu** algorithm implementation from `skimage.filters`.

```
threshold = threshold_otsu(I3)
print(f"Threshold = {threshold}")
```

```
Threshold = 122
```

The extracted threshold value can be used to binarize the image as follows:

```
bw = I3 > threshold
```

Let's take a look at the black-and-white binary image. We see that noise in the image resulted in a few small objects that survived the tresholding step. Also, we want to remove objects that are partially visible in the image because they are cut by the image borders.

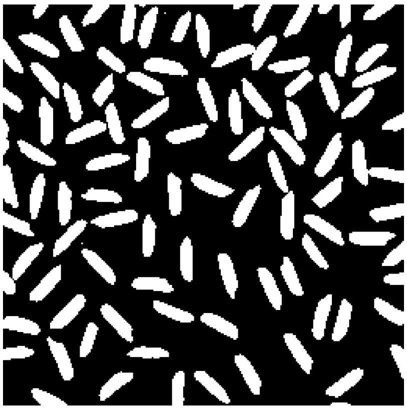


Figure 59: Binary image with spurious small objects due to noise

We can use **area opening** (a morphological operation with a non-fixed structuring elements of given area) to remove these small objects. We set a minimum area of 50 pixels^2 for objects to be preserved[32].

```
bw = area_opening(bw, area_threshold=50)
```

Finally, we plot the clean binary image `bw` that we will use for analysis.

```
plt.imshow(bw, cmap="gray")
```

[32] For a quicker alternative, see iaf.morph.watershed.filter_labels_by_area.

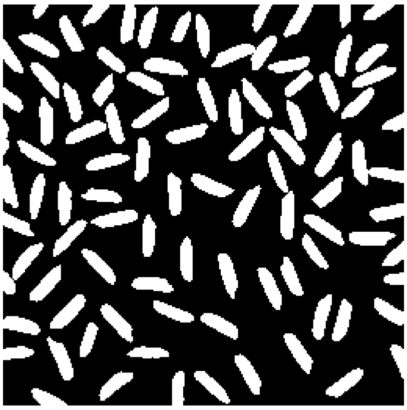


Figure 60: Final binary image ready for analysis

6.2.4 Perform analysis of objects in the image

Now that we have created a binary version of the original image, we can perform some analysis of the objects we see in it.

We will start with finding all the connected components (objects) in the binary image. The accuracy of the results depends on the size of the objects, the connectivity parameter (4, 8, or arbitrary), and whether or not any of the rice grains are touching in the segmented mask (in which case they could be labeled as one object).

We can use the `label` function from `skimage.measure`.

```
labels, num = label(bw, background=0, return_num=True, connectivity=1)
print(f"Found {num} connected components.")
```

```
Found 95 connected components.
```

There are 95 objects in the images. We color-code the extracted objects for a quick visual check.

```
plt.imshow(labels, cmap=get_labels_cmap(), interpolation="nearest");
```

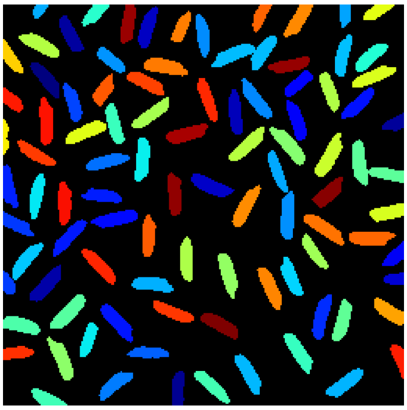


Figure 61: Color-coded objects

Let's have a look at the rice grain labeled 65 in the image. We can use the NumPy function `where()`, that takes a logic comparison as first argument, and then two values to place in the output matrix: the value to put at the position where the logic comparison is satisfied (in our case `True`), and the value to put at the position where the logic comparison fails (in our case `False`).

```
bw_65 = np.where(labels == 65, True, False)
plt.imshow(bw_65, cmap='gray')
```

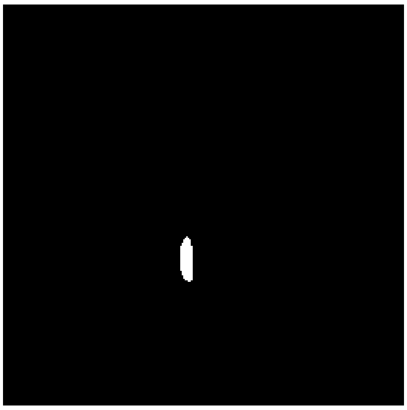


Figure 62: Rice grain number 65

We can compute various properties of each object in the image using `regionprops` from `skimage.measure`.

```
props = regionprops(labels)
```

From the resulting `props` object, we can extract all areas into an `areas` list.

```
areas = []
for prop in props:
    areas.append(prop.area)
```

Let's look at some characteristics of our areas.

```
areas = np.array(areas)
print(f"Median area = {np.median(areas)}; max area = {areas.max()}")
```

```
Median area = 190.0; max area = 404
```

The largest rice grain has twice the area of the median rice grain. This suggests that some of the rice grains were not properly segmented and resulted in the same connected component. Let's see how the largest grain looks like:

```
# We add 1 to the index since `areas` does not contain the background.
index = np.argmax(areas) + 1
print(f"The largest object has area {areas[index - 1]} and index {index}.")
```

```
The largest object has area 404 and index 14.
```

Let's extract it and display it.

```
bw_largest = np.where(labels == index, True, False)
plt.imshow(bw_largest, cmap='gray')
```

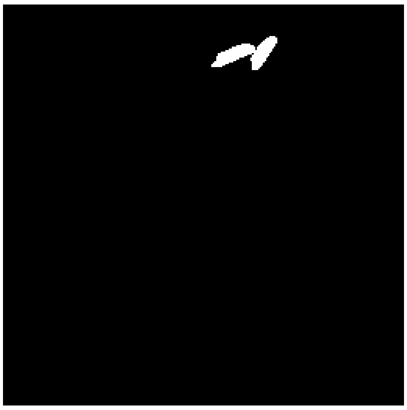


Figure 63: The largest object is two merged rice grains

6.2.5 Perform watershed segmentation

We can use watershed segmentation to separate fused objects. Notice that the raw watershed transform implemented by the watershed function from skimage.segmentation tends to over-segment (that is, *break*) the objects. Since the details of this behavior and the possible steps for compensating are quite convoluted, here we will simply use a more flexible algorithm implemented in iaf.morph.watershed.

```
labels_ws, num_ws, _ = separate_neighboring_objects(bw, labels)
print(f"Found {num_ws} connected components.")
```

```
Found 97 connected components.
```

From the 95 components we had originally, we now have 97. Apparently, we split some fused objects.

Please notice that separate_neighboring_objects() performs advanced watershed segmentation and automatically re-runs the connected component analysis to return an updated label image label_ws. That is, we do not need to re-run skimage.measure.label() on label_ws[33]. Let's plot the new labels:

```
plt.imshow(labels_ws, cmap=get_labels_cmap(), interpolation="nearest");
```

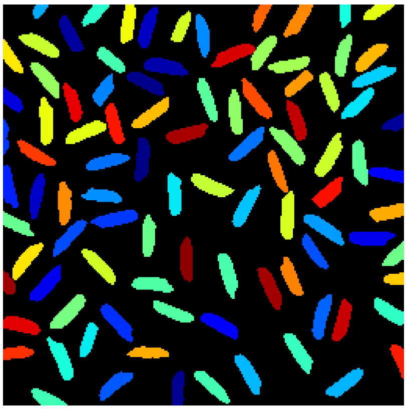


Figure 64: Color-coded objects after watershed

Let's measure the properties of the new objects for comparison.

```
props_ws = regionprops(labels_ws)
```

We compare the areas with those from previous segmentation.

```
areas_ws = []
for prop in props_ws:
    areas_ws.append(prop.area)
areas_ws = np.array(areas_ws)

print(f"Median area = {np.median(areas_ws)}; max area = {areas_ws.max()}")
```

```
Median area = 190.0; max area = 236
```

This is more likely to be correct.

6.3 Example 3: statistical segmentations

This example shows how to use the characteristics of the signal (or rather, in this case, of the background) to segment objects that are otherwise tricky to segment with standard thresholding algorithms. You can find the corresponding statistical_thresholding.ipynb notebook in code/python/notebooks.

The data we have consists of an image N of already segmented (*i.e.*, black and white) nuclei from a fluorescence microscopy acquisition, an image S of the corresponding cells with a (weak and noisy) fluorescent signal that (where present) fills the whole cell.

[33] It may even be harmful, for instance if connectivity = 1.

Our goal is to quantify the fraction of positive cells in the image. The positive cells are those cells that have a cellular signal that is detectable above noise level. We can use the N image to know where all the cells are (because all nuclei are strongly labelled), and the S image to sample the signal at the position of the nuclei and decide whether it is significant or not.

6.3.1 Import all modules and functions

For this example we will need the following modules.

```
import matplotlib.pyplot as plt
import numpy as np
from skimage.filters import threshold_otsu
from skimage.io import imread
```

6.3.2 Read and display the images

We read the images into the N and S arrays.

```
N = imread("stats_nuclei_bw.tif")
S = imread("stats_signal.tif")
```

N is the segmented image of the nuclei, that is, we are given a black-and-white mask. S is the background-corrected cellular signal image.

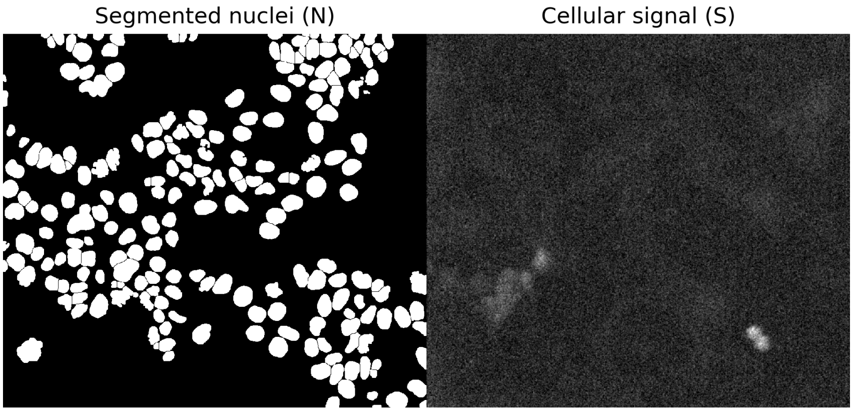


Figure 65: Raw images

6.3.3 Segment with Otsu

How do we know which cells in S have a significant signal? Let's start by naively segmenting S with Otsu.

```
threshold = threshold_otsu(S)
bw_S = S > threshold
plt.imshow(bw_S, cmap="gray");
```

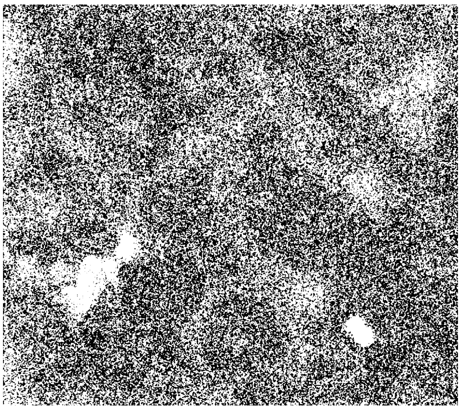


Figure 66: Result of Otsu segmentation

This does not look very good. Let's see if we can find a better approach.

6.3.4 Two words about *sample distributions*

We will start by considering a sample X of 10000 randomly sampled numbers from a Normal distribution $X \sim \mathcal{N}(\mu, \sigma^2)$ with mean $\mu = 100$ and standard deviation $\sigma = 5$, *i.e.*, $X \sim \mathcal{N}(100, 5^2)$.

Numpy's `random.randn()` function returns numbers sampled from a standard normal distribution with mean 0 and standard deviation 1.

```
# We set the seed of the random number generator to get the same result at every run.
np.random.seed(seed=42)
X = 100 + 5 * np.random.randn(10000)
```

Let's check the statistics of the sample X:

```
print(f"X has mean {X.mean():.2f} and standard deviation {X.std():.2f}; " +
      " it should be close to (100.00, 5.00)")
```

```
X has mean 99.99 and standard deviation 5.02; it should be close to (100.00, 5.00)
```

Let's have a look at the distribution of these 10000 numbers.

```
fig, ax = plt.subplots()
ax.hist(X, bins="auto");
ax.plot([X.mean(), X.mean()], [0, 550], 'r-') # Plot the mean
ax.set_xlabel("Possible values of X")
ax.set_ylabel("Counts");
```

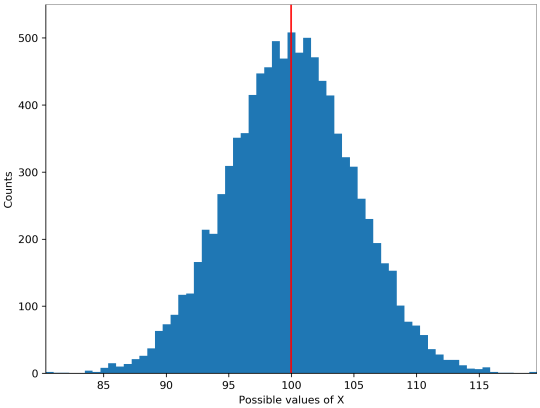


Figure 67: Sample distribution with mean

In a (perfect) normal distribution, approximately 68% of all elements of the samples lie within 1 standard deviation above and below the mean; 95% are within 2 standard deviations, and 99.7% are within 3 standard deviations. Let's calculate those boundaries and plot them.

```
# Calculate
m = X.mean()
s = X.std()
one_std_high = m + s
one_std_low = m - s
two_std_high = m + 2 * s
two_std_low = m - 2 * s
three_std_high = m + 3 * s
three_std_low = m - 3 * s

# Plot
fig, ax = plt.subplots()
ax.hist(X, bins="auto");
ax.plot([m, m], [0, 550], 'r-') # Plot the mean
ax.plot([one_std_high, one_std_high], [0, 550], 'y-') # Plot one_std_high
ax.plot([one_std_low, one_std_low], [0, 550], 'y-') # Plot one_std_low
ax.plot([two_std_high, two_std_high], [0, 550], 'g-') # Plot two_std_high
ax.plot([two_std_low, two_std_low], [0, 550], 'g-') # Plot two_std_low
ax.plot([three_std_high, three_std_high], [0, 550], 'k-') # Plot three_std_high
ax.plot([three_std_low, three_std_low], [0, 550], 'k-') # Plot three_std_low
ax.set_xlabel("Possible values of X")
ax.set_ylabel("Counts");
```

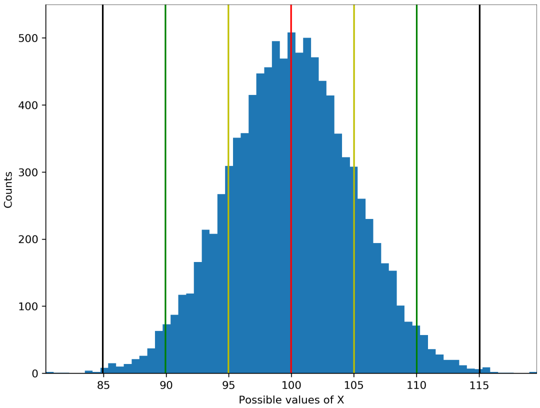


Figure 68: Sample distribution with mean and standard deviations

Let's also count the elements in the partitions and calculate their fractions.

```
n_one_std = X[(X >= one_std_low) & (X <= one_std_high)]
f_one_std = len(n_one_std) / len(X) * 100
n_two_std = X[(X >= two_std_low) & (X <= two_std_high)]
f_two_std = len(n_two_std) / len(X) * 100
n_three_std = X[(X >= three_std_low) & (X <= three_std_high)]
f_three_std = len(n_three_std) / len(X) * 100

print(f"{f_one_std:.2f}% are within 1 std; "
      f"{f_two_std:.2f}% are within 2 std; "
      f"{f_three_std:.2f}% are within 3 std.")
```

```
68.38% are within 1 std; 95.38% are within 2 std; 99.73% are within 3 std.
```

From the plot above, we see that we can find values above ~115 only around 0.3% of the time in our sample. This concept is at the base of **statistical testing**: given a certain sample of elements (that we assume come from a given **distribution** of objects of the same type), if we see another object that is at a certain distance of the mean of the sample, how likely it is to belong to that same type of objects?

In current example, how likely would it be for a value $x = 200$ to belong to the distribution from which we sampled X?

6.3.5 Using sample statistics for thresholding

We can make use of the idea of samples and distributions to try and segment tricky images.

Let's image we have all pixel intensities in the image that belong to the background. If we assume that they belong to the same *class of pixels* (*i.e.*, all background pixels are *generated* by the same process and therefore come from the same distribution), then the intensities from our brighter cells in S should not fall into the distribution of the background intensities.

Let's use the inverse of black and white mask of our nuclei to extract the "non-nucleus" part of the image.

```
plt.imshow(N == 0, cmap='gray');
```

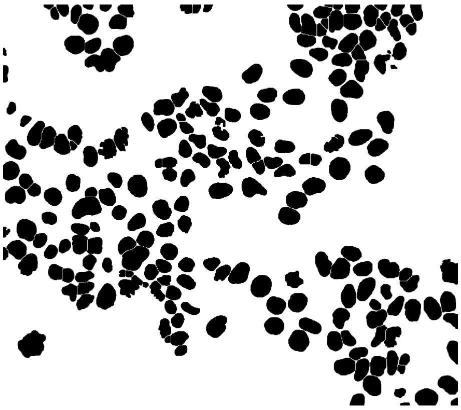


Figure 69: Our background mask

Let's get the intensities from S at the positions where N==0 is True (*i.e.*, for the background pixels).

```
background = S[N == 0]
```

And now, let's calculate our sample statistics.

```
m_bkg = background.mean()
s_bkg = background.std()
print(f"The background has mean = {m_bkg:.2f} and std = {s_bkg:.2f}")
```

```
The background has mean = 11088.81 and std = 4868.70
```

Let's now use the assumption that any intensity higher than 3 standard deviations away from the mean is unlikely to belong to the distribution of background pixels.

```
threshold = m_bkg + 3 * s_bkg
plt.imshow(S > threshold);
```

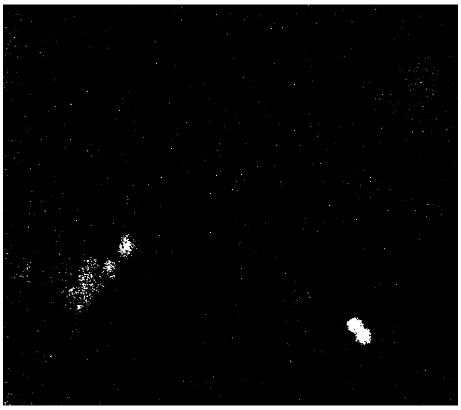


Figure 70: Our cells as outliers of the background population

This looks already more promising than the attempt with the Otsu algorithm above.

Sometimes, the sample we use to estimate the characteristics of a distribution is not good enough. In our example of the background sample, it may contain some autofluorescent pixels, or some cytoplasmic pixels that do not belong neither to the nuclei that we use to define the backround nor to the actual background, since they fall into the cytoplasm.

In this case, we can use **robust statistics** to work around this issue. Imagine having this sample Y:

```
Y = np.array([1, 2, 1, 0, 2, 1, 0, 1, 0, 1, 1, 2, 1, 15])
```

It looks likely that the element 15 does not belong to the rest of the sample. Let's see what is the effect of this **outlier** on our sample statistics.

```
print(f"Sample Y has mean = {Y.mean():.2f} and standard deviation = {Y.std():.2f}")
```

```
Sample Y has mean = 2.00 and standard deviation = 3.66
```

A robust alternative of the mean is given by the **median**, which is the central value in the sorted sample elements; and a robust alternative for the standard deviation is the **median absolute deviation** (**MAD**). The MAD is usually multiplied by a constant factor 1.4826 to bring it in the same range as the standard deviation.

```
md = np.median(Y)
mad = 1.4826 * np.median(np.abs(Y - md))
print(f"Sample Y has median = {md:.2f} and median absolute deviation = {mad:.2f}")
```

```
Sample Y has median = 1.00 and median absolute deviation = 0.74
```

If we use robust statistics on our image, we get the following:

```
median_bkg = np.median(background)
mad_bkg = 1.4826 * np.median(np.abs(background - median_bkg))
print(f"The background has median = {median_bkg:.2f} and MAD = {mad_bkg:.2f}")
```

```
The background has median = 11565.00 and MAD = 5715.42
```

Let's use the robust sample statistics for segmenting our image.

```
threshold = median_bkg + 3 * mad_bkg
plt.imshow(S > threshold);
```

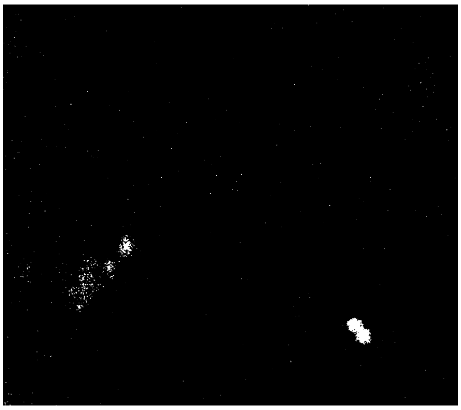


Figure 71: This time using robust statistics

Exercise We will leave it as an hands-on exercise for the course to actually calculate the fraction of positive cells.

6.4 Example 4: blob detection

In this quick demo, we will extract blobs from the Hubble Deep Field image using the **Laplacian of Gaussian (LoG)** and the **Difference of Gaussians** detectors. You can find the corresponding `blob_detection.ipynb` notebook in `code/python/notebooks`.

6.4.1 Import all modules and functions

We start by importing the necessary modules.

```
from math import sqrt
from skimage import data
from skimage.feature import blob_dog, blob_log
from skimage.color import rgb2gray
import matplotlib.pyplot as plt
```

6.4.2 Load the data

Then we load and display (a subset of) the Hubble Deep Field image.

```
image = data.hubble_deep_field()[0:500, 0:500]
image_gray = rgb2gray(image)
plt.imshow(image_gray, cmap='gray')
plt.show()
```

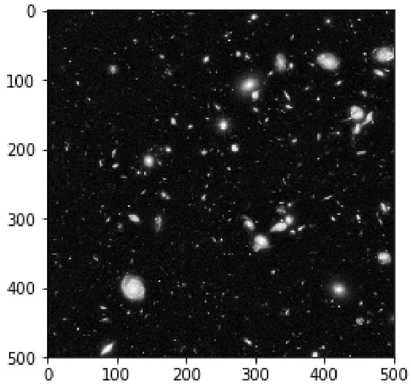


Figure 72: Hubble Deep Field

6.4.3 Run the blob detection

We now initialize the **Laplacian of Gaussian (LoG)** detector from the skimage.feature package. The blob_log function takes several parameters; in particular, the min_sigma, max_sigma and num_sigma are interesting. All three parameters are optional. The function can extract blobs at num_sigma different scales, linearly increasing σ of the Gaussian kernel from min_sigma to max_sigma.

One can obtain help about any function by typing ? followed by the function name.

```
?blob_log
```

```
Signature: blob_log(image, min_sigma=1, max_sigma=50, num_sigma=10, threshold=0.2, overlap=0.5,
                    log_scale=False)
Docstring:
Finds blobs in the given grayscale image.

Blobs are found using the Laplacian of Gaussian (LoG) method [1]_.
For each blob found, the method returns its coordinates and the standard
deviation of the Gaussian kernel that detected the blob.

Parameters
----------
image : ndarray
    Input grayscale image, blobs are assumed to be light on dark
    background (white on black).
min_sigma : float, optional
    The minimum standard deviation for Gaussian Kernel. Keep this low to
    detect smaller blobs.
max_sigma : float, optional
    The maximum standard deviation for Gaussian Kernel. Keep this high to
    detect larger blobs.
num_sigma : int, optional
    The number of intermediate values of standard deviations to consider
    between `min_sigma` and `max_sigma`.
threshold : float, optional.
    The absolute lower bound for scale space maxima. Local maxima smaller
    than thresh are ignored. Reduce this to detect blobs with less
    intensities.
overlap : float, optional
    A value between 0 and 1. If the area of two blobs overlaps by a
    fraction greater than `threshold`, the smaller blob is eliminated.
log_scale : bool, optional
    If set intermediate values of standard deviations are interpolated
    using a logarithmic scale to the base `10`. If not, linear
    interpolation is used.

(...)
```

```
# Run the `blob_log` function
blobs_log = blob_log(image_gray, max_sigma=30, num_sigma=10, threshold=.1)
blobs_log
```

```
array([[ 499.        ,  435.        ,    1.        ],
       [ 499.        ,  386.        ,    4.22222222],
       [ 499.        ,  342.        ,    1.        ],
       ...,
       [   3.        ,  196.        ,    1.        ],
       [   3.        ,  152.        ,    1.        ],
       [   0.        ,  305.        ,    1.        ]])
```

The returned array contains the x and y positions, and the σ of detection of all detected blobs, one per row.

From σ, the radius of the blobs can be calculated as $\sigma \cdot \sqrt{2}$.

```
# We compute the radii from the 3rd columns (indexing is 0-based)
blobs_log[:, 2] = blobs_log[:, 2] * sqrt(2)
```

We can now do the same with the **Difference of Gaussians (DoG)** detector `blob_dog`.

```
?blob_dog
```

```
Signature: blob_dog(image, min_sigma=1, max_sigma=50, sigma_ratio=1.6, threshold=2.0,
                    overlap=0.5)
Docstring:
Finds blobs in the given grayscale image.

Blobs are found using the Difference of Gaussian (DoG) method [1]_.
For each blob found, the method returns its coordinates and the standard
deviation of the Gaussian kernel that detected the blob.

Parameters
----------
image : ndarray
    Input grayscale image, blobs are assumed to be light on dark
    background (white on black).
min_sigma : float, optional
    The minimum standard deviation for Gaussian Kernel. Keep this low to
    detect smaller blobs.
max_sigma : float, optional
    The maximum standard deviation for Gaussian Kernel. Keep this high to
    detect larger blobs.
sigma_ratio : float, optional
    The ratio between the standard deviation of Gaussian Kernels used for
    computing the Difference of Gaussians
threshold : float, optional.
    The absolute lower bound for scale space maxima. Local maxima smaller
    than thresh are ignored. Reduce this to detect blobs with less
    intensities.
overlap : float, optional
    A value between 0 and 1. If the area of two blobs overlaps by a
    fraction greater than `threshold`, the smaller blob is eliminated.

(...)
```

In analogy to the parameters for the `blob_log` function, also the `blob_dog` function allows specifying a range of sigma values from `min_sigma` to `max_sigma`. The ratio of the two σ for the detector is specified with the `sigma_ratio` parameter.

```
# Run the `blob_dog` function
blobs_dog = blob_dog(image_gray, max_sigma=30, threshold=.1)
```

Again, we calculate the radius of the blobs as $\sigma \cdot \sqrt{2}$.

```
# We compute the radii from the 3rd columns (indexing is 0-based)
blobs_dog[:, 2] = blobs_dog[:, 2] * sqrt(2)
```

6.4.4 Display the results

Now we are ready to plot the results.

```
blobs_list = [blobs_log, blobs_dog]
colors = ['yellow', 'lime']
titles = ['Laplacian of Gaussian', 'Difference of Gaussian']
sequence = zip(blobs_list, colors, titles)
```

```
fig, axes = plt.subplots(1, 2, figsize=(9, 3), sharex=True, sharey=True,
                         subplot_kw={'adjustable': 'box-forced'})
ax = axes.ravel()

for idx, (blobs, color, title) in enumerate(sequence):
    ax[idx].set_title(title)
    ax[idx].imshow(image, interpolation='nearest')
    for blob in blobs:
        y, x, r = blob
        c = plt.Circle((x, y), r, color=color, linewidth=2, fill=False)
        ax[idx].add_patch(c)
    ax[idx].set_axis_off()

plt.tight_layout()
plt.show()
```

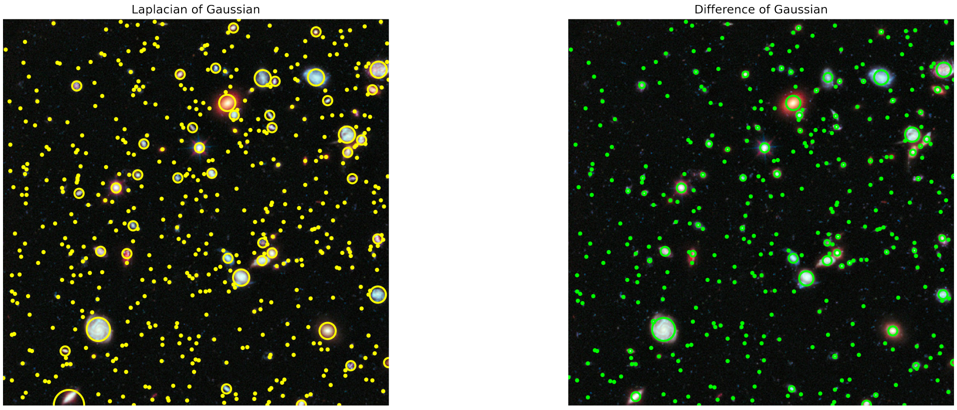


Figure 73: Result of LoG and DoG detection

6.5 Example 5: implementing iterative thresholding

In the last example, we will look at the implementation of the **iterative thresholding** algorithm. The algorithm can be formulated as follows:

Algorithm:

- Choose an initial threshold t, either randomly or according to any method desired
- Segment the image I into two sets:
  - $G_1 = \{I > t\}$
  - $G_2 = \{I \leq t\}$
- Calculate the average of each set:
  - $m_1 = mean\,(G_1)$
  - $m_2 = mean\,(G_2)$
- Update the threshold t as the average of m_1 and m_2
  - $t = (m_1 + m_2)\,/2$
- Repeat steps 2 - 4 until the new threshold t matches the one in the previous iteration (*i.e.*, the algorithm has *converged*)

Our goal is to have a python **function** that implements it, and you can find it ready to go in `code/python/notebooks/`, both as a python file `iterative_thresholding.py` and a Jupyter notebook `iterative_thresholding.ipynb`.

Here we will open the notebook. At the Anaconda prompt, change to the folder containing the code and start Jupyter with `jupyter notebook` (or `jupyter lab`):

```
$ cd code/python/notebooks/
$ jupyter notebook
```

In Jupyter, select `iterative_thresholding.ipynb` to open the notebook. The first code cell in the notebook is the actual algorithm implementation:

```
def iterative_threshold(img, t):
    """Implements iterative thresholding.

    @param img: Image to be segmented (NumPy array)
    @param t  : Initial threshold (float)

    @return Tuple of black-and-white image and final threshold value.
    """

    # Make sure the initial threshold t does not give empty sets
    mx = img.max()
    mn = img.min()

    if t >= mx or t < mn:
        t = (mn + mx) / 2
        print(f"The initial threshold was outside the dynamic "
              "range of the image and was reset to {t}.")

    # This is the actual algorithm
    tlast = t + 1.0

    while abs(t - tlast) > 0.05:

        # Get the two subsets
        G1 = img[img > t]
        G2 = img[img <= t]

        # Calculate the means
        m1 = G1.mean()
        m2 = G2.mean()

        # Calculate the new threshold (but first store last value)
        tlast = t
        t = (m1 + m2) / 2

    # Segment with the calculated threshold
    BW = img > t

    # Return black-and-white mask and last value of t
    return BW, t
```

The actual algorithm starts at the lines:

```
# This is the actual algorithm
tlast = t + 1.0
```

The previous lines take care of the case where the initial threshold is outside of the dynamic range of the image and creates an empty set when splitting the image pixel intensities.

Inside the main loop of the algorithm, we use current threshold `t` to extract to subsets of all pixel intensities: those larger that `t`, and those smaller or equal to `t`. We calculate the mean values of those two sets, and update our current estimate of `t` to correspond to the mean of the means of the sets. Before we update `t`, we keep track of current value, so that we can test the difference `abs(t - tlast)` as a condition to continue iterative in the while loop. When the difference falls under 0.5 we consider the algorithm to have converged to a solution.

Finally, the line:

```
BW = img > t
```

creates a binary mask.

To test our algorithm, following lines open the image file `Nuclei.tif` and pass it to the `iterative_thresholding()` function with a starting threshold of 2000.

```
# Read the example image
img = imread("Nuclei.tif")

# Show image
plt.imshow(img, cmap="gray")

# Our initial t is set to 2000; try other values!
BW, t = iterative_threshold(img, 2000)

# Show image and black-and-white mask
plt.imshow(BW, cmap="gray")

# Print the final threshold
print(f"The final threshold is t = {t}.")
```

Exercise: How does the final threshold depend on the selection of the initial value?

7 Introduction to regression (*curve fitting*)

In this section, we will briefly introduce the concept of **regression**, using a few simple examples. Regression analysis is a branch of statistical modeling or supervised machine learning (depending on whom you ask) that tries to estimate the relationship between a **dependent variable** (or **outcome**) and one or more **independent variables** (or **predictors**, **covariates**, **features**).

The goal of regression is to find the **parameters** of a **model** (for example a **line** in simple **linear regression**) that most closely **fits** some **data** according to a specific mathematical criterion. An important application of regression analysis is to **predict** or **forecast** new and future outcomes based on a model estimated from past data.

The choice of the regression model to estimate is a function of the expected relationship between the predictors and the outcome. The table below gives a quick overview. Common notation is y for the outcome, x for the predictors, b_0, b_1, b_2, ... b_n the parameters. In case of multiple predictors, the notation X (since the data is ordered in a **matrix** instead of a vector) is used.

It is important to note, that data very rarely perfectly fit on the curve predicted by the model. There is always a **residual** deviation between the **predicted values** $\hat{y}$ (pronounced *"y hat"*) calculated using the model equation with the estimated parameters, and the **data values** y. This residual e is calculated for each observation as the difference between the observed value y and the predicted value $\hat{y}$. Mathematically, it can be expressed as $e = \hat{y} - y$.

In regression, it is customary to express a model in the form $y = b_0 + b_1x + e$, where the residual e is not a parameter for which a value is to be found, but rather an *error* that should be minimized. The goal of regression is to find the optimal values for all parameters b_0, b_1, ..., b_n under the **constraint that** e **is minimal** (and $\hat{y}$ is as close to y as possible). For this we need to define a function of the residual e, usually called the **loss function**, that we try to minimize. One possible (and the most commonly used) loss function is the **sum of squared differences**, $\sum(\hat{y} - y)^2$.

For linear regression, one way to find the optimal set of parameters that minimizes the sum of squared differences is the **ordinary least squares** (OLS) method, but in our examples we will use **optimization** techniques (that relax some of the stringent assumptions of OLS).

Finally, the **goodness** of the (linear) fit can be expressed by the value R^2: its **coefficient of determination**. R^2 represents the proportion of the variance in the dependent variable that is predictable from the independent variables. R^2 can vary between 0 and 1, with $R^2 = 0$ showing that there is not relationship (correlation) between changes in the independent values and changes in the dependent value, and $R^2 = 1$ indicating that a given change in x results in a perfectly defined change in y[34].

[34] Please notice that while R^2 can be a useful metric for assessing the goodness of fit (especially for simple linear regression

Type of regression	Model equation	Examples
Simple linear	$y = b_0 + b_1 x$	Sales of product; Economic growth; Impact of GPA on college admissions
Multivariate linear	$y = b_0 + b_1 x_1 + b_2 x_2 + ... + b_n x_n$	CO_2 emission of a car based on engine size, cylinders, fuel consumption
Polynomial	$y = b_0 + b_1 x + b_2 x^2 + ... + b_n x^N$	Length of bluegill fish related to its age (Cook and Weisberg, 1999)
Exponential	$y = b_0 + e^{b_1 x}$	Growth of microorganisms in culture; Rate of food spoilage; Pandemics (such as COVID-19); Fire propagation
Sinusoidal	$y = b_0 \sin(b_1(x + b_2)) + b_3$	Sound waves; Electrical currents; Low and high tides of the ocean
Logarithmic	$y = b_0 + b_1 \ln(x)$	Magnitude of earthquakes; Enzyme reactions; Acidity of a solution; Yields of chemical reactions; Growth of infants.

Table 2: Types of regression.

R^2 [35] is calculated as 1 minus the ratio of the **sum of squared errors** (**SSE**) and the **total sum of squares** (**TSS**):

$$R^2 = 1 - \frac{\sum (\hat{y} - y)^2}{(n-1)\operatorname{var}(y)}.$$

Please note that in statistical modeling, it is also important to assess the **significance** of each parameter contribution to the model outcome; but for this introductory course we will not go into this level of details.

Regression, in particular **linear** regression, is **based on several key assumptions**. Understanding and checking these assumptions is crucial for the validity of the results. For instance, the residuals should be independent of each other and approximately normally distributed, the variance of the residuals should be constant across all levels of the independent variables (homoscedasticity), and there should be no outliers in the data (the assumption of homoscedasticity can be somewhat relaxed in the case of non-linear regression). In this course, we will use **optimization techniques** for regression, that will allows us to address (and/or relax) some of these assumptions.

Let's have a look at a couple of examples to get a better intuition of regression. You can follow along on the Jupyter notebook `code/python/notebooks/regression.ipynb`.

7.1 Simple linear regression

Let's imagine we are given or have measured some data points that we suspect have a linear dependence on some **independent variable** (for instance, time). Because of some noise in the measurement process or

models with one independent variable), it is not an absolute measure of the quality or appropriateness of the model. A high R^2 does not confirm that the chosen model is the most appropriate for the data, nor does it validate the assumptions underlying linear regression, such as linearity, independence, homoscedasticity, or normality of residuals.

[35] You can usually reduce the residuals in a linear model by adding additional terms, *e.g.*, $y = b_0 + b_1 x_1 + b_2 x_2 + ... + e$. When you add more terms, you increase the coefficient of determination, R^2. You get a closer fit to the data, but at the expense of a more complex model, for which R^2 cannot account. However, a refinement of this statistic, *adjusted* R^2, does include a penalty for the number of terms in a model. Adjusted R^2, therefore, is more appropriate for comparing how different models fit to the same data. The adjusted R^2 is defined as: $R^2_{adj} = \left(1 - \frac{\sum (\hat{y}-y)^2}{(n-1)\operatorname{var}(y)}\right) \frac{n-1}{n-p-1}$, where n is the number of observations in your data, and *p* is the total number of explanatory variables in the model (not including the constant term).

some intrinsic variability in what we have measured, even if the underlying relationship is strictly linear, our measurement points will not fall exactly on a line.

As we said in the introduction, we want to fit the model $y = ax + b + e$ to our data[36], where y represents our measured data points, x is the independent variable (time or concentration), a and b are the **parameters** of our model that we want to estimate, and e is the deviation (error) between the real values y and the values $\hat{y}$ predicted by our model. The parameter a represents the **slope** of the line we are fitting to the data, and b is the **intercept** (or **bias**), that is the value of y for $x = 0$.

If the data perfectly falls on the modeled line, the error e is zero. In general, this will not be the case, and the goal of model fitting is to find the curve that passes as close to all (x, y) pairs as possible thus *minimizing* the value of the **residuals** e.

To get an intuition of model fitting, we will try to estimate the values of a and b in a series of attempts of increasing complexity.

7.1.1 Raw data

Let's start by creating some data to fit. Since we create the data ourselves, we know the real values a_{real} and b_{real} that we will try to *discover* by linear fitting. We also add some normally distributed noise to the data to simulate a real measurement or acquisition.

```
# Known parameters of the model
a_real      = 3.0    # Real slope of the line.
b_real      = 10.0   # Real intercept of the line.
noise_level = 1.0    # Standard deviation of the noise around the line to mimick
                     # noisy measurements.

# Fix the seed of the random number generator
rng = np.random.RandomState(1)

# Our raw data
x = np.linspace(start=-10.0, stop=10.0, num=101)
y = a_real * x + b_real + noise_level * rng.randn(len(x))
```

We plot the generated data [37].

```
# Plot the data
plot_data(x, y)
```

[36] The equation of the line is usually represented as $y = ax + b$ which is of course the same as $y = b_0 + b_1 x$ in our earlier notation.

[37] All the plots in this section are created with the `iaf.plot` module.

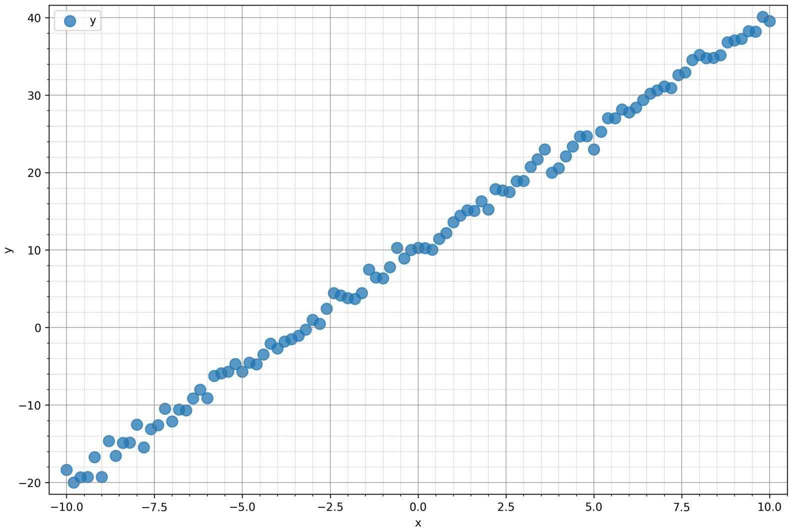


Figure 74: Our noisy, raw data

7.1.2 Our model

Let's first define a function `linear_model()` [38] that we can use in the following attempts to calculate y_{hat} as a function of the current estimate of our parameters a and b for the independent variable x.

```
def linear_model(x, a, b):
    """Our linear model y = a * x + b."""
    y_hat = a * x + b
    return y_hat
```

7.1.3 Assessing the quality of a fit

In order to assess whether a model is a good fit for the data, we have to somehow quantify how close the predicted data is to the real data. The **loss function** quantifies exactly this difference. There are many potential loss functions that could be used, but a very common one in curve fitting is the **sum of squared differences** (**SSE**). The **SSE** represents the *total* distance between the data points y and the predicted values $\hat{y}$.

Let's define a function `calcSSE()` that takes `y` and `y_hat` as arguments and returns the sum of squared differences.

```
def calc_sse(y, y_hat):
    """Calculate the Sum of Squared Errors between prediction y_hat and data y."""
    sse = np.sum(np.power(y_hat - y, 2))
    return sse
```

7.1.4 Naïve fits

As a first attempt, let's try to fit the data by arbitrarily choosing our parameters to be `a = 1.5` and `b = 0`.

```
# "Fit" a line with slope a = 1.5 and intercept b = 0.0.
a = 1.5; b = 0.0;

# Let's calculate y_hat using our model function and our parameters.
y_hat = linear_model(x, a, b);

# And finally the SSE of our first model.
sse = calc_sse(y, y_hat)
```

[38] A few models and metrics are implemented in the `iaf.fit` module.

```
sse =
   18169.485449511147
```

Let's put everything into a plot.

```
plot_data(x, y, y_hat=y_hat, a=a, b=b, sse=sse)
```

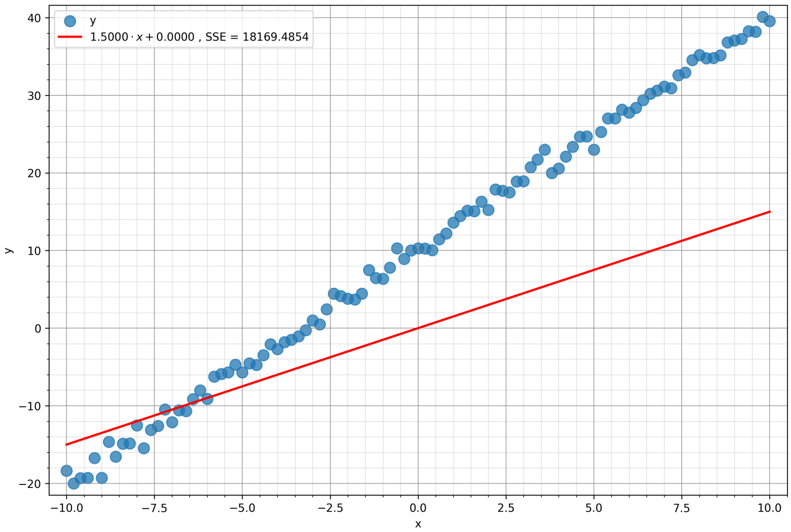


Figure 75: Our first, naïve fit

This does not look like a particularly good fit. The SSE is 18189.4854, which may or may not be small. We cannot say anything about the SSE in **absolute** terms. We will see that we can only use it to **compare** fits to one another. How can we get a better fit to the data? Let's guess again and change our parameters to a = -6 and b = 0.

```
# "Fit" a line with slope a = -6.0 and intercept b = 0.0.
a = -6.0; b = 0.0;

# Let's calculate y_hat using our model function and our parameters.
y_hat = linear_model(x, a, b);

# And finally the SSE of our first model.
sse = calc_sse(y, y_hat)
sse =
   289356.2115105961
```

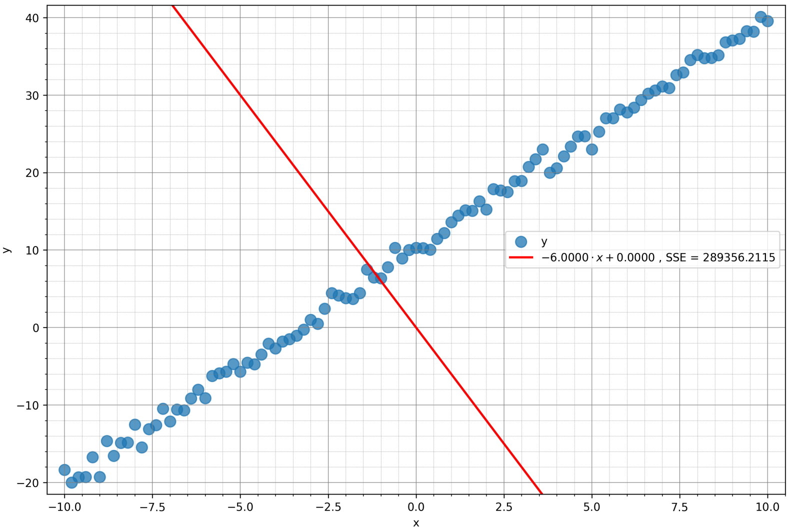


Figure 76: Our second, naïve fit

This looks even worse. Indeed, the value of the SSE increased from about 18,000 to about 290,000!

In both our first fits, we left the value of the intercept b at 0. Indeed, both lines cross at $(0, 0)$. It looks like we should try a slightly steeper slope and an intercept larger than 0. Let's try $a = 3$ and $b = 5$.

```
# "Fit" a line with slope a = 3.0 and intercept b = 5.0.
a = 3.0; b = 5.0;

# Let's calculate y_hat using our model function and our parameters.
y_hat = linear_model(x, a, b);

# And finally the SSE of our first model.
sse = calc_sse(y, y_hat)
sse =
   2660.028670866315
```

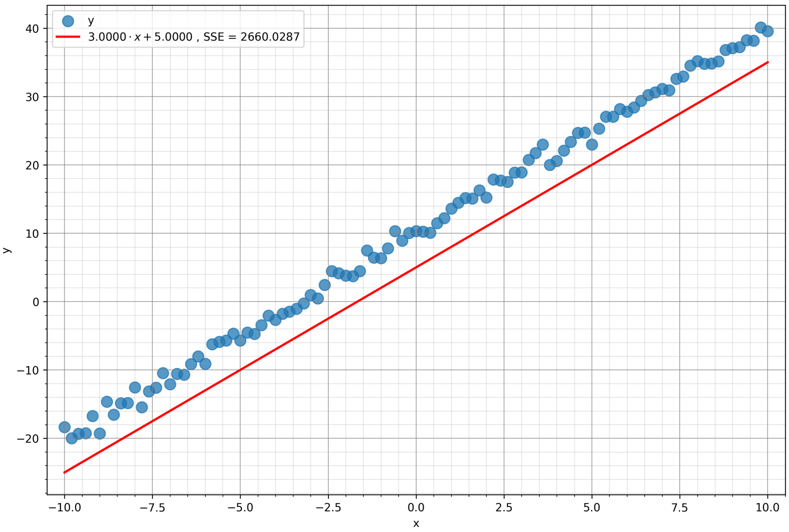


Figure 77: Our third, naïve fit

We are getting closer. The slope $(a = 3)$ seems to be close to the real one, but the intercept b is still too low. Let's try one last time.

```
# "Fit" a line with slope a = 1.5 and intercept b = 0.
a = 3.0; b = 10.0;

# Let's calculate y_hat using our model function and our parameters.
y_hat = linear_model(x, a, b);

# And finally the SSE of our first model.
sse = calc_sse(y, y_hat)
sse =
   78.91710443847684
```

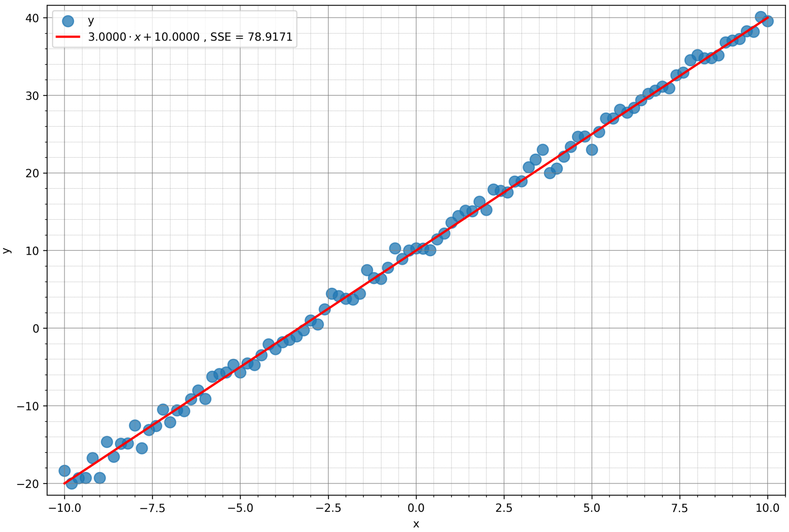


Figure 78: Our last, naïve fit

This is a good fit, as confirmed by the value of the SSE (78.9171), which is clearly the lowest we have seen so far. From this initial investigation, we would then suggest a model $y = 3x + 10$.

7.1.5 First attempt at *searching* for the optimal fit

Trying values at random until we get some apparently correct results does not seem to be the best way to fit a model to some data. Let's see if we can find a better way.

From our first investigation, we seemed to confirm the notion that the best fit to the data is the one with the smallest value of the SSE. Is there a way to actively **search** for the parameters that have the smallest SSE value?

7.1.5.1 Search for the slope a

In our first attempt, we will ignore the intercept b and only search for the slope a with the smallest SSE. We force b to be 0 by calling our `linear_model()` function with $b = 0$.

Let's search over a value of possible values for a.

```
# We search over a range of slope values around a_real = 3.
slopes = np.linspace(start=a_real - 25, stop=a_real + 25, num=51)

# Let's prepare a vector 'SSEs' to store all values of SSE for each of the slopes
SSEs = np.zeros(len(slopes))

# Now we test them all
for i, a in enumerate(slopes):
    y_hat = linear_model(x, a, 0)
    SSEs[i] = calc_sse(y, y_hat)
```

```
# Let's find the best SSE and the corresponding best slope a
SSEs = np.asarray(SSEs)
SSE_best = SSEs.min()
a_best = slopes[np.argmin(SSEs)]
   3.0
```

Let's plot the values of `SSEs` vs. `slopes`.

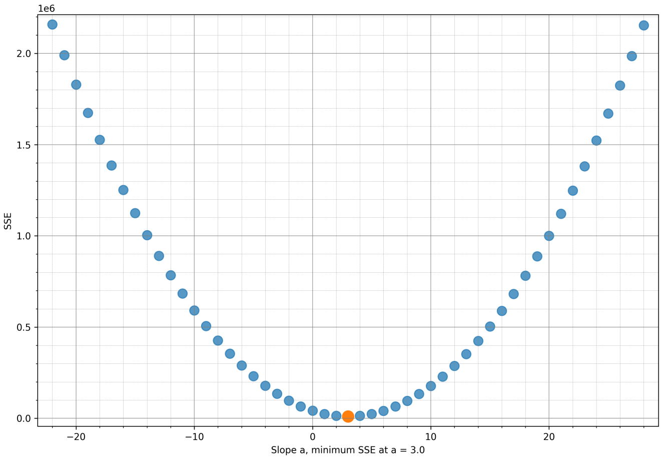


Figure 79: The SSE values as a function of the slope a

Interestingly, it turns out that the values of the SSE as a function of the slope a follow a **convex** curve, with the values decreasing steadily on both sides of the minimum. Our model is therefore $y = 3x + 0 = 3x$.

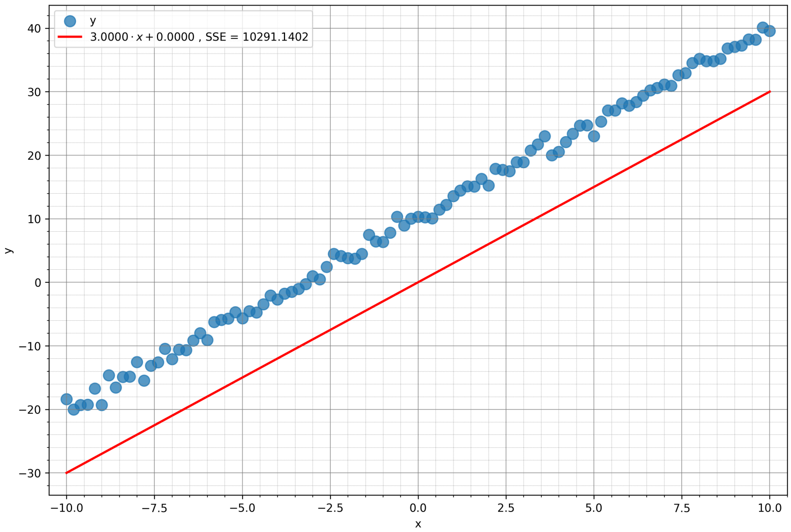


Figure 80: Our fitted data based on the search over the slope a

7.1.5.2 Search for the intercept b

This time we will ignore the slope a and only search for the intercept b with the smallest SSE. We force a to be 0 by calling our `linear_model()` function with a = 0.

Let's search over a value of possible values for b.

```
# We search over a range of intercept values around b_real = 10.
intercepts = np.linspace(start=b_real - 25, stop=b_real + 25, num=51)

# Let's prepare a vector 'SSEs' to store all values of SSE for each of the slopes
SSEs = np.zeros(len(intercepts))

# Now we test the all
for i, b in enumerate(intercepts):
    y_hat = linear_model(x, 0, b)
    SSEs[i] = calc_sse(y, y_hat)

# Find the intercept that corresponds to the smallest SSE value
SSEs = np.asarray(SSEs)
SSE_best = SSEs.min()
b_best = intercepts[np.argmin(SSEs)]
   10.0
```

Let's plot the values of `SSEs` vs. `intercepts`.

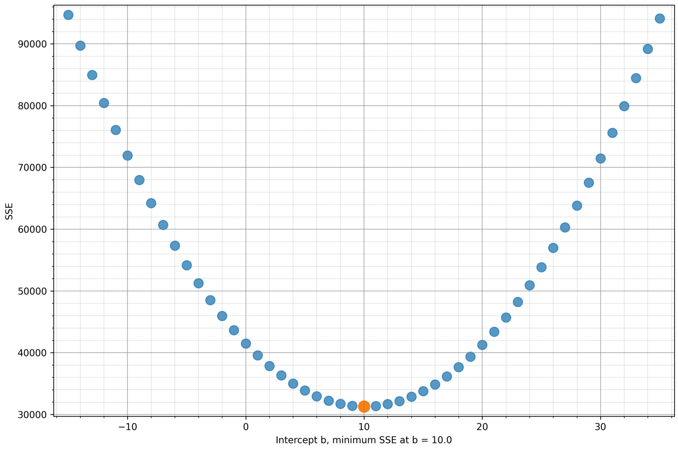


Figure 81: The SSE values as a function of the intercepts b

Again, the SSE values follow a smooth curve with the minimum value in the middle. Our model is therefore $y = 0x + 10 = 10$, *i.e.*, a constant, horizontal line.

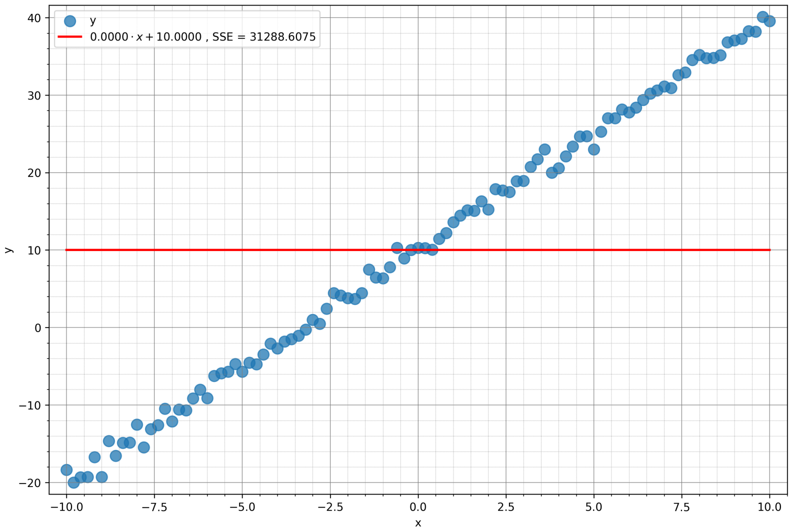


Figure 82: Our fitted data based on the search over the intercepts b

7.1.5.3 Search for the slope a and the intercept b at the same time

In general, one cannot guarantee that optimizing a problem with n parameters one parameter at a time will give the same results as when optimizing over all n parameters in one shot. This is especially true when the relationship between parameters is not linear.

We can extend our **parameter sweep** from one to two dimensions by searching for the best **combination** of parameters a and b that minimize the total SSE of the linear model.

```
# Find the combination of slope and intercept with the smallest SSE
# We use the same slopes and intercepts as before.
slopes = np.linspace(start=a_real - 25, stop=a_real + 25, num=51)
intercepts = np.linspace(start=b_real - 25, stop=b_real + 25, num=51)

# We prepare a matrix to store all values of SSE for each of the (slope, intercept) pairs
SSEs = np.zeros((len(slopes), len(intercepts)))

# Now we test all combinations
for i, a in enumerate(slopes):
    for j, b in enumerate(intercepts):
        y_hat = linear_model(x, a, b)
        SSEs[i, j] = calc_sse(y, y_hat)
```

We can find the optimal values for a and b by searching for the position of the smallest SSE value in the 2D matrix as follows:

```
a_best_index, b_best_index = np.where(SSEs == np.min(SSEs))
a_best = slopes[a_best_index]      # 3.0
b_best = intercepts[b_best_index] # 10.0
```

Since our parameter space is now two-dimensional, the corresponding SSEs cover a surface in 3D space. The blue dot in the figure indicates the position of the minimum SSE for the slope $a = 3$ and the intercept $b = 10$.

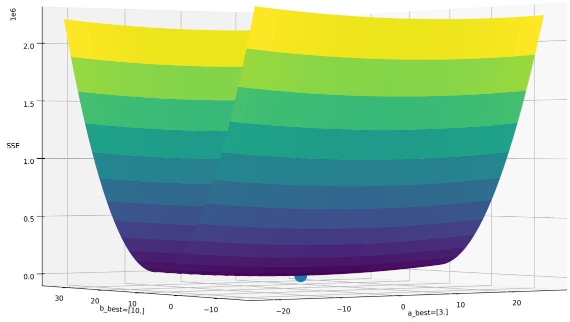


Figure 83: The SSE values as a function of the slopes a and the intercepts b. The blue dot indicates the position of the minimum SSE

Again, we find the expected result $y = 3x + 10$.

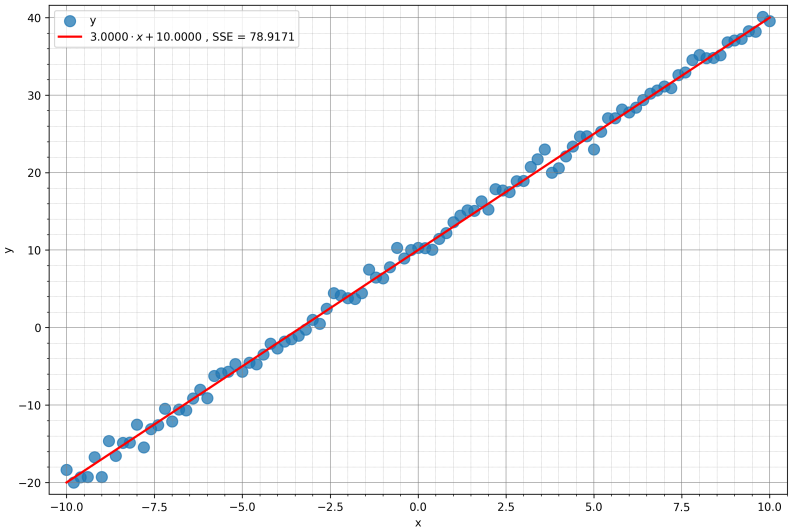


Figure 84: Our fit with the 2-D parameter sweep

7.1.6 Finding the optimal fit by *optimization*

In the previous section, we tested several (combinations of) parameter values to find the model that would predict values $\hat{y}$ as close as possible to the real data points y. In particular, we tested 51 slope values between -22 and 28 (with a step of 1), and 51 intercept values between -15 and 35 (again with a unit step). This gave us a good approximation of the real values $a_{real} = 3$ and $b_{real} = 10$.

Before you get too happy with our results, consider that we engineered our raw data to have parameter values (3 and 10, respectively) that would be among the ones we tested!

What if the real values were $a_{real} = 3.19583$ and $b_{real} = 9.77522$ and we were asked to estimate them with 10^{-5} accuracy? Our previous search was limited by the resolution of the **grid** of values we created. Assuming that we can estimate a reasonable search range around the correct parameters (which is not too complicated for a 2-D linear model by looking at the scatter plot of the data, but can quickly become impossible with more dimensions and more complicated models), we would still have to test at least 10^{-5} values between each two consecutive integers of the search range (assuming no noise). In our specific example, we would need to test $\approx 2.6 \cdot 10^{13}$ combinations of parameters a and b. You can see that this approach rapidly becomes prohibitively expensive.

If we stop one minute to think about the shape of the loss function, we realize that we don't need to test all combinations of parameters. We can imagine the loss function to be the internal surface of a bowl, and any value of the loss function $L(a, b)$ to be a drop of water on that surface. Left alone under the effect of gravity, the droplet will accelerate towards the bottom of the bowl. In analogy, from a position $L(a, b)$ we can use **optimization techniques** to quickly follow the gradient of the loss function towards its minimum. The SciPy (https://scipy.org) scientific library for Python provides the function `fmin()` in its `optimize` package[39], that uses numerical analysis techniques to find the minimum of multi-parameter functions. Use:

```
?optimize.fmin
```

in IPython or a Jupyter notebook to learn more about `fmin()`, or navigate to https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.fmin.html in your browser.

What are we trying to optimize? Our goal is to find the parameters `a_best` and `b_best` that plugged into our linear model `ax + b` result in a predicted value `y_hat` that **minimizes** the error (distance) from the real data `y`. We know how to calculate this error: it is the SSE between `y_hat` and `y`. Therefore, the function to minimize is `calc_sse()`. However, for every value of `a` and `b` we test during the optimization, we need to recalculate `y_hat` and in turn update the current value of `sse`. We keep doing this until `y_hat` gets as close to `y` as possible.

`fmin` minimizes an **objective function** `objf` by searching for the combination of model parameters that minimize the *value* of the objective function. Since `fmin` can optimize over any number of parameters, those parameters are fed by the optimizer to the objective function by packing them into a one-dimensional array or tuple and passing them to the objective function as its first argument. Any additional **constant** arguments required by the objective function can be passed as 2^{nd}, 3^{rd}, … n^{th} input arguments.

In our case, we want to find the parameters `a` and `b` that describe the best linear fit to the data, and therefore we pack them into a tuple `(a, b)`. We also need `x` (our independent variable), and `y` (our target) to update `y_hat` and `sse`. `sse` is the value of the objective function that we are trying to minimize.

```python
def calc_sse(y, y_hat):
    """Calculate the Sum of Squared Errors between prediction y_hat and data y."""
    sse = np.sum(np.power(y_hat - y, 2))
    return sse

def linear_model(x, a, b):
    """Our linear model y = a * x + b."""
    y_hat = a * x + b
    return y_hat

def objf(params, x, y):
    """Our objective function."""
    a = params[0]
    b = params[1]
    y_hat = linear_model(x, a, b)
    sse = calc_sse(y_hat, y)
    return sse
```

In every iteration of the optimisation, `fmin` calls `objf` with current values `(a, b)`, and the constants `x` and `y`. In turns, `objf` calls `linear_model` passing `(a, b)` and `x` to calculate `y_hat`, and then `calc_sse` passing the new value of `y_hat` and the constant value `y` to update `sse`. As long as the values of `a` and `b` get closer to the real values `a_real` and `b_real`, the value of `sse` will decrease.

`fmin()` can now search for the minimum value of SSE from an initial starting point that we provide.

```python
# Starting values for the optimization (initial values for a and b)
starting_values = (0.0, 0.0)

# Run the optimization
res = optimize.fmin(objf, starting_values, args=(x, y))
```

```
Optimization terminated successfully.
```

[39] We could implement gradient descent ourselves, but for this course we will use a *black-box* approach and not dive into the details of implementation.

```
         Current function value: 77.859336  # This is sse
         Iterations: 87
         Function evaluations: 164
```

The results are stored in the res tuple.

```
a_est = res[0]
   3.014742636785444

b_est = res[1]
   10.055531398788752
```

fmin finds an optimal value for slope $a = 3.0147$ and intercept $b = 10.0555$. If we calculate the SSE for the obtained parameter values, we obtain:

```
y_hat = linear_model(x, a_est, b_est)
sse = calc_sse(y, y_hat)
   77.8593
```

Notice that fmin() found a combination of values a and b that give a lower SSE (77.8593) than what we obtained with our sweeps (78.9171).

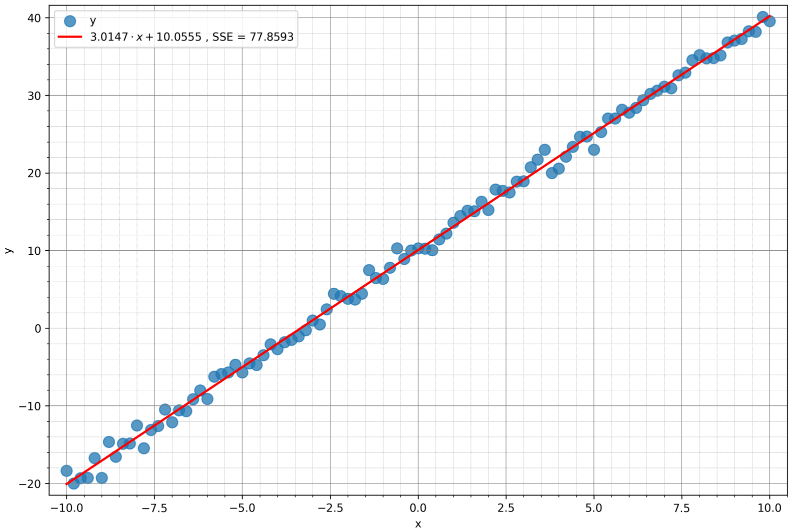


Figure 85: Fit obtained via fmin()

Question: why is the combination of parameters obtained with fmin() giving us a lower value of the SSE for the model $y = 3.0147x + 10.0555$ even though our real data was created with the model $y = 3x + 10$?

7.1.7 Fitting models with repeated measures

In many experiments, we collect several measurements, or **replicates,** y for each value x. In this section, we want to fit a line that takes those repetitions into consideration. For our first example, we will create nReps = 5 replicates y_i with the same normally-distributed error:

```
# New parameters
nReps = 5   # Number of repeated (independent) measurements y.
w0    = 3.0 # Noise level on the repeated measurement example.

# We create nReps lines with some error. We reduce a bit the density of
# the data points for clarity.
x = np.linspace(start=x[0], stop=x[-1], num=21)
yy = np.zeros((nReps, len(x)))
for i in range(nReps):
    yy[i, :] = a_real * x + b_real + w0 * rng.randn(len(x))
```

```
yy.shape
   (5, 21)
```

For the fitting, we will need the same number of values in x as we have in yy. Since yy is a 5x21 matrix, we replicate the 1x21 row vector 5 times to create a 5x21 matrix that we call xx. This way, each row y_i has the same, corresponding range of x values.

```
xx = np.repeat(np.reshape(x, (1, len(x))), nReps, axis=0)

xx.shape
   (5, 21)
```

If we plot YY vs. XX, we get the following:

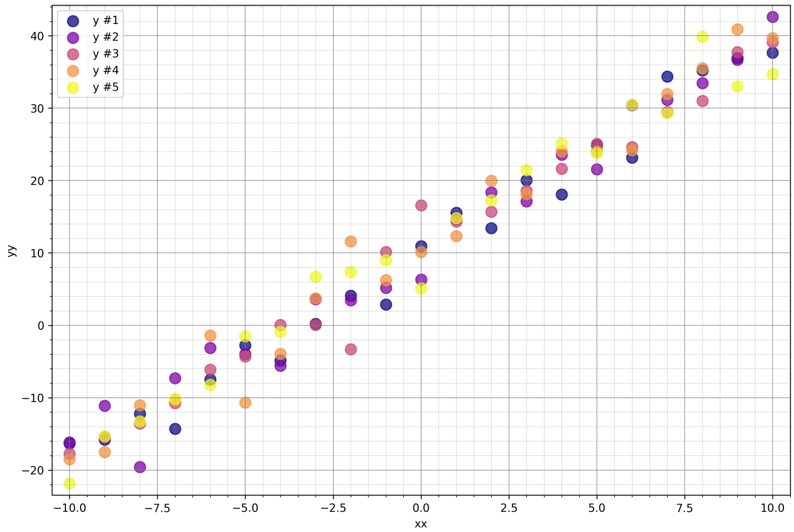


Figure 86: Data with repetitions

Each value x_i has 5 values $y_{i,j=1..5}$ associated to it. For example, we have 5 (x, y_j) pairs for $x = 0$: $(0, 10.9455), (0, 6.3098), (0, 16.5609), (0, 10.1310), (0, 5.1177)$.

As before, we use fmin() to find the best parameters to fit the data with a linear model. fmin() can cope with the repetitions in our data without any problem and fits the line that minimizes the loss of all points. The functions objf, calc_sse and linear_model are the same we defined earlier.

```
# Starting values for the optimization (initial values for a and b)
starting_values = (0.0, 0.0)

# Run the optimization
res_reps = optimize.fmin(objf, starting_values, args=(xx, yy))

# Let's see the results
a_est_reps = res_reps[0]
   2.90722396712126

best_reps = res_reps[1]
   10.412797956593149
```

If we plot the resulting linear model on the data points we obtain the following.

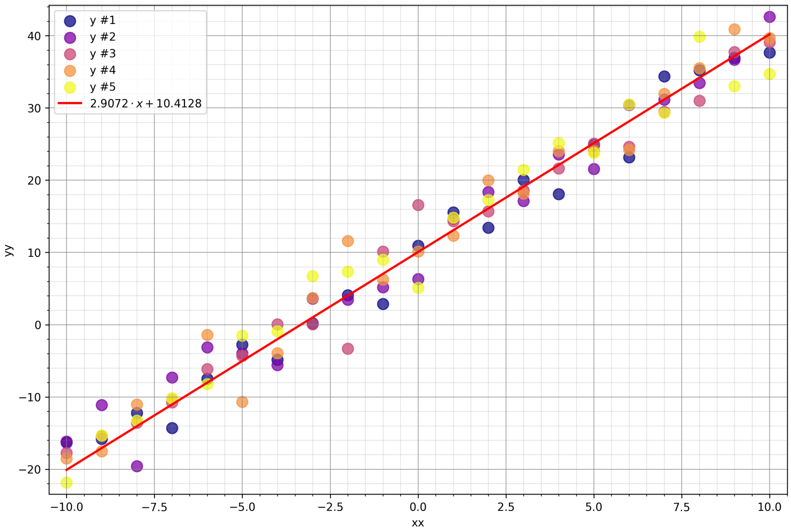


Figure 87: Model fitted to the data with repetitions

Please notice that since all data points are given the **same weight** in the optimization, the obtained result is the same as we would obtain if we fit a line through the mean of the y measurements.

```
# Compare with fitting to the mean of yy
yy_mean = yy.mean(axis=0)

yy_mean.shape
   (1, 21)

# Starting values for the optimization (initial values for a and b)
starting_values = (0.0, 0.0)

# Run the optimization (notice that we use 'x' this time)
res_mean = optimize.fmin(objf, starting_values, args=(x, yy_mean))

# Let's see the results
a_est_mean = res_mean[0]
   2.90722396712126

best_mean = res_mean[1]
   10.412797956593149
```

If we plot the resulting linear model on the data points we obtain the following.

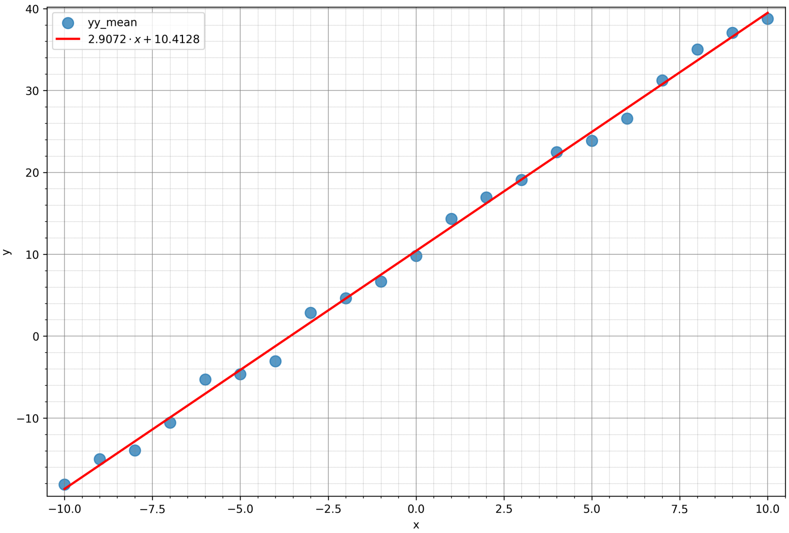


Figure 88: Model fitted to the mean of the data with repetitions

If we compare the model obtained fitting all 5 y replicates or fitting their mean, we obtain the same values for slope and intercept.

```
All data:   (slope = 2.90722397, intercept = 10.41279796)
From mean:  (slope = 2.90722397, intercept = 10.41279796)
Difference: (        0.00000000,              0.00000000)
```

In both cases, we obtain the same model $y = 2.9072 + 10.4128$.

7.1.8 Fitting models with weighted, repeated measures

In the previous example, all replicates y_i had the same error (statistically), since we modeled the repetitions with from a **normal distribution** with mean 0 and a constant standard deviation:

```
for i in range(nReps):
    yy[i, :] = a_real * x + b_real + w0 * rng.randn(len(x))
```

This time, we will simulate some troublesome data acquisition. After an initial phase where everything works fine, something unfortunately breaks, and the second half of the data suddenly displays a large error and even an additional bias (a shift in the values).

We simulate this data with the following code:

```
# New parameters
nReps       = 5   # Number of repeated (independent) measurements y.
w0          = 3.0 # Noise level on the repeated measurement example.
w1          = 0.5 # Weight that scales the standard deviation of the noise
                  # for the first half of the data.
w2          = 5.0 # Weight that scales the standard deviation of the noise
                  # for the second half of the data.
data_offset = 5.0 # Amount by which the second half of the data is moved
                  # away from the ideal fit.

# We create nReps lines with some error. The first half of the points has
# an error of +/- w1, whereas the second half has an error of +/- w2 and
# even a constant shift 'data_offset'.
x = np.linspace(start=x[0], stop=x[-1], num=21)

n1 = len(x) // 2
n2 = len(x) - n1
```

```
yy = np.zeros((nReps, len(x)))
for i in range(nReps):
    noise = np.zeros(len(x))
    noise[:n1] = w1 * rng.randn(n1)
    noise[n1:] = data_offset + w2 * rng.randn(n2)
    yy[i, :] = a_real * x + b_real + w0 * noise

yy.shape
   (5, 21)

# Let's also create the 5 repetitions of x to have the same number of values as in yy.
xx = np.repeat(np.reshape(x, (1, len(x))), nReps, axis=0)

xx.shape
   (5, 21)
```

The generated data looks like this:

```
plot_data(xx, yy)
```

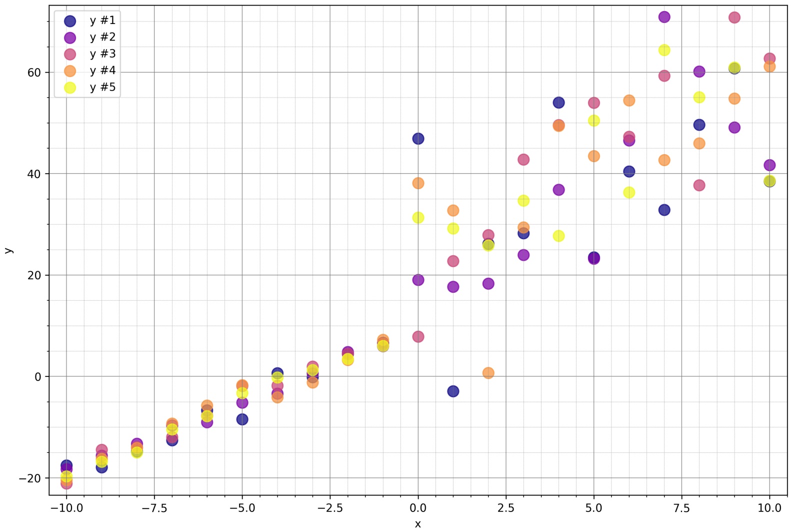


Figure 89: Our bad acquisition

Should we go ahead and fit a model to this data? The second half of the data is obviously going to negatively affect the quality of the fit. Intuitively, we should do something about this. But before we look for any possible solutions, let's just fit this data as we have done so far. What do we get?

```
# Starting values for the optimization (initial values for a and b)
starting_values = (0.0, 0.0)

# Run the optimization
res_reps = optimize.fmin(objf, starting_values, args=(xx, yy))

# Let's see the results.
a_est_reps = res_reps[0]
   4.120894273612725
b_est_reps = res_reps[1]
   17.912701387702306
```

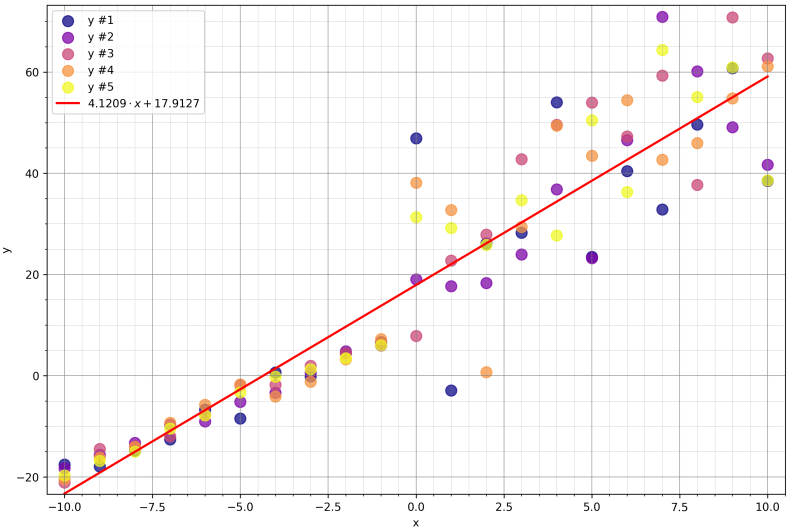


Figure 90: Straightforward fit of our bad acquisition

The optimization gives all points the same importance and causes the red line to be a poor fit for the first part of the data, that we know to be good. The second part of the data may or may not be represented appropriately by the model; it is difficult to say because the spread of the data is so large. In any case, the model we obtain is $4.1209x + 17.9127$ which is very far from the model we used to create the data (ignoring the bias term we introduced, of course).

Let's start thinking about alternative approaches. We might decide to drop the data points for $x \geq 0$, but here we will assume that we need to use them (maybe the acquisition procedure is very expensive). What we can do instead, is to (indirectly) inform our optimizer that we want to give less weight to the bad data points than we do to the good ones. One way of doing this, is by modifying our loss function so that we give more importance to good data points while we penalize bad data. How can we do that?

Since we have 5 values of y for every value of x, we can calculate the **standard error of the mean** for the replicates of y and use them to **weigh the contributions of the data points in the fit**. The standard error of $y_{i,j=1..5}$ for each position x_i is defined as the standard deviation of $y_{i,j=1..5}$ divided by the square root of 5 (the number of elements in $y_{i,j=1..5}$), and gives us a measure of the uncertainty in estimating the real population mean based on our sample mean.

```
# Calculate the standard error of the mean (per point)
sd = yy.std(axis = 0, ddof = 1)
se = sd / np.sqrt(nReps)
```

Let's plot the standard errors on our data points.

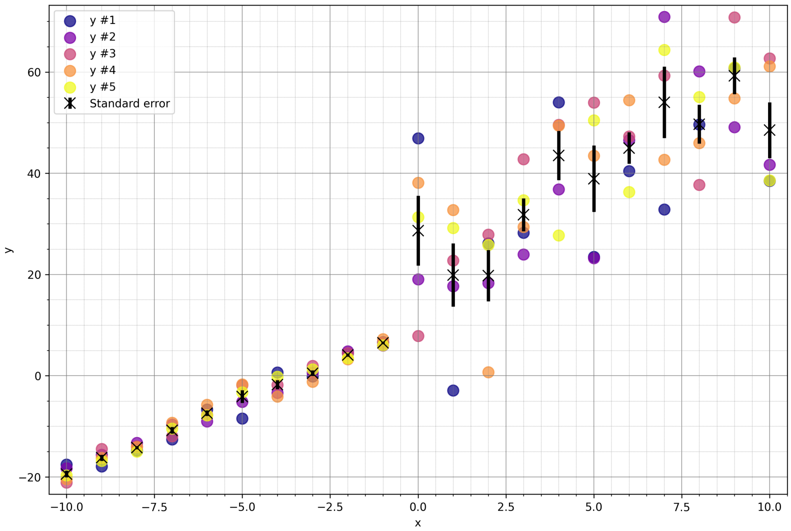


Figure 91: Our bad acquisition with standard errors

The errors for the first half of the data are obviously much smaller than those for the second half.

Since we calculated them, let's go ahead and use the calculated standard errors to add a weighting factor in the calculation of the loss. For each term, the squared error SE_i is divided by the corresponding standard error of the mean se_i:

$$SE_i = (y_i - \hat{y}_i)/se_i$$

This way, terms with a large standard error contributes little to the total SSE and therefore do not penalize the fit much even if the line does not pass very close to them. We modify out `calc_sse` function to take a vector of standard errors as weights and the objective function `objf` to use this new function instead of the old one. The linear model does not change:

```python
# The model does not change
def linear_model(x, a, b):
    """Our linear model y = a * x + b."""
    y_hat = a * x + b
    return y_hat

# In the calculation of the SSE we now penalize terms with a large standard error.
def calc_sse_weighed(y, y_hat, se):
    """Calculate the Sum of Squared Errors between prediction y_hat and data y."""
    sse = np.sum(np.power(y_hat - y, 2) / np.power(se, 2))
    return sse

# In the objective function, we call the new function calc_sse_weighed
def objf_weighed(params, x, y, se):
    """Our objective function (that calls the weighed version of calc_sse())."""
    a = params[0]
    b = params[1]
    y_hat = linear_model(x, a, b)
    sse = calc_sse_weighed(y_hat, y, se)
    return sse
```

We run our optimization using the new functions:

```python
# Starting values for the optimization (initial values for a and b)
starting_values = (0.0, 0.0)

# Run the optimization
res_reps_weighed = optimize.fmin(objf_weighed, starting_values, args=(x, yy_mean, se))
```

If we plot the weighted fit (red line in the next figure) along with the standard fit (blue), we see that the red line nicely fits the good points (for $-10 \leq x \leq -1$) and passes very far from the bad points.

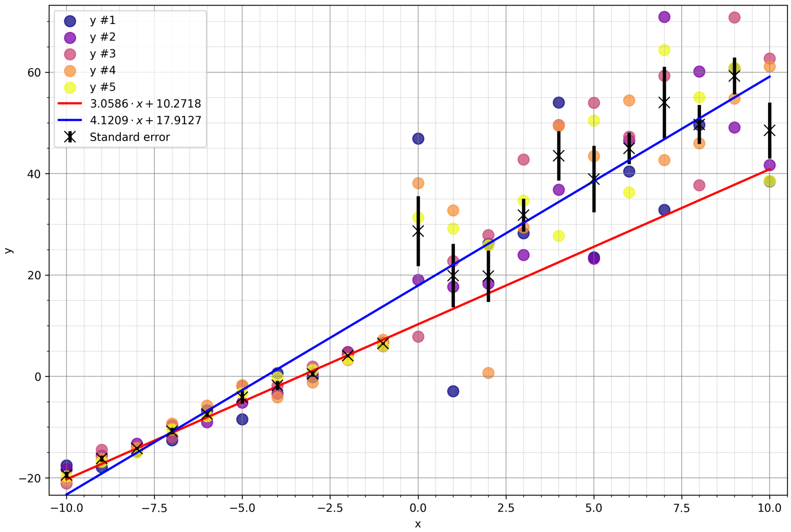


Figure 92: Weighted fit of the replicates

Our weighed model $y = 3.0586x + 10.2718$ is now very robust against the bad measurement points for $x \geq 0$.

7.2 Non-linear models

This builds on the intuitions about linear model fitting. The model we want to fit is an **exponential** of the form $y = a\exp(bx) + e$, where x and y are given and a and b are the parameters of our model that we want to estimate. The term e accounts for deviations (*residuals*) between the model and the noisy data.

7.2.1 Raw data

It is often assumed that cells proliferate exponentially, following a model of the form:

$$M(t) = M(0)e^{\lambda t},$$

where t is time, $M(0)$ is the number of cells at time 0 and $\lambda > 0$ is the proliferation rate. This model assumes that the cells grow synchronously and are homogeneous in the proliferation pattern, which is a good enough approximation for our purposes.

Let's say we followed the proliferation rate by counting the number of cells every 4 hours over a 48-hour experiment and we recorded the following data.

```
t = [0, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 48]
M = [541, 592, 726, 862, 1161, 1203, 1805, 1950, 2516, 2987, 3965, 4048, 5423]
```

Let's have a look at the data.

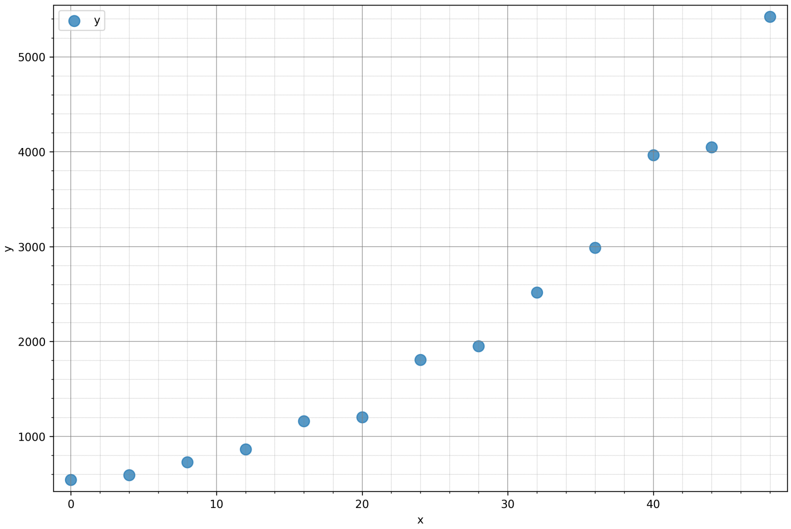


Figure 93: Exponential data

7.2.2 Fit a non-linear model using `optimize.fmin()`

To keep with our previous nomenclature, we will label our independent variable *time* as x and our dependent variable *cell count* as y.

The data shows an **exponential growth** of the form

$$y = a \cdot e^{b \cdot x},$$

where $b > 0$. A series of numbers x increases or decays exponentially, if every number x_i in the series is obtained by scaling the previous number x_{i-1} by a constant factor b. If $b > 0$, the numbers increase, if $b < 0$, the numbers decrease.

As we have done in the case of the linear fit, we can use `fmin()` to find the parameters a and b that minimize the SSE between the real and the predicted values y_{hat}. Notice that we kept the noise level small.

We define our model as follows:

```
def exponential_model(x, a, b):
    """Exponential model y_hat = a * exp(b * x)."""
    y_hat = a * np.exp(b * x)
    return y_hat
```

As we did earlier, we define our `calc_sse()` and `objf()` functions.

```
def calc_sse(y, y_hat):
    """Calculate the Sum of Squared Errors between prediction y_hat and data y."""
    sse = np.sum(np.power(y_hat - y, 2))
    return sse

def objf(params, x, y):
    """Our objective function."""
    a = params[0]
    b = params[1]
    y_hat = exponential_model(x, a, b)
    sse = calc_sse(y_hat, y)
    return sse
```

Since the exponential is a geometric growth from an initial value, we choose the starting value for the parameter a to be the same as the first value in y and a positive value for b since the exponential causes the values of y to increase over time:

```
# Starting values for the optimization (initial values for a and b)
starting_values =[y[0], 1.0]
```

We can now run the optimization:

```
# Now we use fmin to find the solution
res_exp = optimize.fmin(objf, starting_values, args=(x, y))

Optimization terminated successfully.
         Current function value: 312800.956454
         Iterations: 75
         Function evaluations: 144

a_exp = res_exp[0]
   516.9158688896242
b_exp = res_exp[1]
   0.04869571366678492
```

Let's calculate the predicted values and the SSE using our model:

```
y_hat = exponential_model(x, a_exp, b_exp)
sse = calc_sse(y, y_hat)
```

We plot them and get the following graph.

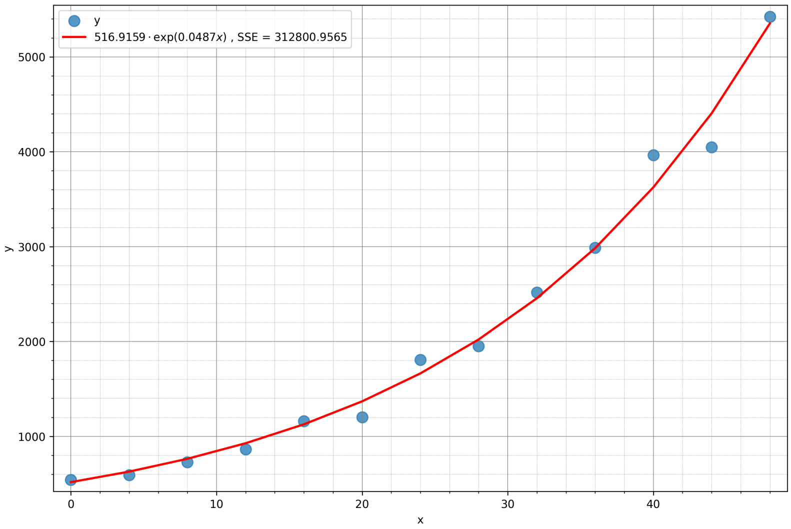


Figure 94: Exponential fit using the fmin() function

Our model is therefore $y = 516.9159 \cdot e^{0.0487 \cdot x}$, with $M(0) = 516.9159$ and $\lambda = 0.0487$.

7.2.3 Regularisation

In the previous section, we discussed how to penalize data replicates with large variances by **weighting** their contribution in the calculation of the SSE (Sum of Squared Errors). However, sometimes one or a few variances may dominate the model parameter selection, leading to **overfitting** or **underfitting**. More complex models with many parameters are generally more prone to overfitting. In data following an exponential trend, for instance, we can expect that the variability around much larger y values will also be significantly larger.

To mitigate this, we can use **regularization**. In regularization, we add a **weighted penalty term** to the objective function being minimized (in our case, the weighted SSE). This penalty term is a function of the current parameter values.

Common types of regularisation include **L1** (**lasso**), **L2** (**ridge**), and combinations of both. For example, L2 regularisation is defined as $\mathrm{L2} = \mathrm{SSE} + \lambda \sum_{j=1}^{p} b_j^2$, where p is the number of parameters b_j and λ is a weighting factor.

Choosing the optimal λ is crucial. Methods for this include **Cross-validation**, **Bayesian Information Criterion** (**BIC**) or **Akaike Information Criterion** (**AIC**).

A detailed treatment of regularization is outside the scope of our introductory course. For our purposes, we will use a simpler approach. In data following an exponential trend, the standard error around large values may be significantly larger than around small values. To address this, we can normalize the standard error to its mean, creating a **relative standard error**:

```
# Calculate the mean across replicates
yy_mean = yy.mean(axis=0)

# Calculate the standard error of the mean (per point)
sd = yy.std(axis = 0, ddof = 1)
se = sd / np.sqrt(nReps)

# Normalize to the absolute values
se = se / yy_mean
```

We can use this corrected vector `se` to weigh our exponential fit in `calc_sse_weighted()`.

7.3 An alternative to optimize.fmin(): optimize.curve_fit()

The `scipy.optimize` package offers an alternative function that is a higher-level approach to model fitting: `curve_fit()`. This function still uses optimization behind the curtains but does most of the work for us. Use:

```
?optimize.curve_fit
```

in IPython or a Jupyter notebook to learn more about `curve_fit()`, or navigate to https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html in your browser.

`curve_fit()` expects a model of the curve to fit. We can again use our exponential function.

```
def exponential_model(x, a, b):
    """Exponential model y_hat = a * exp(b * x)."""
    y_hat = a * np.exp(b * x)
    return y_hat
```

Let's try fitting a curve to our cell count vector `y` above. Please notice that `curve_fit` is quite sensitive to the initial values of the parameters!

```
start = (y[0], 0.5)
popt, _ = optimize.curve_fit(exponential_model, x, y, p0 = start)

a = popt[0]
   516.9158584532371

b = popt[1]
   0.04869571470187618
```

Let's calculate `y_hat` and `sse` with the model and plot the obtained curve on the raw data.

```
y_hat = exponential_model(x, popt[0], popt[1])
sse = calc_sse(y, y_hat)
```

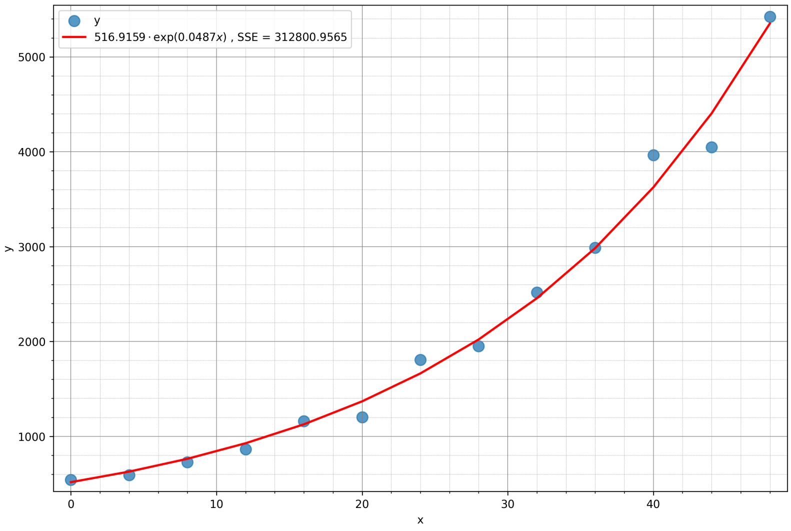


Figure 95: Exponential fit using the curve_fit() function

The obtained model parameters are the same as the ones we obtained using `optimize.fmin()` in the previous section.

7.3.1 Fitting models with repeated, weighed measures using `curve_fit()`

Let's consider the example with (bad) replicates we studied above.

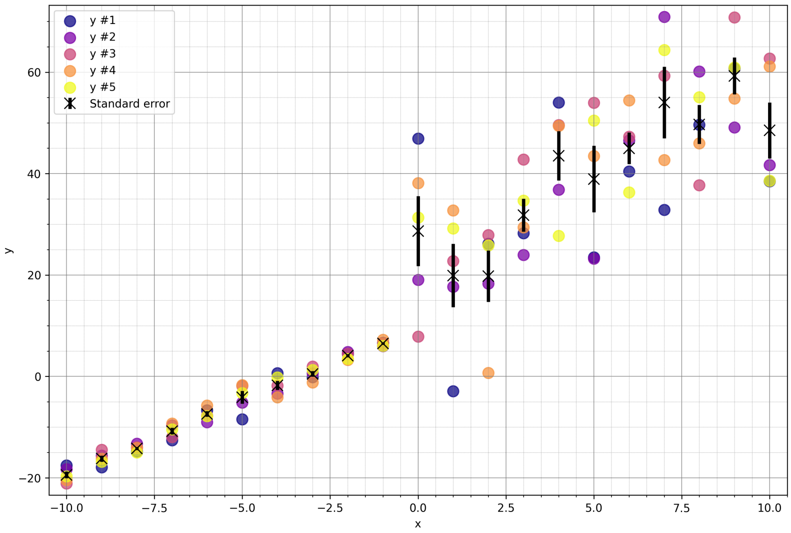


Figure 96: Our bad acquisition with standard errors

As was the case with `fmin`, `curve_fit()` allows us to assign different weights to the points in `y` to modulate their contribution during optimization. The following result is obtained by fitting a linear model through the mean intensities of the `y` replicates weighed by their standard error.

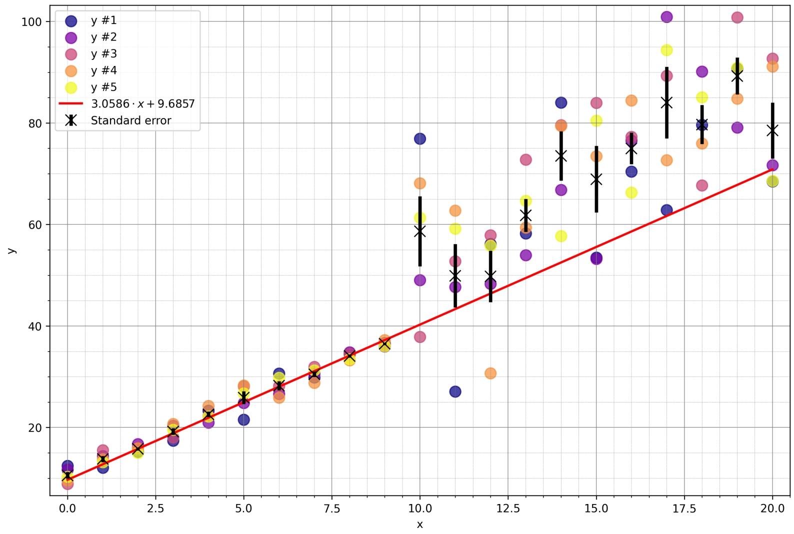


Figure 97: Our bad acquisition with standard errors and a weighted fit

Exercise Try to replicate the result displayed in the figure above using `curve_fit()`.

8 Appendix A

8.1 Setting up our environment (extended instructions)

All libraries needed for this course are marked as dependencies of the `iaf` library. This way, all we need to explicitly install is the `iaf` library itself. Some of these dependencies, however, are published on http://pypi.org as **source** distributions, which means that they need to be compiled on the target machine on installation. To simplify matters, those libraries have already been compiled for this course into binary packages for Windows, Linux and macOS (both on Intel and M1 processors) and published on an alternative (*extra*) pypi index (https://ia-res.ethz.ch/pypi/), so that no local compilation *should* be needed.

If the following installation instructions work, you should see `0.5.0` printed on the console, that is, current version if the `iaf` library.

```
$ conda create -n iaf-env python=3.11
$ conda activate iaf-env
$ pip install --extra-index-url https://ia-res.ethz.ch/pypi iaf
$ python -c "import iaf; print(iaf.__version__)"
```

However, if the installation fails[40], the binary version of the libraries from https://ia-res.ethz/ch is not compatible with the target machine. In this case, a C++ compiler is required on the target machine to build those dependencies.

- On **Windows**, please install the Microsoft C++ Build Tools from https://visualstudio.microsoft.com/visual-cpp-build-tools/. When the Visual Studio Installer starts, select **Desktop development with C++** and click Install. You will have to reboot your machine before you can try again.
- On **Linux**, the commands will depend on your distribution:
  - on **Ubuntu**, please run the following in the terminal: `sudo apt install build-essential;`
  - on **Fedora**, please run `sudo dnf group install "C Development Tools and Libraries" "Development Tools";`
  - for other distributions, please Google: '`sudo apt build-essential equivalent for {your_distro}`'.
- On **macOS**, start the terminal and execute `xcode-select --install`. Follow the instructions on the screen to install the XCode Command Line Tools.

[40] Most likely it will fail with an error similar to `ERROR: Failed building wheel for centrosome`. Please follow the instructions in the text.

Once the C++ compiler is installed, you are ready to proceed with the creation of the `iaf` environment and the installation of all required packages. Open the **Anaconda Prompt** (Windows) or the **terminal** (macOS and Linux) and retry the installation using the commands above.

9 Appendix B

9.1 Selection of good Python libraries

In the main part of the text, we introduced **`scikit-image`**, **`scipy.ndimage`**, and the tiny helper package **`iaf`**. Other very good image processing libraries are **`OpenCV`**, **`SimpleITK`**, **`Mahotas`**, and **`Pillow`**. In this section, we will give a quick introduction and offer relevant links for these libraries.

9.1.1 openCV

Figure 98: openCV logo

OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. The library has more than 2500 optimized algorithms, including a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms.

9.1.1.1 License

OpenCV version 4.5.0 and higher is released under the terms of the Apache License 2.0 (https://opencv.org/license/).

9.1.1.2 Documentation

Website: https://opencv.org/

Tutorials: https://docs.opencv.org/4.8.0/d6/d00/tutorial_py_root.html

9.1.1.3 Installation

To install `OpenCV` using `conda`, use:

```
$ conda install opencv -c conda-forge
```

To install `OpenCV` using `pip`, use:

```
$ pip install opencv-python
```

9.1.2 SimpleITK

Figure 99: SimpleITK logo

SimpleITK is a simplified programming interface to the algorithms and data structures of the Insight Toolkit (ITK). ITK is a large, open-source, cross-platform library written in C++ that provides developers with an extensive suite of software tools for registering, segmenting, analyzing, and quantifying (medical) image data. SimpleITK offers Python binding and a simplified API for ITK.

9.1.2.1 License

SimpleITK is released under the terms of the Apache License 2.0 (https://github.com/SimpleITK/SimpleITK/blob/master/LICENSE).

9.1.2.2 References

- R. Beare, B. C. Lowekamp, Z. Yaniv, "Image Segmentation, Registration and Characterization in R with SimpleITK", J Stat Softw, 86(8), doi: 10.18637/jss.v086.i08, 2018.
- Z. Yaniv, B. C. Lowekamp, H. J. Johnson, R. Beare, "SimpleITK Image-Analysis Notebooks: a Collaborative Environment for Education and Reproducible Research", J Digit Imaging., doi: 10.1007/s10278-017-0037-8, 31(3): 290-303, 2018.
- B. C. Lowekamp, D. T. Chen, L. Ibáñez, D. Blezek, "The Design of SimpleITK", Front. Neuroinform., 7:45. doi: 10.3389/fninf.2013.00045, 2013.

9.1.2.3 Documentation

Website: https://simpleitk.org/

Current documentation: https://simpleitk.readthedocs.io/en/master/index.html

9.1.2.4 Installation

To install `SimpleITK` using `conda`, use:

```
$ conda install simpleitk -c simpleitk
```

To install `SimpleITK` using `pip`, use:

```
$ pip install simpleitk
```

9.1.3 Pillow

Figure 100: scikit-image logo

Pillow is a friendly fork of the original **Python Imaging Library** (PIL) that adds image processing capabilities to Python. Even though the original PIL library is not officially dead, it has not been developed since 2009 and does not support Pyhon 3.x. Pillow was born as a PIL fork to continue its development and migrate it to Python 3.x .

9.1.3.1 License

Pillow and PIL are distributed under the open-source HPND license: https://pillow.readthedocs.io/en/stable/about.html#license.

9.1.3.2 Documentation

Website: https://github.com/python-pillow/Pillow

Current documentation: https://pillow.readthedocs.io/en/stable/

9.1.3.3 Installation

To install `Pillow` using `conda`, use:

```
$ conda install pillow
```

To install `Pillow` using `pip`, use:

```
$ pip install Pillow
```

9.1.4 mahotas

Mahotas: Computer Vision in Python

Figure 101: Mahotas (does not have a logo)

Mahotas is a library of fast computer vision algorithms (all implemented in C++ for speed) operating over numpy arrays.

9.1.4.1 License

Mahotas is available free of charge and free of restrictions: https://github.com/luispedro/mahotas/blob/master/COPYING.

9.1.4.2 References

- **Luis Pedro Coelho** Mahotas: Open source software for scriptable computer vision in Journal of Open Research Software, vol 1, 2013 (https://dx.doi.org/10.5334/jors.ac).

9.1.4.3 Documentation

Website: https://github.com/luispedro/mahotas

Current documentation: https://mahotas.readthedocs.io/en/latest/index.html

9.1.4.4 Installation

To install `Mahotas` using `conda`, use:

```
$ conda install mahotas -c conda-forge
```

To install `Mahotas` using `pip`, use:

```
$ pip install mahotas
```