Spikes Data Sciences

Fabien Campillo

Sep 15, 2025

CONTENTS

This Jupyter Book is designed to help you get started in data science by exploring electrophysiological data, especially spike train recordings, in a practical and accessible way.

Even though we are mainly interested in processing electrophysiology measurements such as spikes, we will attempt an overview of neuroscience resources.

We will focus on electrophysiology data processing and distinguish between:

- M/EEG data, non-invasive/extracranial,

- and invasive data at the single neuron level or from a population of neurons, notably using MEA (Multi-Electrode Array).

It is this second category that is of most interest to us (MathNeuro). The first category is very well developed. Processing extracranial electrophysiological data (EEG/MEG) is generally more complex than processing intracranial measurements (spikes, LFP, ECoG). In intracranial recordings, electrodes are close to neurons: the signal is more localized, with a better signal-to-noise ratio, which facilitates the identification of action potentials or local fields. In contrast, extracranial signals are heavily attenuated, resulting from the summation of millions of neurons and distorted by cranial tissues. They are also contaminated by numerous artifacts. Analysis therefore requires advanced processing (filtering, correction, modeling) and solving the inverse problem (retrieving brain sources from incomplete and ambiguous measurements, which is a mathematically ill-posed problem).

This Jupyter Book is part of the Data Science Bootcamp for MathNeuro and is made openly accessible to the broader community.

This Jupyter book https://fabien-campillo.github.io/data-science-spikes/ The GitHub repository https://github.com/fabien-campillo/data-science-spikes The PDF version of this book.

*Several parts of this book, including sections of the Markdown content and Python source code, were generated or refined with the assistance of **ChatGPT-4**, which also provided guidance on building this Jupyter Book. Some of the prompts used with ChatGPT are preserved as comments in the Markdown cells, providing a peek into the questions and guidance that shaped the content. While this tool was helpful in drafting and organizing content, all remaining errors and final decisions are entirely my own.*

RESOURCES AND REFERENCES

I've put together some resources and references using Python (but keep in mind, R is another popular route into data science).

1.1 Some Python packages

First, I list some indispensable Python libraries used in data science. In addition to core Python, you should also start getting familiar with a few other tools:

- Python's classics:
    - NumPy – numerical computing and array manipulation.
    - SciPy – scientific computing and statistics.
    - Matplotlib – basic plotting library.
- Data Manipulation:
    - Pandas – data structures and analysis tools.
- Statistical Analysis:
    - statsmodels – estimation of statistical models, statistical tests, and data exploration.
- Machine Learning:
    - Scikit-learn – widely used machine learning library.
- Natural Language Processing (NLP):
    - NLTK – platform for working with human language data.
    - SpaCy – main library for NLP tasks.
    - Gensim – topic modeling library.
- Data Visualization:
    - Seaborn – statistical data visualization.
    - Plotly – interactive graphing library.
- Web Scraping:
    - BeautifulSoup – extracting data from HTML files.

1.2 Resources and references for general data sciences

1.2.1 Books

- **An Introduction to Statistical Learning with Applications in Python**, Springer 2023, by Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani, and Jonathan Taylor. See Book Homepage and Resources with the PDF and the associated Youtube videos with Trevor Hastie & Jonathan Taylor (and it starts with Trevor complimenting Jonathan on his new haircut, why not…). Trevor Hastie is one of the big dudes in statistics (see his book "The Elements of Statistical Learning: Data Mining, Inference, and Prediction"), and Jonathan Taylor is a younger statistician with a nice new haircut.

- **Python for Data Analysis** (3rd ed.), O'Reilly 2022, by Wes McKinney a creator of Panda. The open edition is avalaible, with the codes, see his GitHub for other resources.

- **Python Data Science Handbook** (2nd ed.), O'Reilly 2022, by Jake VanderPlas – full text, and the associated Jupyter Notebook (very nice!), see his GitHub for other resources.

- **Practical Statistics for Data Scientists: 50+ Essential Concepts Using R and Python** (2nd ed.), O'Reilly 2020, by Peter Bruce, Andrew Bruce, Peter Gedeck. See the GitHub with the Python codes and notebooks.

- **Python for Probability, Statistics, and Machine Learning** (3rd ed.), Springer 2022, by José Unpingco. See his GitHub for other resources.

- **Hands-On Machine Learning with Scikit-Learn, Keras & Tensorflow** (3rd ed.), O'Reilly 2022, by Aurélien Géron, and the associated notebooks, see his GitHub for other resources. Machine learning and deep Learning.

- **Deep Learning Illustrated**, Addison-Wesley 2019, by Jon Krohn, with the associated notebooks, the concept if quite interesting. See also this github.

I do not provide references on the basic mathematical foundations of data science, which usually include linear algebra, calculus (with a focus on optimization), probability theory, statistics (both elementary and inferential), discrete mathematics (graphs, combinatorics, logic), and sometimes numerical methods. I also do not include general references on statistics, machine learning, or Python programming itself, as well as topics related to databases such as SQL, relational database design, and NoSQL systems. There are numerous high-quality resources available for all these areas.

1.2.2 Jupyter (note)books

Among the previous references:

- Jake VanderPlas' Python Data Science Handbook

- Aurélien Géron's notebooks, *a series of Jupyter notebooks that walk you through the fundamentals of Machine Learning and Deep Learning in Python using Scikit-Learn, Keras and TensorFlow 2.*

- Wes McKinney's "Python for Data Analysis" open edition and notebooks.

1.3 Resources and references for data sciences in neurosciences

1.3.1 References

- **Python in Neuroscience**, E. Muller, J. A. Bednar, M. Diesmann, M.-O. Gewaltig, M. Hines, and A. P. Davison Frontiers in Neuroinformatics, 9, 2015.

- **Case Studies in Neural Data Analysis**, 2016 - The book presents MATLAB tools, but there is an associated GitHub repository for Python. The book primarily covers extracranial data, except Chapter 8: *Basic Visualizations and Descriptive Statistics of SpikeTrainData*.

- **Neural Data Science** (2020–23), Aaron J. Newman from the NeuroCognitive Imaging Lab (Dalhousie University, Halifax).
  Starts from scratch, especially in Python. Includes a section on Single Unit Data. See the GitHub repository for the Jupyter Book and the YouTube channel Neural Data Science with Python.

- **Neural Data Science: A Primer with MATLAB and Python**, Erik Lee Nylen and Pascal Wallisch, 2017. See the table of contents.

1.3.2 Spike Train and Electrophysiology Data Analysis

Python packages

- **syncopy** - Systems Neuroscience Computing in Python: a Python package for large-scale analysis of electrophysiological data, with the following article.

- **MNE** - Open-source Python package for exploring, visualizing, and analyzing human neurophysiological data: MEG, EEG, sEEG, ECoG, NIRS, and more).

- **pynapple** – Python Neural Analysis Package. Pynapple is a lightweight Python library for neurophysiological data analysis. See the article: Pynapple, a toolbox for data analysis in neuroscience, 2023.

- **osl-ephys** - This package contains models for analysing electrophysiology data. It builds on top of the widely used MNE-Python package and contains analysis tools for M/EEG sensor and source space analysis. From the Oxford Centre for Human Brain Activity Analysis Group, with this GitHub repository and this 2025 paper: osl-ephys: a Python toolbox for the analysis of electrophysiology data.

- **NeuralEnsemble** – a community-based initiative to promote and coordinate open-source software development in neuroscience. **Inactive since 2022.**

Jupyter (note)book(s)

- **Spike sorting the 'Do It Yourself' way** a Jupyter book by Christophe Pouzat with the gitlab repository. See also the Probabilistic Spiking Neuronal Nets: Companion associated with tge book Probabilistic Spiking Neuronal Nets co-authored with Antonio Galves and Eva Löcherbach.

1.3.3 Blog(s) and blog posts.

- Spikes and Bursts — an interesting blog by David Cabrera-Garcia, where he explores various concepts. He also runs a YouTube channel and shares projects on GitHub. An interesting post:

  - Patch-clamp data analysis in Python: animate time series data.

- Patch clamp electrophysiology analysis with Python (2023) by Vincenzo Mastrolia

1.3.4 Misc.

- **ElecFeX** - A MATLAB-based Electrophysiological Feature eXtraction toolbox for single-cell intracellular recordings. See the article: *ElecFeX is a user-friendly toolbox for efficient feature extraction from single-cell electrophysiological recordings*

1.3.5 Other tools

Before analyzing data, we first need to read electrophysiology recordings and handle the different standards used.

- The pyABF library was created by Scott Harden. We will return to that package in a future section.

DATA STORAGE, SHARING, AND DATABASES FOR ELECTROPHYSIOLOGY

2.1 Databases

2.1.1 Main databases in electrophysiology

By the way, data really matter! First you can look at the Wikipedia List of neuroscience databases, and at among many:

- **NLM Dataset Catalog** - A catalog of biomedical datasets from various repositories, NIH National Library of Medecine.

- **Dryad** - An open data publishing platform and a community committed to the open availability and routine re-use of all research data. Examples of data sets:

  - a dataset presenting electrophysiological properties from whole-cell patch clamped nucleus accumbens core medium spiny neurons from male rats and female rats recorded in different estrous cycle phases.

  - a dataset containing whole-cell electrophysiological recordings (patch-clamp recordings) from three cell types in mice.

- **Dandi Archive** - The BRAIN Initiative archive for publishing and sharing neurophysiology data including electrophysiology, optophysiology, and behavioral time-series, and images from immunostaining experiments. Example of data sets:

  - Patch-seq recordings from mouse visual cortex. Whole-cell Patch-seq recordings from neurons of the mouse visual cortex from the Allen Institute for Brain Science, released in June 2020. The majority of cells in this dataset are GABAergic interneurons, but there are also a small number of glutamatergic neurons from layer 2/3 of the mouse visual cortex.

- **zenodo.org** - An open-access research data repository built and hosted by CERN (the European Organization for Nuclear Research), with support from the European Commission. Example of data sets:

  - a data set of CA1 pyramidal cell recordings using an intact whole hippocampus preparation, including recordings of rebound firing (V2).

2.1.2 Focus on NLM, Dryad, Dandi, and Zenodo

NLM Dataset Catalog (NIH/NLM)

- The NLM Dataset Catalog is essentially a registry pointing to datasets, not a direct host.

- You'll usually get redirected to the actual host (sometimes PhysioNet, OpenNeuro, or institutional repositories).

- If the dataset is downloadable via URL, you can use requests or wget in Python to fetch the .abf files, then open with pyabf.

- No standard Python API for the catalog itself, but you can scrape metadata via their JSON endpoints (if provided per dataset).

Dryad

- Dryad datasets are hosted at *datadryad.org*.

- They expose a REST API (https://datadryad.org/api/v2/).

- You can query a DOI, get file download links, and then download with requests.

Example:

```python
import requests
import pyabf

doi = "10.5061/dryad.xxxxx"  # replace with dataset DOI
r = requests.get(f"https://datadryad.org/api/v2/datasets/{doi}")
files = r.json()["included"]
for f in files:
    if f["attributes"]["filename"].endswith(".abf"):
        url = f["attributes"]["downloadUrl"]
        abf_data = requests.get(url)
        with open(f["attributes"]["filename"], "wb") as fh:
            fh.write(abf_data.content)
        abf = pyabf.ABF(f["attributes"]["filename"])
```

DANDI Archive

- DANDI hosts primarily neurophysiology data in NWB (Neurodata Without Borders) format, not ABF.

- They have a Python client: `dandi`.

Example to download:

```
pip install dandi
dandi download DANDI:000003  # replace with dataset identifier
```

If the dataset includes `.abf` files (rare, usually `NWB`), you can load them directly with `pyabf`. More commonly, you'd work with NWB using `pynwb`, not `pyabf`.

Zenodo

- Zenodo provides a REST API: https://zenodo.org/api/.

- Each record has a DOI and you can list associated files.

Example:

```python
import requests
import pyabf

record_id = "1234567"  # Zenodo record ID
r = requests.get(f"https://zenodo.org/api/records/{record_id}")
for f in r.json()["files"]:
    if f["key"].endswith(".abf"):
        url = f["links"]["self"]
        abf_data = requests.get(url)
        with open(f["key"], "wb") as fh:
            fh.write(abf_data.content)
        abf = pyabf.ABF(f["key"])
```

2.2 Storing, reading, and sharing electrophysiology records

Electrophysiology records can be cumbersome and often require specific formats to be quickly accessible and usable. Traditionally, these formats were dependent on the devices used to collect the data—proprietary formats. It soon became clear that this approach was slowing down scientific progress, which relies on making data easily accessible to the widest possible community. To address this, new formats were developed.

2.3 Types of Electrophysiology Recordings

Electrophysiology covers a wide range of recording techniques, each suited to different biological questions. Here are the main categories:

- **Intracellular Recordings** - *Sharp electrode recordings*: measure the membrane potential inside a single cell • *Whole-cell patch clamp*: provides detailed access to ionic currents, membrane potential, and synaptic inputs • *Single-channel recordings*: resolve the activity of individual ion channels.

- **Extracellular Recordings** - *Single-unit recordings*: detect action potentials ("spikes") from individual neurons using fine electrodes • *Multi-unit recordings*: capture spikes from small groups of neurons near the electrode tip • *Multi-electrode arrays (MEA)*: record from dozens to thousands of electrodes simultaneously across a neural population.

- **Field Potential Recordings** - *Local field potentials (LFPs)*: measure summed synaptic activity and slower fluctuations in a local region • *ECoG (electrocorticography)*: records field potentials directly from the cortical surface.

- **Non-Invasive Recordings** - *EEG (electroencephalography)*: scalp recordings of brain activity with high temporal resolution • *MEG (magnetoencephalography)*: detects magnetic fields generated by neuronal currents.

- **Other Specialized Methods** - *EMG (electromyography)*: records muscle activity • *ERG (electroretinography)*: records retinal responses to light • *Patch-clamp in slices/in vivo*: advanced combinations allowing intracellular access in complex preparations.

2.4 Common Electrophysiology Data Formats

For-mat	Full Name	Typical Use	Open-ness	Notes
ABF	Axon Binary File (Molecular Devices)	Patch-clamp and intracellular recordings (pCLAMP, Clampex)	Pro-pri-etary	Very common in cellular electrophysiology; ABF1 (older), ABF2 (newer).
IBW	Igor Binary Wave (Igor Pro)	Waveform storage and analysis	Pro-pri-etary	Widely used in physiology labs with Igor Pro; supports multidimensional data.
HEKA	Patchmaster (DAT/PGF/PUF)	Patch-clamp recordings (HEKA amplifiers)	Pro-pri-etary	Popular in Europe; rich metadata but tied to vendor software.
CED SON	Cambridge Electronic Design (Spike2)	Extracellular recordings, spike trains	Pro-pri-etary	Common for multi-electrode recordings and stimulus protocols.
WCP	WinWCP File (Strath-clyde)	Patch-clamp and voltage-clamp recordings	Semi-open	Free Windows-only software; popular in teaching and some labs; less standardized than ABF/HEKA.
NWB	Neurodata Without Borders	Sharing neurophysiology data (spikes, LFPs, imaging)	**Open standard**	Community-driven, based on HDF5; widely promoted for reproducibility and FAIR data.
BIDS-iEEG	Brain Imaging Data Structure (intracranial EEG extension)	EEG, ECoG, iEEG	**Open standard**	Growing adoption; integrates with neuroimaging standards.
MAT	MATLAB File (.mat)	Custom lab storage/analysis	Semi-open	Extremely common, but not standardized for sharing.
HDF5	Hierarchical Data Format 5	Large, structured datasets	Open	Flexible backbone format; NWB is built on HDF5.
EDF/	European Data Format / BioSemi Data Format	EEG, PSG, clinical neuro-physiology	Open	Standard in sleep studies and EEG; supported by many clinical systems.
CSV/T	Comma-/Tab-separated Values	Generic export of time series or metadata	Open	Universally readable, but loses rich metadata and structure.
WAV	Waveform Audio File (Windows)	Audio-like exports of signals/spike trains	Open	Native Windows format; sometimes used for stimulus or simplified data export.

2.5 Most Used Electrophysiology Data Formats

- **ABF (Axon Binary File)** - ABF is a proprietary binary format developed by **Molecular Devices**. It is widely used for **patch-clamp and intracellular recordings**, especially in cellular electrophysiology research. ABF files store both the raw electrophysiological signals and metadata, such as sampling rate, stimulus protocols, and experimental parameters. Versions include ABF1 (older) and ABF2 (newer).

- **NWB (Neurodata Without Borders)** - NWB is an **open, community-driven standard** designed for **sharing and long-term storage of neurophysiology data**. It is based on HDF5 and supports raw signals, metadata, experimental design, stimuli, and derived data. NWB promotes **reproducibility and FAIR principles**, and is increasingly adopted by labs worldwide.

- **IBW (Igor Binary Wave)** - IBW is the binary wave format used by **Igor Pro** software. It is popular for **waveform storage and analysis**, particularly in physiology labs. IBW supports multi-dimensional data, annotations, and is

often part of custom analysis pipelines. Despite being proprietary, it remains widely used because of historical and workflow reasons.

2.5.1 Summary Table

Format	Developer	Proprietary / Open	Free to use?	Main Use	Adoption
ABF (Axon Binary File)	Originally Axon Instruments → now **Molecular Devices**	**Propri-etary** (specs partially available, but tied to pCLAMP)	Reading libraries are free (e.g., `pyABF`), but **creating/using ABF files requires pCLAMP**, which is commercial	Electrophysiology (patch-clamp, voltage/current recordings)	Widely used in labs with Molecular Devices equipment
IBW (Igor Binary Wave)	**WaveMetrics, Inc.**	**Proprietary** (closed format, documentation partly available)	Requires **Igor Pro** (commercial software). Some third-party libraries exist to read IBW for free	General scientific data analysis and visualization	Popular in biophysics, neuroscience, spectroscopy
NWB (Neurodata Without Borders)	Community-driven (Allen Institute, HHMI Janelia, Berkeley Lab, INCF, etc.)	**Fully open-source** and community standard	**Free** (developed and maintained openly on GitHub)	Standardized storage/sharing of neuroscience data (e-phys, imaging, behavioral, metadata)	Growing adoption in large-scale projects, especially for data sharing and FAIR science

2.6 Exploring `abf` files

This first notebook aims to demonstrate how to analyze electrophysiological recordings from a single cell by:

- Extracting basic characteristics of the recorded signals.
- Plotting, for each recording trial (sweep, see below), the evolution of the membrane potential (voltage) and, if applicable, the injected current.

We import a few libraries:

```
import numpy as np
import matplotlib.pyplot as plt
import pyabf                    # load the pyABF library (cf. infra)
import seaborn as sns           # used for a Kernel Density Estimate (KDE) plot
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[1], line 1
----> 1 import numpy as np
      2 import matplotlib.pyplot as plt
      3 import pyabf                    # load the pyABF library (cf. infra)

ModuleNotFoundError: No module named 'numpy'
```

We focus on specific cell, the cell # 89:

```
file_path = "lhb_bursting/cell89basal.abf"
print('path to the abf file : ',file_path)
```

```
path to the abf file :  lhb_bursting/cell89basal.abf
```

2.6.1 `abf` files and `pyabf` library

ABF File Overview

An ABF (Axon Binary Format) file is a proprietary file format developed by Axon Instruments (now part of Molecular Devices) to store electrophysiological data from experiments. ABF files are commonly used to save data from experiments like patch-clamp recordings, where researchers measure electrical signals from biological systems (such as neurons or muscle cells). These files can store a variety of information, including:

- Data Traces: Time series data for one or more channels, representing signals such as voltage or current.
- Metadata: Information about the experiment, including settings for the recording, such as sampling rate, experiment type, and device configuration.
- Multiple Sweeps: An ABF file can contain multiple sweeps (individual trials or experimental runs), which may differ in parameters or conditions.

The ABF format is binary, making it efficient for large datasets, but it is not easily readable without specialized software or libraries.

`pyabf` Library

The `pyabf` library is a Python package designed to facilitate working with ABF files. It provides an easy-to-use interface to read and manipulate data stored in ABF files. The library makes it simpler for researchers to extract relevant information from ABF files, without having to manually parse the binary data.

Key Features of pyabf:

1. Load ABF Files: Load an ABF file into memory and provide access to its data.
2. Access Data Traces: Extract time-series data, such as voltage and current traces (from ADC channels).
3. Multiple Sweep Support: Handle multiple sweeps (individual experimental runs) within a single ABF file.
4. Extract Metadata: Retrieve metadata like channel names, experiment parameters, and other settings.
5. Sweep Navigation: Select and navigate through multiple sweeps (trials) and analyze their data individually.

Common Functions in pyabf:

- `ABF(file_path)`: Initializes an ABF object from a given file path, loading the data into memory.
- `setSweep(sweep_index)`: Selects a specific sweep (experimental run) by its index.
- `sweepY`: Extracts the voltage (or other signal) data for the current sweep.
- `sweepX`: Extracts the time vector for the current sweep.
- `sweepC`: Extracts the command input (if available) for the current sweep.
- `adcNames`: List of ADC channel names.
- `dacNames`: List of DAC channel names.

On the net

The pyABF library was created by Scott Harden. Scott Harden has made pyABF available as an open-source library, aiming to simplify the process of working with ABF files in Python, making it easier for researchers to analyze and visualize their data. You can find more about pyABF and its documentation on his website: pyABF - A simple Python interface for Axon Binary Format ABF files.

- pyABF - A simple Python interface for Axon Binary Format ABF files, with git repository

- a tutorial by [Scott W Harden]

- in Python Package Index pypi

2.6.2 Basics about `abf` objects

Where we import the `pyabf` package and load a record file:

```
abf = pyabf.ABF(file_path)              # we load it
print(abf)                              # record characteristics
```

```
ABF (v2.6) with 1 channel (pA), sampled at 10.0 kHz, containing 30 sweeps, having
↪no tags, with a total length of 11.11 minutes, recorded without a protocol file.
```

Here `abf = pyabf.ABF(file_path)` creates an `abf` object that have:

- **attributes**: data stored in the object, and

- **methods**: functions that belong to an object and can be called to perform actions

Atributes

We can print more attributes:

```
print(f"{'File Path:':>20} {abf.abfFilePath}")
print(f"{'File Version:':>20} {abf.abfVersionString}")
print(f"{'Sampling Rate:':>20} {abf.dataRate} Hz")
print(f"{'Total Sweeps:':>20} {abf.sweepCount}")
print(f"{'ADC Channels:':>20} {abf.adcNames}")
print(f"{'DAC Channels:':>20} {abf.dacNames}")
print(f"{'Channel Units:':>20} {abf.sweepUnitsY}")
print(f"{'Experiment Date:':>20} {abf.abfDateTime}")
```

```
        File Path: /Users/campillo/Documents/1-now/_git.nosync/pepyna/data/lhb_
↪bursting/cell89basal.abf
     File Version: 2.6.0.0
    Sampling Rate: 10000 Hz
     Total Sweeps: 30
     ADC Channels: ['Waveform']
     DAC Channels: ['AO #0']
    Channel Units: pA
  Experiment Date: 2024-11-08 00:00:50.086000
```

Methods

You can list all the methods of an `abf` object with `print(abf.__dict__)`.

```python
methods = [method for method in dir(abf) if callable(getattr(abf, method)) and not
 ↪method.startswith("__")]
print("\n".join(methods))
```

```
_dtype
_getAdcNameAndUnits
_getDacNameAndUnits
_ide_helper
_loadAndScaleData
_makeAdditionalVariables
_readHeadersV1
_readHeadersV2
getAllXs
getAllYs
headerLaunch
launchInClampFit
saveABF1
setSweep
sweepD
```

You have private methods (Prefixed with _), and:

- `getAllXs()`: Returns all time points (X-values) for every sweep, useful for plotting.

- `getAllYs()`: Returns all recorded signal values (Y-values) for every sweep.

- `headerLaunch()`: Likely a utility function for debugging or inspecting header information.

- `launchInClampFit()`: Opens the ABF file in ClampFit, a software from Molecular Devices used for electrophysiology data analysis.

- `saveABF1()`: Converts and saves the ABF file in version 1 format, which is older but sometimes required for compatibility.

- `setSweep(sweepIndex)`: Sets the current sweep (i.e., trial or recording segment) to a given index for further processing.

- `sweepD`: Likely an attribute or method that provides the time duration of a sweep.

Of course the main parts of the sweep are the **recorded signal** and the **command inpput**:

```python
# Print voltage trace (recorded signal)
print(f"{'Voltage Trace (mV):':>25} {abf.sweepY}")

# Print command input (if available)
print(f"{'Command Input (mV):':>25} {abf.sweepC}")
```

```
        Voltage Trace (mV): [-61.2   -60.856 -61.2   ... -56.228 -56.196 -56.353]
        Command Input (mV): [0. 0. 0. ... 0. 0. 0.]
```

2.6.3 Basic `abf` file exploration

Main `abf` attributes and methods

The `abf` object contains various attributes and methods that allow you to access metadata and data from the .abf file. Here are some useful attributes and how to call them:

```
print("List of sweep indexes:", ", ".join(map(str, abf.sweepList)))
```

```
    List of sweep indexes: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,␣
    ↪17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29
```

```
sweep_index = 0  # Choose a specific sweep (e.g., first sweep -> index 0)
abf.setSweep(sweep_index)

# Print voltage trace (recorded signal)
print(f"{'Voltage Trace (mV):':>25} {abf.sweepY}")

# Print command input (if available)
print(f"{'Command Input (mV):':>25} {abf.sweepC}")

# Check all available ADC channels (recorded signals)
print(f"{'Recorded Channels:':>25} {abf.adcNames}")

# Check DAC channels (command input signals)
print(f"{'Command Channels:':>25} {abf.dacNames}")
```

```
        Voltage Trace (mV): [-61.2   -60.856 -61.2   ... -56.228 -56.196 -56.353]
        Command Input (mV): [0. 0. 0. ... 0. 0. 0.]
         Recorded Channels: ['Waveform']
          Command Channels: ['AO #0']
```

Some statistics of 1 sweep

Here, we have selected `sweep_index = 0`, representing the first sweep in the ABF file. We then compute some basic statistics of the corresponding voltage trace, such as the mean, median, min, max, standard deviation, and range of the signal:

```
data = abf.sweepY  # The voltage trace for the specific swwep

stats = {
    "Mean (mV)": np.mean(data),
    "Median (mV)": np.median(data),
    "Min (mV)": np.min(data),
    "Max (mV)": np.max(data),
    "Std Dev (mV)": np.std(data),
    "Range (mV)": np.ptp(data),  # Max - Min
}

print("\nVoltage Trace Statistics:")
for key, value in stats.items():
    print(f"{key:>20}: {value:.3f}")
```

```
Voltage Trace Statistics:
        Mean (mV): -55.656
      Median (mV): -56.853
         Min (mV): -66.204
         Max (mV): 49.191
     Std Dev (mV): 9.172
       Range (mV): 115.395
```

2.6.4 Basics plottings

**Plotting the voltage trace distribution**

```python
# Create a figure with two subplots (1 row, 2 columns)
fig, axes = plt.subplots(1, 2, figsize=(10, 4))

# Plot histogram on the first subplot
axes[0].hist(data, bins=50, edgecolor='black', alpha=0.7)
axes[0].set_xlabel("Voltage (mV)")
axes[0].set_ylabel("Frequency")
axes[0].set_title("Voltage Trace Distribution")
axes[0].grid(True)

# Plot KDE on the second subplot
sns.kdeplot(data, bw_adjust=0.5, fill=True, color="b", alpha=0.5, ax=axes[1])
axes[1].set_xlabel("Voltage (mV)")
axes[1].set_ylabel("Density")
axes[1].set_title("Voltage Trace Density")
axes[1].grid(True)

# Adjust layout
plt.tight_layout()
plt.show()
```
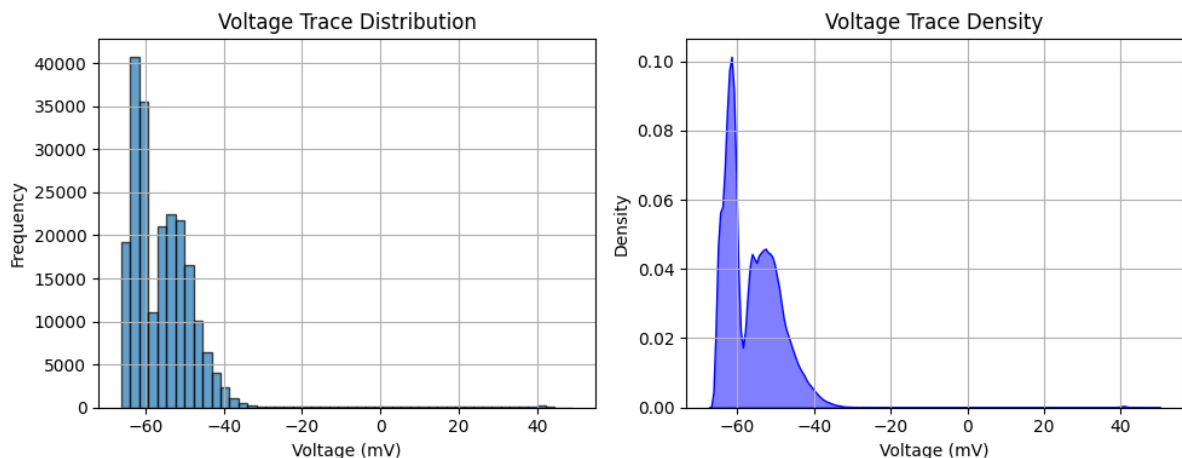

Of course, the previous plots are not particularly interesting, as they represent the distribution of the signal while mixing both inter-burst and burst phases. A more insightful approach would be to plot the distributions separately: one for the inter-burst signal and another for the intra-burst signal. Note that the previous plots exhibit a bimodal distribution, which is certainly related to the two phases of the signal—burst and non-burst (quiescent or background activity).
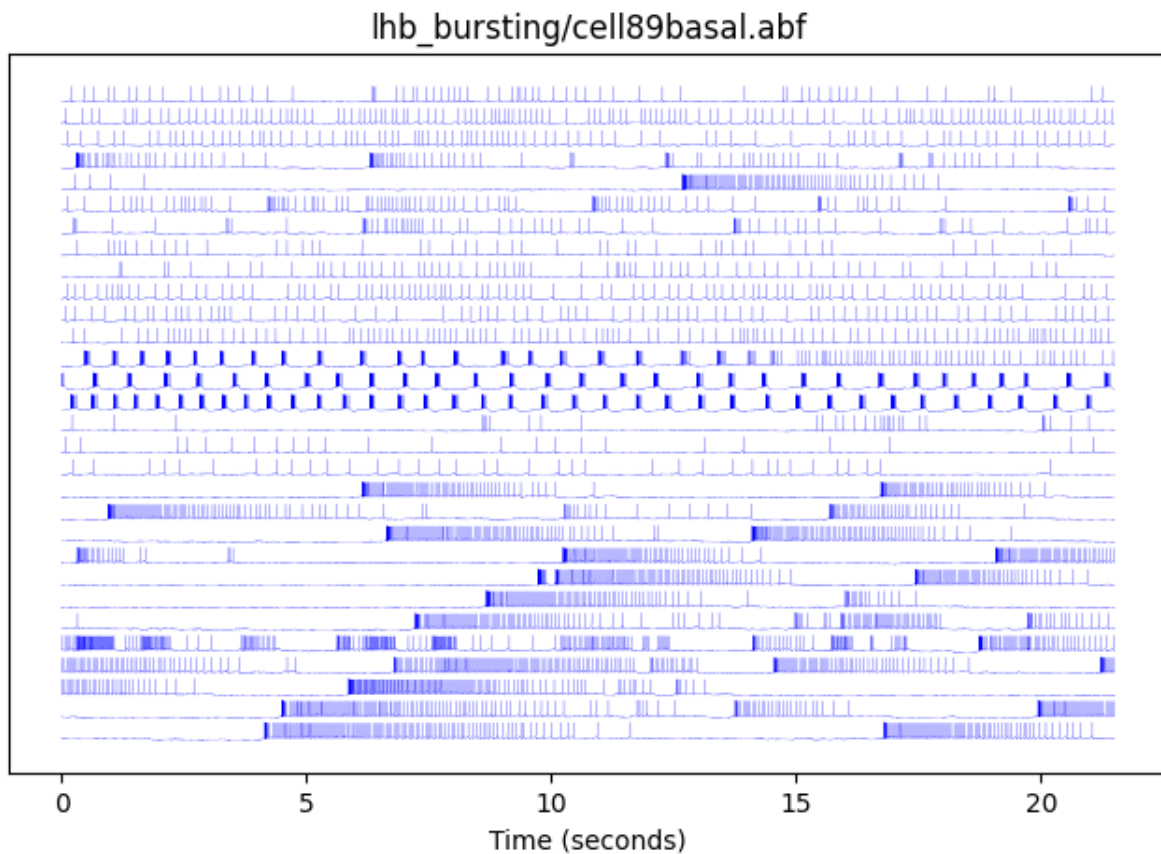
Plotting all the sweeps of an `abf` file

```python
def plot_abf():
    plt.figure(figsize=(8, 5))

    # plot every sweep (with vertical offset)
    for sweepNumber in abf.sweepList:
        abf.setSweep(sweepNumber)
        offset = 140*sweepNumber
        plt.plot(abf.sweepX, abf.sweepY+offset, color='b',lw=0.1)

    # decorate the plot
    plt.gca().get_yaxis().set_visible(False)   # hide Y axis
    plt.title(file_path)
    plt.xlabel(abf.sweepLabelX)
    plt.show()
```

```python
plot_abf()
```


The same but we also plote the input voltage on each sweep: here it's not so interesting because thay are null (not recorded ?):

```python
def plot_abf():
    plt.figure(figsize=(8, 5))

    # Plot every sweep with vertical offset
```

```
    for sweepNumber in abf.sweepList:
        abf.setSweep(sweepNumber)
        offset = 140 * sweepNumber
        plt.plot(abf.sweepX, abf.sweepY + offset, color='b', lw=0.1)  # Recorded␣
↪voltage
        plt.plot(abf.sweepX, abf.sweepC + offset, color='r', lw=0.5)  # Command␣
↪waveform in red

    # Decorate the plot
    plt.gca().get_yaxis().set_visible(False)  # Hide Y axis
    plt.title(file_path)
    plt.xlabel(abf.sweepLabelX)
    plt.show()

plot_abf()
```


lhb_bursting/cell89basal.abf
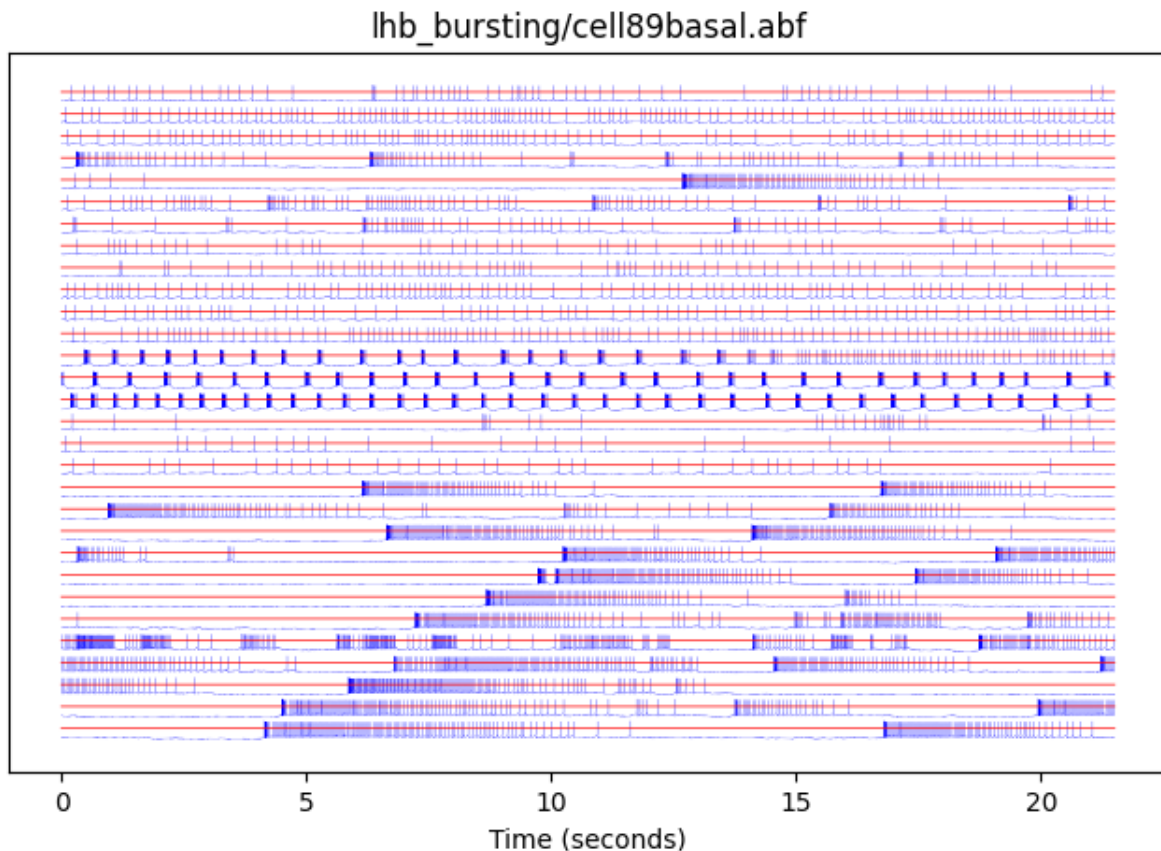
Plotting a single sweep

```
# Create the figure
fig, ax1 = plt.subplots(figsize=(8, 5))

# Plot the recorded curve (ADC) on the left axis
ax1.plot(abf.sweepX, abf.sweepY, color='b', lw=0.2, label="ADC waveform")
ax1.set_xlabel(abf.sweepLabelX)
ax1.set_ylabel(abf.sweepLabelY, color='b')
ax1.tick_params(axis='y', labelcolor='b')

# Create a second y-axis for the control curve (DAC)
ax2 = ax1.twinx()
ax2.plot(abf.sweepX, abf.sweepC, color='r', lw=0.2,label="DAC waveform")
ax2.set_ylabel(abf.sweepLabelC, color='r')
ax2.tick_params(axis='y', labelcolor='r')

# Improve the layout
fig.tight_layout()
plt.show()
```
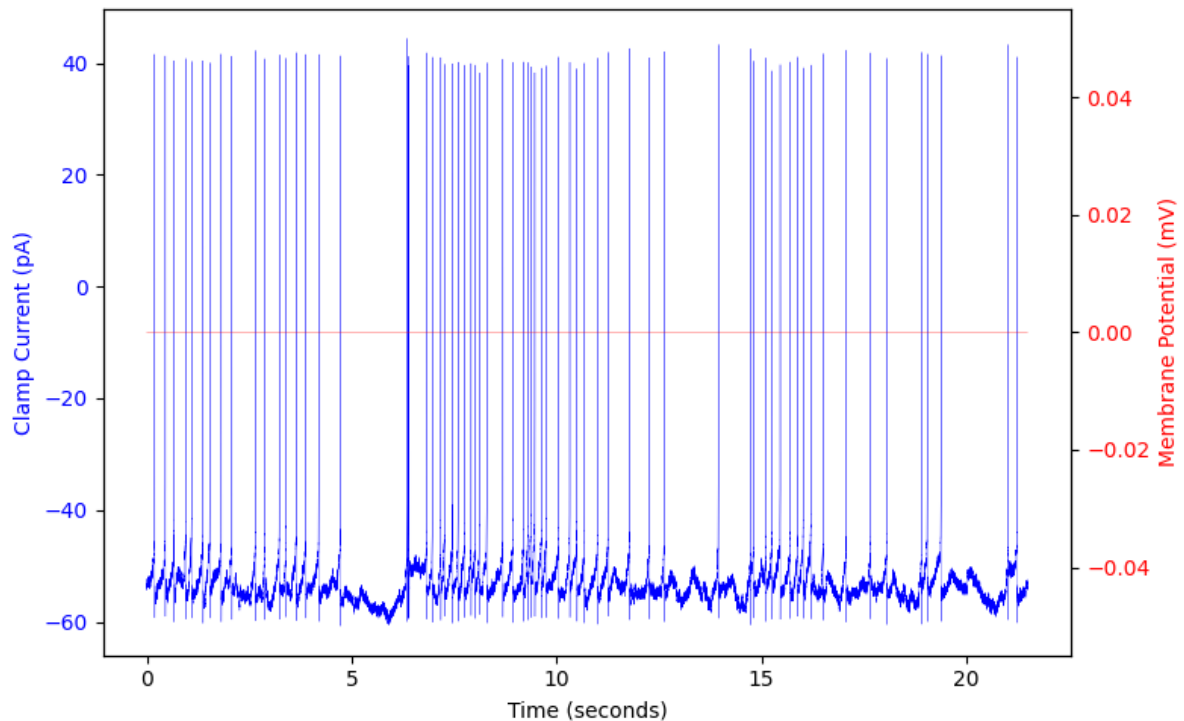
2.6.5 Data to explore

We define a list of tuples called `data_to_explore`. Each element in `data_to_explore` is a tuple containing:

- a string = the file path of an `.abf` file,

- An integer = a sweep number associated with the file,

representing the cell records and the specific number of the sweep we want to explore.

```python
data_to_explore = []
data_to_explore.append(("lhb_bursting/cell21basal.abf",   5))
data_to_explore.append(("lhb_bursting/cell89basal.abf",  15))
data_to_explore.append(("lhb_bursting/cell91basal.abf",   4))
data_to_explore.append(("lhb_bursting/cell98basal.abf",   9))
data_to_explore.append(("lhb_bursting/cell104basal.abf", 10))
data_to_explore.append(("lhb_bursting/cell105basal.abf",  7))
data_to_explore.append(("lhb_bursting/cell107basal.abf",  0))
data_to_explore.append(("lhb_bursting/cell209basal.abf",  4))

for abf_file, sweep_number in data_to_explore:
    print(f"ABF File: {abf_file:<30} {'Sweep Number:':>15} {sweep_number:>3}")
```

```
ABF File: lhb_bursting/cell21basal.abf    Sweep Number:   5
ABF File: lhb_bursting/cell89basal.abf    Sweep Number:  15
ABF File: lhb_bursting/cell91basal.abf    Sweep Number:   4
ABF File: lhb_bursting/cell98basal.abf    Sweep Number:   9
ABF File: lhb_bursting/cell104basal.abf   Sweep Number:  10
ABF File: lhb_bursting/cell105basal.abf   Sweep Number:   7
ABF File: lhb_bursting/cell107basal.abf   Sweep Number:   0
ABF File: lhb_bursting/cell209basal.abf   Sweep Number:   4
```

```python
# Créer la figure
plt.figure(figsize=(10, 6))

# Tracer chaque fichier/sweep avec un décalage vertical
for i, (file_path, sweep_num) in enumerate(data_to_explore):
    abf = pyabf.ABF(file_path)
    abf.setSweep(sweep_num)
    offset = i * 150  # décalage vertical entre les tracés
    plt.plot(abf.sweepX, abf.sweepY + offset, lw=0.3, label=f"{file_path} [sweep
 ↪{sweep_num}]")

# Mise en forme
plt.gca().get_yaxis().set_visible(False)  # cacher l'axe Y
plt.xlabel("Temps (s)")
plt.title("Tracés en tension pour différents fichiers/sweeps")
plt.legend(fontsize="xx-small", loc="upper right")
plt.tight_layout()
plt.show()
```

Tracés en tension pour différents fichiers/sweeps
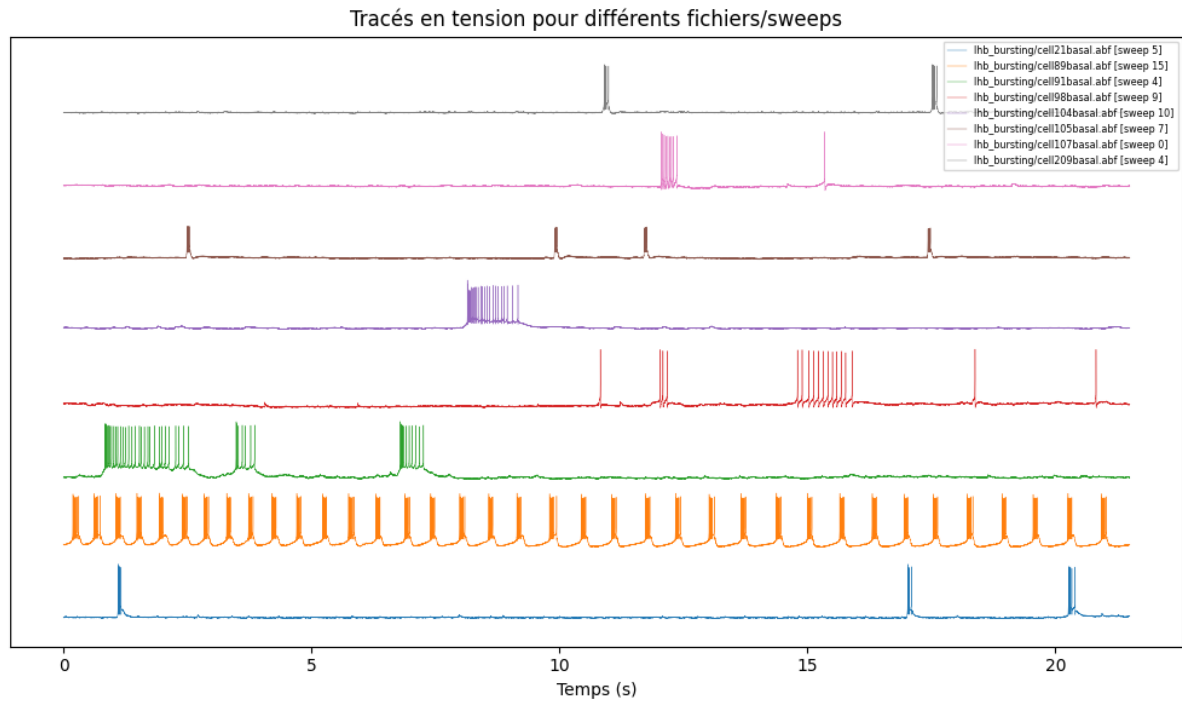
APPENDIX A: INSTALLING PYTHON AND SOME TOOLS

3.1 Main Python distributions for data sciences

When starting with Python for data science, it's important to know the main distributions you can use. These distributions include Python itself, plus tools to manage packages and environments. Here's an overview that works across macOS, Linux, and Windows.

3.1.1 System Python

- Many operating systems come with **Python pre-installed**:
    - macOS and most Linux distributions include Python.
    - Windows does not come with Python pre-installed (you need to download it from Python.org).
- Usually an older version (e.g., Python 3.8 or 3.9).
- Good for simple scripts, but installing additional packages may conflict with system tools.

3.1.2 Official Python from Python.org

- The official Python distribution is available at python.org.
- Works on **macOS, Linux, and Windows**.
- You can manually install any additional data science packages (e.g., `numpy`, `pandas`, `matplotlib`) using `pip`.
- Lightweight and cross-platform, but you need to manage dependencies yourself.

3.1.3 Anaconda

- A **full Python distribution for scientific computing and data science**.
- Includes:
    - Python itself
    - Hundreds of pre-installed libraries (numpy, pandas, matplotlib, scipy, etc.)
    - Jupyter Notebook / JupyterLab
- Works on **macOS, Linux, and Windows**.
- Large download (~3 GB), but everything is ready-to-use.

- Good choice if you want a complete environment for data science without installing each library manually.

3.1.4 Miniconda

- A **minimal version of Anaconda**, including only Python + the `conda` package manager.

- You install only the packages you need.

- Works on **macOS, Linux, and Windows**.

- Lightweight, flexible, and suitable for reproducible environments.

- Often preferred for creating isolated Python environments per project.

3.1.5 Platform-Specific Package Managers (optional)

- **macOS:** Homebrew can install Python (`brew install python@3.13`).

- **Linux:** System package managers like `apt` (Debian/Ubuntu) or `dnf/yum` (Fedora/CentOS) can install Python.

- **Windows:** Chocolatey can install Python (`choco install python`) if you prefer command-line installation.

⚠ These install **system-wide Python**, not isolated environments, so careful with package conflicts.

⚠ Mac users: Homebrew is a great tool to install system software on macOS, but it's generally **not recommended to use brew-installed Python for data science projects**. Why? Because brew installs Python system-wide, which can **conflict with project-specific environments** like conda or venv. For isolated, reproducible Python environments, prefer **Miniconda or Anaconda** instead.

3.1.6 Summary Table

Distribution	Platforms	Main Feature	Notes
System Python	macOS/Linux	Pre-installed	Might be old; not isolated
Python.org	macOS/Linux/Windows	Official Python	Lightweight; manual package management
Anaconda	macOS/Linux/Windows	Full scientific stack	Large; ready-to-use
Miniconda	macOS/Linux/Windows	Minimal + conda	Lightweight; flexible
Homebrew / apt / dnf / Chocolatey	macOS/Linux/Windows	System package manager	Installs Python and other software system-wide; not isolated

This gives you a clear overview of the **main Python distributions you can use for data science**, regardless of your operating system. Installation instructions and environment setup can be covered later.

3.2 Installing Anaconda and Miniconda

3.2.1 Installing Anaconda

macOS: Go to the Anaconda Downloads page, download the macOS installer (Graphical or command-line), open the `.pkg` file, and follow the instructions. Open a terminal and verify the installation:

```
conda --version
```

Linux: Download the Linux installer from Anaconda Downloads. Open a terminal and run:

```
bash ~/Downloads/Anaconda3-<version>-Linux-x86_64.sh
```

Follow the prompts to complete the installation. Verify with:

```
conda --version
```

Windows: Download the Windows installer from Anaconda Downloads. Run the `.exe` file and follow the instructions. Open **Anaconda Prompt** or **PowerShell** and verify:

```
conda --version
```

3.2.2 Installing Miniconda

macOS: Go to the Miniconda Downloads page, download the macOS installer, open the `.pkg` file, and follow the instructions. Open a terminal and verify:

```
conda --version
```

Linux: Download the Linux installer from Miniconda Downloads. Open a terminal and run:

```
bash ~/Downloads/Miniconda3-latest-Linux-x86_64.sh
```

Follow the prompts to complete the installation. Verify with:

```
conda --version
```

Windows: Download the Windows installer from Miniconda Downloads. Run the `.exe` file and follow the instructions. Open **Anaconda Prompt** or **PowerShell** and verify:

```
conda --version
```

3.2.3 Tips and Notes

- Optionally add conda to your PATH during installation to use it from any terminal.

- Update conda after installation:

```
conda update conda
```

- Miniconda is recommended for a lightweight setup.

- Usage of conda environments and package installation will be covered in later sections.

3.2.4 Summary Table

Step	macOS	Linux	Windows
Download installer	Anaconda / Miniconda	Same	Same
Run installer	`.pkg`	`bash ~/Downloads/ installer.sh`	`.exe`
Verify installation	`conda --version`	`conda --version`	`conda --version`
Notes	Graphical or CLI installer	Terminal-based	Use Anaconda Prompt or PowerShell

3.3 Conda

Conda is a package manager for Python and other languages. It helps you install packages and manage dependencies easily.

3.3.1 Basics

```
# check that it is correctly installed:
conda --version

# keep Conda up-to-date with:
conda update conda

# install a package (Replace `numpy` with the desired package name):
conda install numpy

# install a specific version:
conda install numpy=1.25

# install multiple packages at once:
conda install numpy pandas matplotlib

# updating packages:
conda update numpy

# removing packages:
conda remove numpy

# searching for packages(replace `package_name` with the
# name of the package you want to find):
conda search package_name
```

3.3.2 Conda tips

Update Conda regularly to get bug fixes and security updates.

If a package is not found, check alternative channels:

```
conda install -c conda-forge package_name
```

3.3.3 Summary of common Conda commands

Task	Command
Check Conda version	`conda --version`
Update Conda	`conda update conda`
Install package	`conda install package_name`
Install specific version	`conda install package_name=version`
Install multiple packages	`conda install package1 package2`
Update package	`conda update package_name`
Remove package	`conda remove package_name`
Search for package	`conda search package_name`
Install from channel	`conda install -c conda-forge package_name`

3.4 Conda virtual environment

3.4.1 Using Virtual Environments

Why Use a Virtual Environment in Python?

The problem without a virtual environment:

- By default, when you install a library with `pip install`, it goes into the **system-wide Python**.
- Risks:
  - ⚠ Version conflicts between projects (e.g., one project needs `numpy==1.20`, another `numpy==1.26`).
  - ⚠ Risk of breaking system tools that rely on Python (macOS and Homebrew depend on it).
  - ⚠ Environment quickly polluted with dozens of unnecessary packages.

Solution: Virtual Environments

A virtual environment = an **isolated copy of Python** with its own libraries.

Advantages:

- Project-by-project isolation.
- No conflicts between library versions.
- Easier to share and reproduce a project (`requirements.txt` or `environment.yml`).
- You can delete a project without polluting the system.

Two Main Choices: `venv` vs `conda`

`venv` (native Python virtual environments)

- Included in Python (`python -m venv myenv`).

- Lightweight, simple to use.

- Package management via `pip install`.

- Good for:

    – Lightweight projects (Flask, Django, scripts).

    – General development.

⚠ Limitations:

- `pip` installs only Python libraries.

- Some heavy libraries (numpy, scipy, torch, tensorflow…) may require compilation → possible errors.

`conda` (Anaconda/Miniconda environments)

- Also creates isolated environments (`conda create -n myenv python=3.10`).

- Can install not only Python libraries, but also **system dependencies** (BLAS, MKL, CUDA, etc.).

- Precompiled package distribution → fast and reliable installation.

- Good for:

    – Data science and machine learning (numpy, pandas, scikit-learn, PyTorch, TensorFlow).

    – Multi-language projects (Python + R + CUDA…).

⚠ Limitations:

- Heavier than `venv`.

- Package management can be slightly slower at times.

3.4.2 Conda Virtual Environments

Conda: Creating a New Environment

```
# create a new Conda environment with a specific Python version (Replace `myenv`
# with the name of your environment and `3.11` with the desired Python version)
conda create --name myenv python=3.11

# Activate the environment before working in it:
conda activate myenv

# When you are done, deactivate the environment to return to the base environment:
conda deactivate
```

Conda: Listing and Removing Environments

```
# list all available environments:
conda env list

# remove an environment completely:
conda remove --name myenv --all
```

Conda: Installing Packages in an Environment

```
# install a package in the active environment:
conda install numpy

# install a specific version of a package:
conda install numpy=1.25

# install multiple packages at once:
conda install numpy pandas matplotlib
```

Conda: Updating and Removing Packages

```
# update a package in the current environment:
conda update numpy

# remove a package from the environment:
conda remove numpy
```

Conda: Exporting and Reproducing Environments

```
# to share or reproduce an environment, export it to a YAML file:
conda env export > environment.yml

# create an environment from a YAML file:
conda env create -f environment.yml
```

3.4.3 Conda Tips

- Always use separate environments for different projects to avoid conflicts.

- Update Conda regularly with `conda update conda`.

- Use the `conda-forge` channel if a package is not found in the default channels:

```
conda install -c conda-forge package_name
```

Summary Table of Common Conda Environment Commands

Task	Command
Create environment	`conda create --name myenv python=3.11`
Activate environment	`conda activate myenv`
Deactivate environment	`conda deactivate`
List environments	`conda env list`
Remove environment	`conda remove --name myenv --all`
Install package	`conda install package_name`
Install specific version	`conda install package_name=version`
Install multiple packages	`conda install package1 package2`
Update package	`conda update package_name`
Remove package	`conda remove package_name`
Export environment	`conda env export > environment.yml`
Create from file	`conda env create -f environment.yml`