DATA SCIENCE & SPIKES

Fabien Campillo

2025

Data Sciences and Spikes

Fabien Campillo

Oct 03, 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).

Important

This notebook relies on Python packages such as `numpy`, `matplotlib`, `pyabf`, `seaborn`, and others. To ensure reproducibility and avoid conflicts with other Python projects, it is **strongly recommended to use a dedicated virtual environment**. For detailed instructions on setting up the environment, see the beginning of Chapter *Exploring records with pyabf*.

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

*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.*

Part I

Content

CHAPTER

ONE

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

Math books

- The contributions of **Robert E. Kass** are noteworthy. Rob Kass is a renowned statistician, and he has also contributed to the modeling and statistical analysis of Neural Spike Train Data, and to machine learning. One can refer to his book Analysis of Neural Data, which is actually an excellent introductory book on probability and statistics through the lens of neural data. His page Contributions to Analysis of Neural Spike Train Data also provides an overview of his contributions to the field.
- **Analysis of Parallel Spike Trains** edited by S. Grün and S. Rotter (Springer, 2010).
- **Stochastic Models for Spike Trains of Single Neurons** by G. Sampath and S. K. Srinivasan

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.
- **Elephant - Electrophysiology Analysis Toolkit** is an emerging open-source, community centered library for the analysis of electrophysiological data in the Python programming language. Elephant focuses on generic analysis functions for spike train data and time series recordings from electrodes GitHub repository
- **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.

1.4 Sometimes we don't even know what we're talking about

> Data science, statistics, math, machine learning—sure, they're all great when applied to modeling and analyzing spikes and bursts. But let's not forget: we also need to paddle upstream to the very source of those signals. Where do the spikes and bursts records come from? The experimental lab. And what do they actually represent? The wild and real dynamics of real neurons.

- **Guide to Research Techniques in Neuroscience** by Matt Carter, Rachel Essner, Nitsan Goldstein, and Manasi Iyer (2022, 3rd Edition)
- **Electrophysiological Recording Techniques** edited by Robert P. Vertes and Timothy Allen (Springer, 2022).
- **Introduction to Electrophysiological Methods and Instrumentation** by Franklin Bretschneider and Jan R. de Weille (Academic Press, Second edition, 2019).
- **Basic Electrophysiological Methods** edited by Matt Carter and Ellen Covey (Oxford University Press, 2015)
- **The Laboratory Computer: A Practical Guide for Physiologists and Neuroscientists** by John Dempster (Academic Press, 2001).

CHAPTER

TWO

ELECTROPHYSIOLOGY DATA

Accessing electrophysiology records can be difficult, and working with them is often cumbersome, as they typically require specific formats to be quickly accessible and usable. To make data more widely available, it is crucial to develop not only open-access databases but also standardized file formats. Historically, data formats were tied to the devices that generated them—often proprietary and incompatible with other systems. This fragmentation soon became a barrier to scientific progress. In what follows, we first introduce common file formats, then present several relevant databases, and finally show how to work with them in Python.

2.1 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.

2.2 Common electrophysiology data formats

Format	Full Name	Typical Use	Openness	Notes
ABF	Axon Binary File (Molecular Devices)	Patch-clamp and intracellular recordings (pCLAMP, Clampex)	Proprietary	Very common in cellular electrophysiology; ABF1 (older), ABF2 (newer).
IBW	Igor Binary Wave (Igor Pro)	Waveform storage and analysis	Proprietary	Widely used in physiology labs with Igor Pro; supports multidimensional data.
HEKA	Patchmaster (DAT/PGF/PUF)	Patch-clamp recordings (HEKA amplifiers)	Proprietary	Popular in Europe; rich metadata but tied to vendor software.
CED SON	Cambridge Electronic Design (Spike2)	Extracellular recordings, spike trains	Proprietary	Common for multi-electrode recordings and stimulus protocols.
WCP	WinWCP File (Strathclyde)	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/I	European Data Format / BioSemi Data Format	EEG, PSG, clinical neurophysiology	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.

2.3 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.

2.3.1 Summary table

Format	Developer	Proprietary / Open	Free to use?	Main Use	Adoption
ABF (Axon Binary File)	Originally Axon Instruments → now **Molecular Devices**	**Proprietary** (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
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
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

2.3.2 Companies and formats

Company	HQ	Main Products (Neuronal EP)	Native Data Formats	Conversion / Notes
Molecular Devices (Axon Instruments)	USA	Patch-clamp amplifiers (Axopatch, MultiClamp), digitizers (Digidata), pCLAMP software	**ABF (Axon Binary File)**: ABF1 (legacy), ABF2 (current)	Widely used in patch-clamp. Readable with AxoGraph, Clampfit. Python: `pyABF`, `Neo`. Export to **NWB** possible.
HEKA Elektronik (Harvard Bioscience)	Germany	Patch-clamp amplifiers (EPC series), Patchmaster software	**.dat / .pgf / .pul**	Proprietary binary; support in **Neo**. Conversion pipelines exist for **NWB/NIX**.
Multi Channel Systems (MCS, Harvard Bioscience)	Germany	Multi-electrode arrays (MEAs), in vitro & in vivo systems	**.mcd** (proprietary), **.h5** (newer systems)	SDK/API for reading `.mcd`. `.h5` is HDF5 and integrates more easily. Neo + NWB supported.
Blackrock Neurotech	USA	Utah arrays, high-density neural recording for BCI	**NSx** (continuous), **NEV** (spike/event)	Openly documented. MATLAB, Python APIs. Supported in **Neo**. Standard in BCI research.
Neuralynx	USA	In vivo extracellular recording systems	**NCS, NSE, NEV, etc.**	ASCII headers + binary. Readers available (MATLAB, Python). Supported in **Neo**, can export to NWB.
Ripple Neuro	USA	High-channel neural recording & stimulation systems (BCI, clinical)	Similar to Blackrock: **NSx / NEV**	MATLAB/Python SDK. Actively supports **NWB**.
Tucker-Davis Technologies (TDT)	USA	High-throughput recording, optogenetics, stimulation	**.tsq / .tev / .sev**	Proprietary binary. TDT SDK + Neo support. NWB export possible.
Intan Technologies	USA	Low-power amplifier chips & headstages (RHD, RHS series)	**.rhd / .rhs**	Binary with header metadata. Intan provides readers (C++, MATLAB, Python). Neo + NWB supported. Widely used in open-source rigs.
ADInstruments	New Zeala	PowerLab DAQ, LabChart software, teaching/research physiology tools	**.adicht** (LabChart files)	Proprietary but supported by LabChart & APIs. Neo support available. Some pipelines to NWB.
Kerr Scientific Instruments	New Zeala	In vitro slice & tissue electrophysiology rigs	Exports via DAQ hardware (often LabChart, NI, etc.)	Less standardized — depends on DAQ choice (often integrates with **ADInstruments**).

2.4 Databases

By the way, data really matter! You can look at the Wikipedia list of neuroscience databases. We will consider four of them.

2.4.1 NLM Dataset Catalog

NLM Dataset Catalog is a catalog of biomedical datasets from various repositories, NIH National Library of Medecine (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).

2.4.2 Dryad

Dryad (DAta Archive for Neuroscience DIscovery) is an open data publishing platform and a community committed to the open availability and routine re-use of all research data.

- 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.

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.

Example of usage:

```
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"])
```

2.4.3 Dandi Archive

Dandi Archive (Distributed Archive for Neuroscience Data Integration) is the BRAIN Initiative archive for publishing and sharing neurophysiology data including electrophysiology, optophysiology, and behavioral time-series, and images from immunostaining experiments.

- DANDI hosts primarily neurophysiology data in NWB (Neurodata Without Borders) format (link, not ABF.
- They have a Python client: `dandi` (link).

An example of data set:

- 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.

Example of usage:

```
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`.

2.4.4 Zenodo

zenodo.org (named after Zenodotus, the first librarian of Alexandria) is an open-access research data repository built and hosted by CERN (the European Organization for Nuclear Research), with support from the European Commission.

- Zenodo provides a REST API: https://zenodo.org/api/.
- Each record has a DOI and you can list associated files.

Example of data sets:

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

Example:

```
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"])
```

2.5 Gateways with Python

Formats commonly used in electrophysiology (ABF, NBW, IBW) are not always straightforward to handle. Fortunately, several Python libraries act as gateways to read and convert:

- **PyNWB** is a Python package for working with NWB files (doc github).
- **pyabf** is a Python library for reading electrophysiology data from Axon Binary Format (ABF) files (github), see also abf explorer, a simple graphical application for quickly viewing axon binary format (ABF).
- **IBW** (Igor Binary Wave) is a Python parser for IBW (.ibw) and Packed Experiment (.pxp) files written by WaveMetrics' IGOR Pro software (see igor2). IBW is a bit less active in Python.

As we already had a quick peep at above, several Python tools are available for working with records from these databases, see Chapter *Exploring records with pyabf* for more details. Also in Chapter *Exploring a database with csv* we present some tools and ideas to explore a database.

CHAPTER

THREE

EXPLORING A DATABASE WITH CSV

Important

Usually, directories of experimental records are produced by colleagues in the lab. It is essential that there is ongoing discussion between the experimentalists and the people who will later analyze the data, such as data scientists. If data scientists modify the structure of the records directory, it becomes much harder to maintain this communication and can significantly slow down the process of data exploitation and analysis. Therefore, following good practices for organizing and managing the data is crucial.

See Section "*csv in the context of panda and csvkit*" for an introduction to `csv` in the context with `panda` and `csvkit`.

3.1 Building a csvfile

We consider a dataset in `data/records_fake`.

This dataset consists of a set of (empty) records compiled in 2024, from March 15 to June 26, by Joanna Danielewicz at the Mathematical, Computational and Experimental lab, headed by Serafim Rodrigues at BCAM.

The directory structure is the same as in the real dataset, but the files are empty, since the original recordings are too large. Later, we will work with selected real recordings.

In the `data` directory I have a subdirectory `records_fake` that contains empty `abf` and `atf` files, the structure is:

```
!tree -L 2 data/data_bcam_2024/records
```

```
data/data_bcam_2024/records
├── 03.15
│   ├── C1
│   └── C2
├── 03.20
│   └── C1
├── 04.10
│   ├── C1
│   ├── C2\ immature
│   └── C3
├── 05.07
│   ├── C1\ DG
│   └── C2\ DG
├── 05.20
│   ├── C1
│   ├── C2
│   ├── C3
│   ├── C4
```

(continues on next page)

(continued from previous page)

```
│   └── C5
├── 05.23
│   ├── C1
│   ├── C2
│   ├── C3
│   └── C4
├── 05.29
│   ├── C1
│   ├── C2
│   └── C3\ immature
├── 05.30
│   ├── C1
│   └── C2
├── 06.19
│   ├── C1
│   └── C2
├── 06.26
│   ├── C1
│   ├── C2
│   └── C3
└── README.md

37 directories, 1 file
```

First we present somes good practices about records directory.

3.2 Good practrices about records directory

3.2.1 Directory structure: flatten or not?

- Keep the hierarchy (date → cell → files).
- It mirrors the experimental workflow (date of recording, then cell).
- Easier to reason about provenance ("what did we do on May 30?").
- Helps you separate sessions and avoid accidental filename collisions.
- Flattening could be useful for scripts, but you can always create virtual flattening in Python (e.g., by walking the directory tree). So I'd keep the hierarchy and let code do the flattening.

Recommendation: Keep the directory hierarchy. Use scripts to index/flatten when needed.

3.2.2 Metadata handling

- Do not rely only on filenames — they're fragile and inconsistent.
- Best practice: keep a `metadata` table (CSV, TSV, JSON, YAML, or SQLite DB), eg. a `metadata.csv` to store all experiment details.
  - You can add or correct metadata later without touching raw files.
  - Easy to query and filter in Python (e.g., with `pandas`).
  - Prevents the need to rename files whenever metadata changes.

Recommendation: Maintain a single central metadata file (CSV for simplicity, or SQLite if the dataset grows large).

3.2.3 `README.md`

Add a `README.md` in the data file that includes:

- Directory structure
- Naming conventions
- Protocol/temperature shorthand
- Instructions for using metadata

3.2.4 File Naming

- Keep original filenames from acquisition (ground truth).
- Do not overwrite raw files.
- If you want clean names, create symbolic links or derived copies like: `cellID_date_protocol.abf`, eg. you can make a (flat) dir of symbolic links:

```
symlinks/
├── C21_05.30.abf
├── C22_05.30.abf
├── C23_06.19.abf
```

3.2.5 Keep raw data read-only

- Protects raw data integrity → accidental edits, renames, or deletions won't happen.
- Clear separation between:
  - Raw data (immutable, read-only)
  - Derived data / analysis results (reproducible, regeneratable)
- Works well with the principle: "never touch your raw data, always derive."
- In multi-user setups (lab server, shared cluster), permissions protect against colleagues accidentally overwriting.

Things to keep in mind

- You may still want to add new files (new experiments) → so you don't want to lock the whole records root permanently. Instead, set read-only permissions per experiment once it's finalized.
- If you ever need to move or reorganize, you'll have to re-enable write permissions (chmod u+w).
- A safer alternative:
  - Keep raw records as read-only
  - Mirror it to a version-controlled metadata database (metadata.csv or SQLite), which you can edit freely

3.3 Back to the case study

To make everything in 'data/records_fake/' read-only:

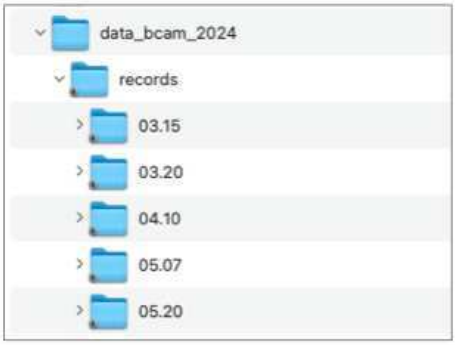
```
!chmod -R a-w data/data_bcam_2024/records
!ls -l data/data_bcam_2024/records/*/*/* | head -n 5
```

```
-r--r--r--  1 campillo  staff  0  3 oct 20:59 data/data_bcam_2024/records/03.15/
 ↪C1/2024_03_15_0000 IC steps 23.abf
-r--r--r--  1 campillo  staff  0  3 oct 20:59 data/data_bcam_2024/records/03.15/
 ↪C1/2024_03_15_0001 IC ramp 23.abf
-r--r--r--  1 campillo  staff  0  3 oct 20:59 data/data_bcam_2024/records/03.15/
 ↪C1/2024_03_15_0002 IC sin 23.abf
-r--r--r--  1 campillo  staff  0  3 oct 20:59 data/data_bcam_2024/records/03.15/
 ↪C1/2024_03_15_0003 VC ramp 23.abf
-r--r--r--  1 campillo  staff  0  3 oct 20:59 data/data_bcam_2024/records/03.15/
 ↪C1/2024_03_15_0004 IC ramp 25.abf
ls: stdout: Undefined error: 0
```

Now everybody (owner, group, all) car read the files but cannot write (or execute), files can still be read and copied. Still under MacOS with `Finder` you can make some damages, you can make them immutable:

```
!chflags -R uchg data/data_bcam_2024/records
```

Note the little lockers in Finder:



(To undo: `chflags -R nouchg records`)

> **Tip**
>
> Best practices
>
> - Keep raw data untouched.
> - Keep raw data read-only once experiments are finalized.
> - Track metadata in metadata.csv — symlinks are just for easier file access, not metadata storage.
> - If necessary, use symlinks for convenience (clean naming, flat access).

3.4 Pandas DataFrame

pandas DataFrame (df) is far better than built-in Python tools like lists, dictionaries, or arrays for the following reasons:

- Tabular structure built-in
  - A DataFrame behaves like a table or spreadsheet: rows = records, columns = fields.
  - You don't have to manually manage parallel lists or nested dictionaries.
- Easy indexing and filtering: doing the same with lists/dicts would require loops and conditionals — much more verbose
- Powerful aggregation & grouping: compute counts, averages, sums, or custom statistics without writing loops

- Built-in handling of missing data
  - Pandas understands `NaN` values automatically.
  - Built-in functions handle missing data gracefully.
- Integration with plotting and analysis
- Easy I/O; Load/save CSV, Excel, SQL, JSON, and more with a single command.

See infra.

3.5 Back to the case study: the boring job !

This part is as necessary as it is boring ! From directory `data/data_bcam_2024/records` (which we will not modify), we generate a metadata file `ddata/data_bcam_2024/records_metadata.csv` in CSV format. The procedure is somewhat tricky and was developed **step by step** with the help of ChatGPT.

The final python script (not very informative) is `data/data_bcam_2024/create_metadata.py`.

```
!cd data/data_bcam_2024 && python3 create_metadata.py
metadata_file = "data/data_bcam_2024/records_metadata_clean.csv"
```

```
Step 1: columns cell_id, file_path, file_name
Step 2: column date
Step 3: column exp_nb
Step 4: column comments
Step 5: column protocol
Step 6: column prot-opt
Step 7: column tp
```

```
Step 8: refine comments
Step 9: separate bad records
Final CSV saved to :
   records_metadata.csv        that contains references to all records
   records_metadata_clean.csv  that contains references to all clean records
   records_metadata_bad.csv    that contains references to all bad records
   records_metadata.csv =  records_metadata_clean.csv + records_metadata_bad.csv
```

```
import pandas as pd
df = pd.read_csv(metadata_file) # load the metadata
df.head(1000)                   # display the first few rows
```

```
     exp_nb cell_id        date protocol prot-opt    tp comments  \
0         1      C1  2024-03-15       IC    steps  23.0       NaN
1         2      C1  2024-03-15       IC     ramp  23.0       NaN
2         3      C1  2024-03-15       IC      sin  23.0       NaN
3         4      C1  2024-03-15       VC     ramp  23.0       NaN
4         5      C1  2024-03-15       IC     ramp  25.0       NaN
..      ...     ...         ...      ...      ...   ...       ...
639     655     C27  2024-06-26       IC     ramp  34.0       NaN
640     656     C27  2024-06-26       IC      sin  34.0       NaN
641     657     C27  2024-06-26       VC     ramp  34.0       NaN
642     658     C27  2024-06-26       IC     ramp  37.0       NaN
643     659     C27  2024-06-26       VC     ramp  37.0       NaN

                                  file_path                          file_name
0    03.15/C1/2024_03_15_0000 IC steps 23.abf  2024_03_15_0000 IC steps 23.abf
1     03.15/C1/2024_03_15_0001 IC ramp 23.abf   2024_03_15_0001 IC ramp 23.abf
2      03.15/C1/2024_03_15_0002 IC sin 23.abf    2024_03_15_0002 IC sin 23.abf
3     03.15/C1/2024_03_15_0003 VC ramp 23.abf   2024_03_15_0003 VC ramp 23.abf
```

(continues on next page)

(continued from previous page)

```
4     03.15/C1/2024_03_15_0004 IC ramp 25.abf   2024_03_15_0004 IC ramp 25.abf
..                                          ...                             ...
639   06.26/C3/2024_06_26_0044 IC ramp 34.abf   2024_06_26_0044 IC ramp 34.abf
640    06.26/C3/2024_06_26_0045 IC sin 34.abf    2024_06_26_0045 IC sin 34.abf
641   06.26/C3/2024_06_26_0046 VC ramp 34.abf   2024_06_26_0046 VC ramp 34.abf
642   06.26/C3/2024_06_26_0047 IC ramp 37.abf   2024_06_26_0047 IC ramp 37.abf
643   06.26/C3/2024_06_26_0048 VC ramp 37.abf   2024_06_26_0048 VC ramp 37.abf

[644 rows x 9 columns]
```

We end up with:

- a complete consolidated CSV (`records_metadata.csv`) with all metadata columns for downstream analysis
- the subset of "clean" records references (`records_metadata_clean.csv`)
- the subset of "bad" records references (`records_metadata_bad.csv`)

and `records_metadata.csv` = `records_metadata_clean.csv` ∪ `records_metadata_bad.csv`

3.6 Exploring the metadata

3.7 First we read the csv and create a dataframe object:

The `.info()` method prints a concise summary of the DataFrame. Here's what it shows:

- Index range → e.g. RangeIndex: 100 entries, 0 to 99
- Number of columns and their names
- Column data types (e.g. int64, float64, object for strings, datetime64, etc.)
- Number of non-null values per column (useful for spotting missing data)
- Memory usage of the DataFrame

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 644 entries, 0 to 643
Data columns (total 9 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   exp_nb     644 non-null    int64
 1   cell_id    644 non-null    object
 2   date       644 non-null    object
 3   protocol   372 non-null    object
 4   prot-opt   344 non-null    object
 5   tp         635 non-null    float64
 6   comments   19 non-null     object
 7   file_path  644 non-null    object
 8   file_name  644 non-null    object
dtypes: float64(1), int64(1), object(7)
memory usage: 45.4+ KB
```

```
import pandas as pd

# Load the metadata
```

(continues on next page)

(continued from previous page)

```
df = pd.read_csv(metadata_file)
# Display the first few rows
df.head()
```

```
   exp_nb cell_id        date protocol prot-opt    tp comments  \
0       1      C1  2024-03-15       IC    steps  23.0      NaN
1       2      C1  2024-03-15       IC     ramp  23.0      NaN
2       3      C1  2024-03-15       IC      sin  23.0      NaN
3       4      C1  2024-03-15       VC     ramp  23.0      NaN
4       5      C1  2024-03-15       IC     ramp  25.0      NaN

                                file_path                         file_name
0  03.15/C1/2024_03_15_0000 IC steps 23.abf  2024_03_15_0000 IC steps 23.abf
1   03.15/C1/2024_03_15_0001 IC ramp 23.abf   2024_03_15_0001 IC ramp 23.abf
2    03.15/C1/2024_03_15_0002 IC sin 23.abf    2024_03_15_0002 IC sin 23.abf
3   03.15/C1/2024_03_15_0003 VC ramp 23.abf   2024_03_15_0003 VC ramp 23.abf
4   03.15/C1/2024_03_15_0004 IC ramp 25.abf   2024_03_15_0004 IC ramp 25.abf
```

```
# Show unique protocols nicely
print("Unique protocols:", df['protocol'].unique())
# Count files per protocol
protocol_counts = df['protocol'].value_counts()
print("\nFiles per protocol:")
print(protocol_counts.to_string())
# Count files per day
day_counts = df['date'].value_counts()
print("\nFiles per day:")
print(day_counts.to_string())
```

```
Unique protocols: ['IC' 'VC' 'DC' nan]

Files per protocol:
protocol
IC    213
VC    130
DC     29

Files per day:
date
2024-05-23    126
2024-05-20     96
2024-05-07     70
2024-05-30     68
2024-03-15     67
2024-05-29     63
2024-04-10     61
2024-06-26     49
2024-06-19     24
2024-03-20     20
```

We can also use pandas styling (works in Jupyter Notebook):

```
# Count files per protocol
df['protocol'].value_counts().sort_index().to_frame().style.set_caption("Files␣
↪per Protocol").format("{:.0f}")
```

```
<pandas.io.formats.style.Styler at 0x128772f50>
```

```
# Count files per day
```

(continues on next page)

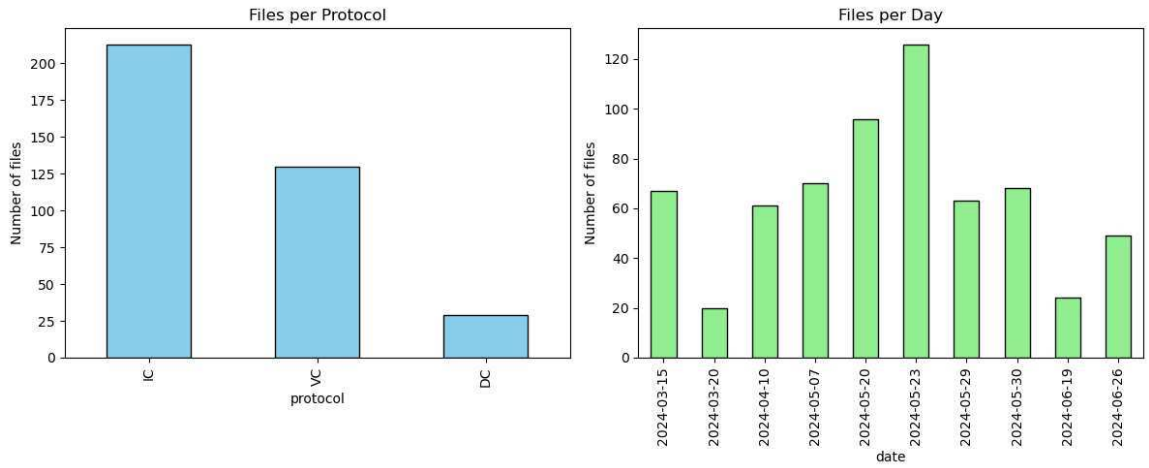
(continued from previous page)

```
df['date'].value_counts().sort_index().to_frame().style.set_caption("Files per Day
↪").format("{:.0f}")
```

```
<pandas.io.formats.style.Styler at 0x128783a10>
```

Now we propose a snippet that generates two side-by-side bar charts from the `df` dataframe: The result is a quick visual summary of how your dataset is distributed by protocol type and by date. Do you want me to show you how to make the x-axis labels more readable (e.g. rotating dates so they don't overlap)?

```
import matplotlib.pyplot as plt

# Create subplots: 1 row, 2 columns
fig, axes = plt.subplots(1, 2, figsize=(12, 5))  # adjust figsize as needed

# Files per protocol
protocol_counts = df['protocol'].value_counts()
protocol_counts.plot(
    kind='bar',
    color='skyblue',
    edgecolor='black',
    ax=axes[0],
    title='Files per Protocol'
)
axes[0].set_ylabel('Number of files')

# Files per day
day_counts = df['date'].value_counts().sort_index()
day_counts.plot(
    kind='bar',
    color='lightgreen',
    edgecolor='black',
    ax=axes[1],
    title='Files per Day'
)
axes[1].set_ylabel('Number of files')

plt.tight_layout()
plt.show()
```



3.8 Filtering

3.8.1 All dead cells experiments

```
# Filter rows where comments contain 'dead' (case-insensitive)
dead_cells = df[df['comments'].str.contains('dead', case=False, na=False)]

# Display the result
dead_cells
```

```
     exp_nb cell_id        date protocol prot-opt    tp comments  \
66       67      C2  2024-03-15       IC     ramp  38.0     dead
627     643     C26  2024-06-26       DC      NaN  25.0     dead

                                        file_path  \
66   03.15/C2/2024_03_15_0066 IC ramp 38 dead.abf
627       06.26/C2/2024_06_26_0032 25 DC dead.abf

                                file_name
66   2024_03_15_0066 IC ramp 38 dead.abf
627       2024_06_26_0032 25 DC dead.abf
```

3.8.2 All experiment of the day '2024-05-29'

```
# Filter rows for a specific day
day_data = df[df['date'] == '2024-05-29']

# Display the result
day_data
```

```
     exp_nb cell_id        date protocol prot-opt    tp  comments  \
440     449     C18  2024-05-29       IC     ramp  25.0       NaN
441     450     C18  2024-05-29       IC     ramp  28.0       NaN
442     451     C18  2024-05-29       IC     ramp  31.0       NaN
443     452     C18  2024-05-29       IC     ramp  34.0       NaN
444     453     C18  2024-05-29       IC     ramp  37.0       NaN
..      ...     ...         ...      ...      ...   ...       ...
498     507     C19  2024-05-29      NaN      NaN  32.0       NaN
499     508     C20  2024-05-29      NaN      NaN  33.0  immature
500     509     C20  2024-05-29      NaN      NaN  34.0  immature
501     510     C20  2024-05-29      NaN      NaN  35.0  immature
502     511     C20  2024-05-29      NaN      NaN  36.0  immature

                                 file_path                 file_name
440       05.29/C1/05.29 C1 IC ramp 25.atf  05.29 C1 IC ramp 25.atf
441       05.29/C1/05.29 C1 IC ramp 28.atf  05.29 C1 IC ramp 28.atf
442       05.29/C1/05.29 C1 IC ramp 31.atf  05.29 C1 IC ramp 31.atf
443       05.29/C1/05.29 C1 IC ramp 34.atf  05.29 C1 IC ramp 34.atf
444       05.29/C1/05.29 C1 IC ramp 37.atf  05.29 C1 IC ramp 37.atf
..                                     ...                       ...
498           05.29/C2/2024_05_29_0032.abf      2024_05_29_0032.abf
499  05.29/C3 immature/2024_05_29_0033.abf      2024_05_29_0033.abf
500  05.29/C3 immature/2024_05_29_0034.abf      2024_05_29_0034.abf
501  05.29/C3 immature/2024_05_29_0035.abf      2024_05_29_0035.abf
502  05.29/C3 immature/2024_05_29_0036.abf      2024_05_29_0036.abf

[63 rows x 9 columns]
```

3.8.3 All `IC ramp`

```
# Filter rows with protocol 'IC' and protocol_option 'ramp'
ic_ramp_cells = df[(df['protocol'] == 'IC') & (df['prot-opt'] == 'ramp')]

# Display the result
ic_ramp_cells
```

```
     exp_nb cell_id        date protocol prot-opt    tp comments  \
1         2      C1  2024-03-15       IC     ramp  23.0      NaN
4         5      C1  2024-03-15       IC     ramp  25.0      NaN
7         8      C1  2024-03-15       IC     ramp  27.0      NaN
10       11      C1  2024-03-15       IC     ramp  29.0      NaN
13       14      C1  2024-03-15       IC     ramp  31.0      NaN
..      ...     ...         ...      ...      ...   ...      ...
629     645     C27  2024-06-26       IC     ramp  25.0      NaN
634     650     C27  2024-06-26       IC     ramp  28.0      NaN
636     652     C27  2024-06-26       IC     ramp  31.0      NaN
639     655     C27  2024-06-26       IC     ramp  34.0      NaN
642     658     C27  2024-06-26       IC     ramp  37.0      NaN

                                  file_path                       file_name
1    03.15/C1/2024_03_15_0001 IC ramp 23.abf  2024_03_15_0001 IC ramp 23.abf
4    03.15/C1/2024_03_15_0004 IC ramp 25.abf  2024_03_15_0004 IC ramp 25.abf
7    03.15/C1/2024_03_15_0007 IC ramp 27.abf  2024_03_15_0007 IC ramp 27.abf
10   03.15/C1/2024_03_15_0010 IC ramp 29.abf  2024_03_15_0010 IC ramp 29.abf
13   03.15/C1/2024_03_15_0013 IC ramp 31.abf  2024_03_15_0013 IC ramp 31.abf
..                                       ...                             ...
629  06.26/C3/2024_06_26_0034 IC ramp 25.abf  2024_06_26_0034 IC ramp 25.abf
634  06.26/C3/2024_06_26_0039 IC ramp 28.abf  2024_06_26_0039 IC ramp 28.abf
636  06.26/C3/2024_06_26_0041 IC ramp 31.abf  2024_06_26_0041 IC ramp 31.abf
639  06.26/C3/2024_06_26_0044 IC ramp 34.abf  2024_06_26_0044 IC ramp 34.abf
642  06.26/C3/2024_06_26_0047 IC ramp 37.abf  2024_06_26_0047 IC ramp 37.abf

[136 rows x 9 columns]
```

3.9 `csvkit` the command-line Swiss Army knife

OK, notebooks are great, but when navigating through data directories, you may come across a CSV file—or, if you're less lucky, an Excel file. Don't panic: there's a handy tool for that: csvkit, it is a (nice) suite of command-line tools for converting to and working with CSV, the king of tabular file formats. See Section "*csvkit the command-line Swiss Army knife*" and/or the tutorial.

Look at column names:

```
!csvcut -n  data/data_bcam_2024/records_metadata_clean.csv
```

```
  1: exp_nb
  2: cell_id
  3: date
  4: protocol
  5: prot-opt
  6: tp
  7: comments
  8: file_path
  9: file_name
```

Pretty print of the (head of) columns 2,3,8:

```
!csvcut -c 2,3,8 data/data_bcam_2024/records_metadata_clean.csv | csvlook | head
```

```
| cell_id |       date | file_path                                      |
| ------- | ---------- | ---------------------------------------------- |
| C1      | 2024-03-15 | 03.15/C1/2024_03_15_0000 IC steps 23.abf       |
| C1      | 2024-03-15 | 03.15/C1/2024_03_15_0001 IC ramp 23.abf        |
| C1      | 2024-03-15 | 03.15/C1/2024_03_15_0002 IC sin 23.abf         |
| C1      | 2024-03-15 | 03.15/C1/2024_03_15_0003 VC ramp 23.abf        |
| C1      | 2024-03-15 | 03.15/C1/2024_03_15_0004 IC ramp 25.abf        |
| C1      | 2024-03-15 | 03.15/C1/2024_03_15_0005 IC sin 25.abf         |
| C1      | 2024-03-15 | 03.15/C1/2024_03_15_0006 VC ramp 25.abf        |
| C1      | 2024-03-15 | 03.15/C1/2024_03_15_0007 IC ramp 27.abf        |
```

Stats on “date” and “tp:

```
!csvcut -c date,tp data/data_bcam_2024/records_metadata_clean.csv | csvstat
```

```
  1. "date"

        Type of data:          Date
        Contains null values:  False
        Non-null values:       644
        Unique values:         10
        Smallest value:        2024-03-15
        Largest value:         2024-06-26
        Most common values:    2024-05-23 (126x)
                               2024-05-20 (96x)
                               2024-05-07 (70x)
                               2024-05-30 (68x)
                               2024-03-15 (67x)

  2. "tp"

        Type of data:          Number
        Contains null values:  True (excluded from calculations)
        Non-null values:       635
        Unique values:         73
        Smallest value:        0,
        Largest value:         69,
        Sum:                   18 044,5
        Mean:                  28,417
        Median:                28,
        StDev:                 12,181
        Most decimal places:   1
        Most common values:    25, (96x)
                               34, (66x)
                               31, (50x)
                               28, (45x)
                               40, (33x)

Row count: 644
```

Show a nice table of all files where “prot-opt” is steps, including only the “cell_id”, “protocol”, “prot-opt”, and “file_name” columns:

```
!csvcut -c cell_id,protocol,prot-opt,file_name data/data_bcam_2024/records_
↪metadata_clean.csv | csvgrep -c prot-opt -m steps | csvlook
```

```
| cell_id | protocol | prot-opt | file_name                              |
| ------- | -------- | -------- | -------------------------------------- |
| C1      | IC       | steps    | 2024_03_15_0000 IC steps 23.abf        |
```

(continues on next page)

(continued from previous page)

```
| C1      | IC       | steps    | 2024_03_15_0018 IC steps 34.abf        |
| C2      | IC       | steps    | 2024_03_15_0036 IC steps 23.abf        |
| C3      | IC       | steps    | 2024_03_20_0007 IC square steps 38.abf |
| C23     | IC       | steps    | 2024_06_19_0000 IC steps 25.abf        |
| C23     | IC       | steps    | 2024_06_19_0009 IC steps 34.abf        |
| C24     | IC       | steps    | 2024_06_19_0014 IC steps 25.abf        |
| C25     | IC       | steps    | 2024_06_26_0000 IC steps 25.abf        |
| C25     | IC       | steps    | 2024_06_26_0016 IC steps 34.abf        |
| C25     | IC       | steps    | 2024_06_26_0022 IC steps 40.abf        |
| C26     | IC       | steps    | 2024_06_26_0028 IC steps 25.abf        |
| C27     | IC       | steps    | 2024_06_26_0033 IC steps 25.abf        |
| C27     | IC       | steps    | 2024_06_26_0043 IC steps 34.abf        |
```

You can pipe all the line commands. For example, get cell_id and file_name for all IC steps:

```
!bash -c "csvgrep -c protocol -m IC data/data_bcam_2024/records_metadata_clean.
↪csv \
| csvgrep -c prot-opt -m steps \
| csvcut -c cell_id,file_name \
| csvlook"
```

```
| cell_id | file_name                              |
| ------- | -------------------------------------- |
| C1      | 2024_03_15_0000 IC steps 23.abf        |
| C1      | 2024_03_15_0018 IC steps 34.abf        |
| C2      | 2024_03_15_0036 IC steps 23.abf        |
| C3      | 2024_03_20_0007 IC square steps 38.abf |
| C23     | 2024_06_19_0000 IC steps 25.abf        |
| C23     | 2024_06_19_0009 IC steps 34.abf        |
| C24     | 2024_06_19_0014 IC steps 25.abf        |
| C25     | 2024_06_26_0000 IC steps 25.abf        |
| C25     | 2024_06_26_0016 IC steps 34.abf        |
| C25     | 2024_06_26_0022 IC steps 40.abf        |
| C26     | 2024_06_26_0028 IC steps 25.abf        |
| C27     | 2024_06_26_0033 IC steps 25.abf        |
| C27     | 2024_06_26_0043 IC steps 34.abf        |
```

(`bash -c` is a way to tell Bash to execute a command string as if you typed it directly in a terminal)

CHAPTER

FOUR

EXPLORING RECORDS WITH `pyABF`

4.1 Using External Python Packages

We assume that the correct Python environment is already set up; see Section "*Setting Up the Conda Virtual Environment for This Project*" for details.

Let's import the external libraries needed to work with the notebook:

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import matplotlib as mpl
import seaborn as sns
import pyabf

print(f"numpy: {np.__version__}")
print(f"matplotlib: {matplotlib.__version__}")
print(f"seaborn: {sns.__version__}")
print(f"pyabf: {pyabf.__version__}")
```

```
numpy: 1.26.4
matplotlib: 3.8.4
seaborn: 0.12.2
pyabf: 2.3.8
```

4.2 `pyABF` 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:

- a (good) tutorial by [Scott W Harden]
- pyABF - A simple Python interface for Axon Binary Format ABF files, with git repository
- in Python Package Index pypi

4.3 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 consider `abf` files contained in the data directory `data/data_patch_clamp_bcam/records`:

```
from pathlib import Path
records_dir = Path("data/data_patch_clamp_bcam/records")
abf_files = list(records_dir.rglob("*.abf")) # recursive glob
print("the directory ", records_dir, " contains ", len(abf_files)," abf files")
```

```
the directory  data/data_patch_clamp_bcam/records  contains  63  abf files
```

We focus on specific records:

```
file_path = "data/data_patch_clamp_bcam/records/2024_06/06.26/C2/2024_06_26_0028.
↪abf"
```

4.3.1 `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.

4.3.2 `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.

4.3.3 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 (v2.9) with 2 channels (mV, pA), sampled at 10.0 kHz, containing 14 sweeps,␣
↪having no tags, with a total length of 2.35 minutes, recorded with protocol
↪"IV-DG".
```

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/0-git.nosync/data-science-spikes/
↪data/data_patch_clamp_bcam/records/2024_06/06.26/C2/2024_06_26_0028.abf
       File Version: 2.9.0.0
      Sampling Rate: 10000 Hz
       Total Sweeps: 14
       ADC Channels: ['IN 0', 'IN 1']
       DAC Channels: ['OUT 0', 'OUT 1']
      Channel Units: mV
   Experiment Date: 2024-06-26 17:23:16.253000
```

Methods

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

```
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 input**:

```
# 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): [-65.4175 -65.4175 -65.4236 ... -64.6851 -64.6851 -64.
↪6851]
     Command Input (mV): [0. 0. 0. ... 0. 0. 0.]
```

4.3.4 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
```

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

print(f"{'Voltage Trace (mV):':>25} {abf.sweepY}")  # Print voltage trace␣
↪(recorded signal)
print(f"{'Command Input (mV):':>25} {abf.sweepC}")  # Print command input (if␣
↪available)
print(f"{'Recorded Channels:':>25} {abf.adcNames}") # Check all available ADC␣
↪channels (recorded signals)
print(f"{'Command Channels:':>25} {abf.dacNames}")  # Check DAC channels (command␣
↪input signals)
```

```
      Voltage Trace (mV): [-65.4175 -65.4175 -65.4236 ... -64.6851 -64.6851 -64.
↪6851]
      Command Input (mV): [0. 0. 0. ... 0. 0. 0.]
       Recorded Channels: ['IN 0', 'IN 1']
        Command Channels: ['OUT 0', 'OUT 1']
```

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): -81.502
         Median (mV): -91.797
            Min (mV): -93.097
            Max (mV): -64.545
         Std Dev (mV): 12.916
          Range (mV): 28.552
```

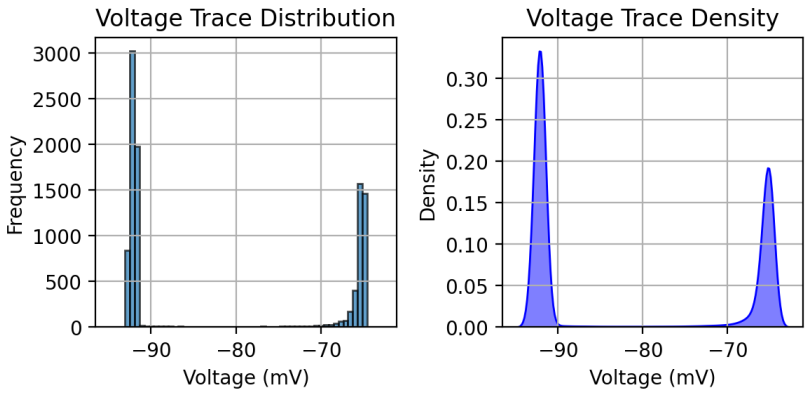
4.4 Plots

4.4.1 Plotting the voltage trace distribution

```
%config InlineBackend.figure_format = 'retina'
```

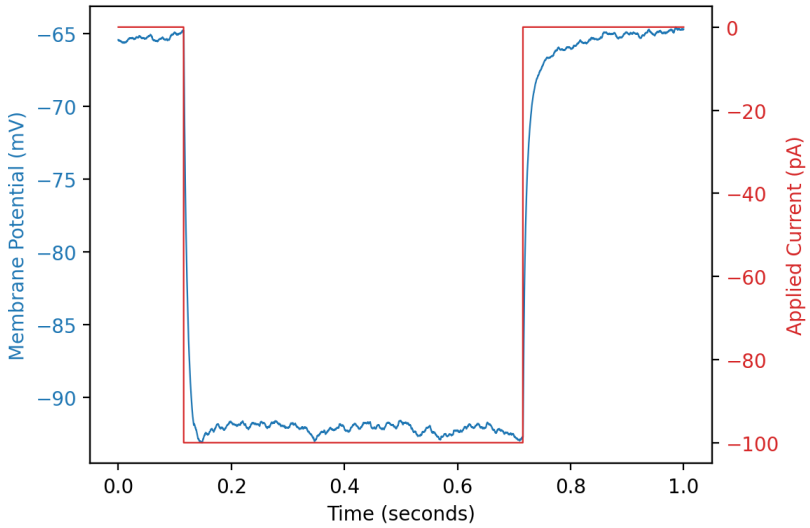
this configuration, known as the inline backend, helps achieve a balance between good visual quality and manageable file size, see Section *Jupyter backends*.

```
# to avoid warning from Seaborn internally calling a Pandas option
# (mode.use_inf_as_na) that has been deprecated in pandas ≥ 2.1.
import warnings  # needed for filterwarnings
warnings.filterwarnings("ignore", category=FutureWarning, module="seaborn")

mpl.rcParams['figure.figsize'] = (6, 3)

# Create a figure with two subplots (1 row, 2 columns), sharing x-axis
fig, axes = plt.subplots(1, 2, sharex=True)

# 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=0.3, 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()
```



4.4.2 Plotting one sweep with input current

```
color_adc = "C0"
color_dac = "C3"
my_lw = 0.8

mpl.rcParams['figure.figsize'] = (6, 4)


# Create the figure
fig, ax1 = plt.subplots()

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

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

# Improve the layout
fig.tight_layout()
plt.show()
```



let put that last plotting in a function `plot_abf_sweep`

```
from utils.plots import plot_abf_sweep

help(plot_abf_sweep) # help/info for the function plot_abf_sweep
```

```
Help on function plot_abf_sweep in module utils.plots:

plot_abf_sweep(abf, sweep=0, color_adc='C0', color_dac='C3', lw=0.8)
    Plot a single sweep from an ABF file with ADC on the left y-axis
    and DAC on the right y-axis.

    Parameters:
    -----------
```

(continues on next page)

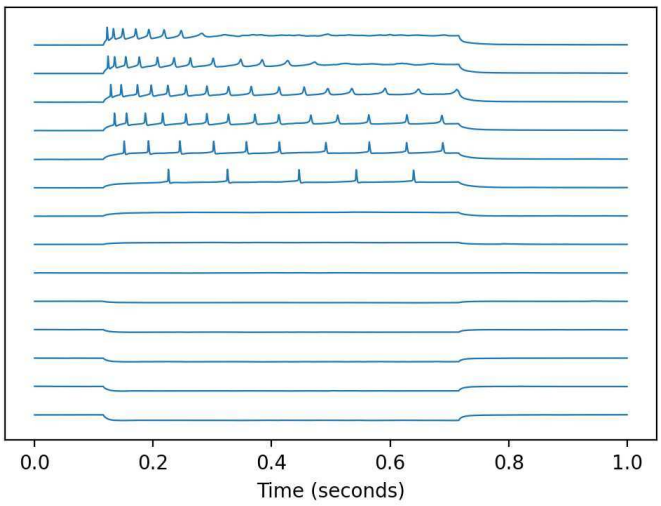
(continued from previous page)

```
    abf : pyabf.ABF
        The ABF object.
    sweep : int
        Sweep number to plot (default 0).
    color_adc : str
        Color for the ADC waveform (default "C0").
    color_dac : str
        Color for the DAC waveform (default "C3").
    lw : float
        Line width (default 0.8).
```

`plot_abf_sweep?` gives the `Signature` of the function, so you can see the parameters and defaults, and the `Docstring` of the function (ie., everything inside the """ … """ in the function)

4.4.3 Plotting all sweeps

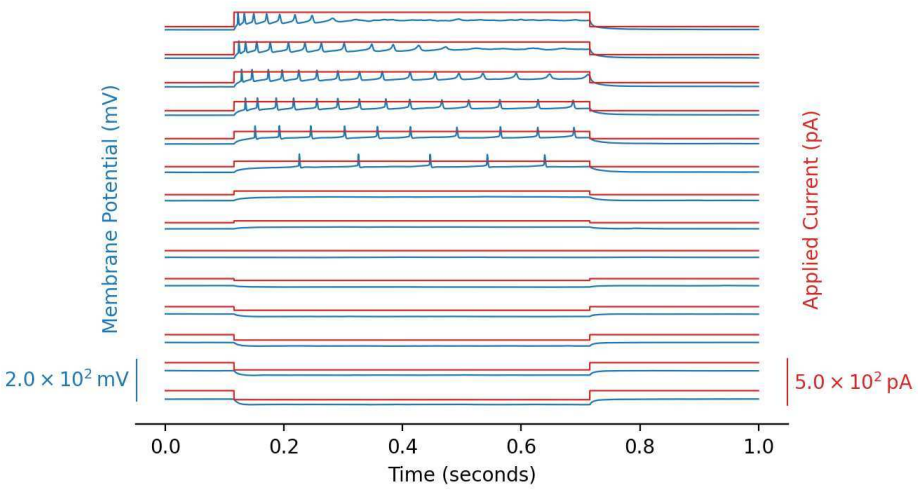
```
# plot every sweep (with vertical offset)
for sweepNumber in abf.sweepList:
    abf.setSweep(sweepNumber)
    offset = 140*sweepNumber
    plt.plot(abf.sweepX, abf.sweepY+offset, color=color_adc, lw=my_lw)

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



4.4.4 Plotting all sweeps with all inputs

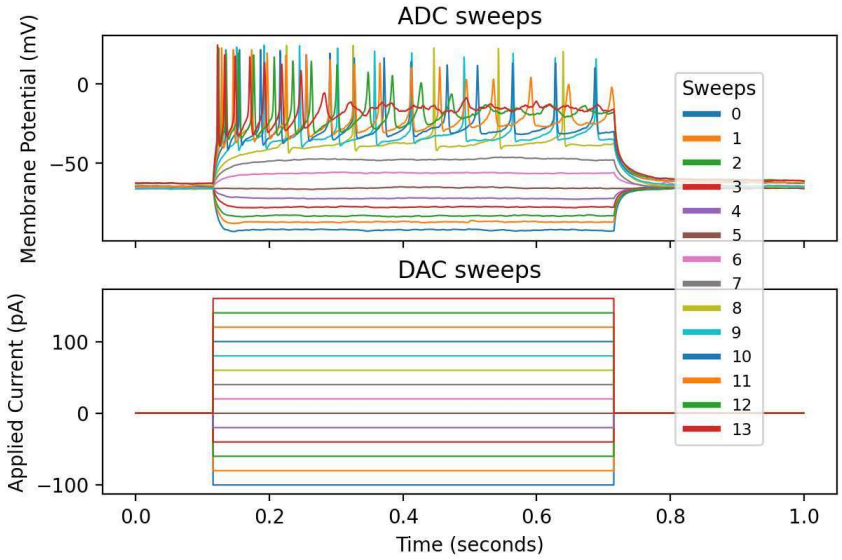
We can improve the previous plot, see the python script `utils/plots.py`:

```
from utils.plots import plot_abf_traces_with_scalebar

fig, ax1, ax2 = plot_abf_traces_with_scalebar(abf)
plt.show()
```



Another possibility:

```
from utils.plots import plot_abf_sweeps_with_legend

fig, (ax1, ax2) = plot_abf_sweeps_with_legend(
    abf, legend_loc='upper left', legend_bbox=(0.8, 0.87), legend_pad=0
)
plt.show()
```



Part II

Appendices

CHAPTER

FIVE

INSTALLING PYTHON AND SOME TOOLS

5.1 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.

5.1.1 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.

5.1.2 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.

5.1.3 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.

5.1.4 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.

5.1.5 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.

5.1.6 Summary Table

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

Important

⇨ We will therefore focus on the Anaconda solution

5.2 Installing Anaconda and Miniconda

5.2.1 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
```

5.2.2 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
```

5.2.3 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.

5.2.4 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

5.3 Conda

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

5.3.1 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
```

5.3.2 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
```

5.3.3 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`

5.4 Conda virtual environment

5.4.1 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.

Important

⇨ Another reason to focus on the Anaconda solution

5.4.2 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
```

5.4.3 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`

5.5 Setting Up the Conda Virtual Environment for This Project

This section is intended for macOS users.

To run this Jupyter Book, I make use of a conda virtual environment, whose recipe i contained in the file `environment.yml` that describes everything needed to create the conda environment named `python-dsspikes-env`:

```
!cat environment.yml
```

```
name: python-dsspikes-env
channels:
  - conda-forge
  - defaults
dependencies:
  - python=3.11
  - numpy=1.26
  - matplotlib=3.8
  - pandoc=3.8       # removed build hash
  - seaborn=0.12
  - jupyter-book=1.0.4
  - jupyterlab=4.1
  - ipykernel
  - csvkit
  - pip
  - pip:
      - pyabf==2.3.8
```

`name: python-dsspikes-env` tells conda what name you assign to the environment. WHen you run:

```
conda env create -f environment.yml
```

Conda reads that line and creates an environment with that name. So after creation, you'll activate it with:

```
conda activate python-dsspikes-env
```

Each time you run it, conda will move one step "up": If you're inside python-dsspikes-env, it will go back to (base). If you're already in (base), it will deactivate completely (no environment active). So the cycle is:

```
conda activate python-dsspikes-env
# ... work here ...
conda deactivate
```

The quickest way to check which conda environment is active is:

```
conda info --envs
```

or its shorthand:

```
conda env list
```

hence:

```
(python-dsspikes-env) data-science-spikes$ conda env list
# conda environments:
#
base                  /Users/campillo/miniforge3
myenv                 /Users/campillo/miniforge3/envs/myenv
python-dsspikes-env   * /Users/campillo/miniforge3/envs/python-dsspikes-env
```

- The `*` shows which environment is currently active.
- In your shell prompt, the active environment name also appears in parentheses, e.g. `(python-dsspikes-env) data-science-spikes$` (here `(conda_virtual_env) directory_name`.

To check the active conda environment inside a Jupyter notebook, you have a few options:

1. Check `sys.executable`

```
import sys
sys.executable # This shows the path to the Python binary being used.
```

```
'/Users/campillo/miniforge3/envs/python-dsspikes-env/bin/python'
```

2. Check environment variables

```
import os
os.environ.get("CONDA_DEFAULT_ENV")
```

```
'python-dsspikes-env'
```

3. Print Python packages & versions To confirm everything is coming from the right env:

```
!which python
!python --version
!pip list | grep -E "pyabf|matplotlib|seaborn"
```

```
/Users/campillo/miniforge3/envs/python-dsspikes-env/bin/python
```

```
Python 3.11.13
```

```
matplotlib                    3.8.4
matplotlib-inline             0.1.7
pyabf                         2.3.8
seaborn                       0.12.2
```

and `!pip list` for the complete list.

Setting Up the Conda Virtual Environment for This Project

This project uses Python packages such as `numpy`, `matplotlib`, `seaborn`, and `pyabf`.
To ensure reproducibility and avoid conflicts with other Python projects, we recommend using a **dedicated Conda virtual environment**.

1. Create the environment - Run the following command in your terminal:

```
conda env create -f environment.yml
```

This will create a new environment named jupyter-env (as specified in `environment.yml`). All required packages for this Jupyter Book project will be installed.

2. Activate the environment

```
conda activate jupyter-env
```

Your terminal prompt should now show `(jupyter-env)`, indicating the environment is active.

3. Make the environment available in Jupyter

```
python -m ipykernel install --user --name=jupyter-env --display-name "Python␣
↪(jupyter-env)"
```

This allows notebooks to select the correct kernel.

4. Verify installation - You can test that everything is installed correctly:

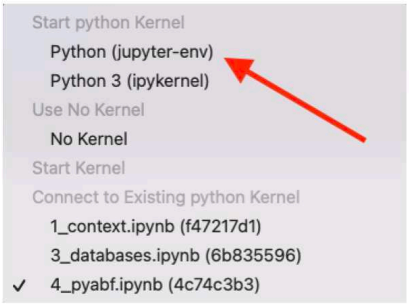
```
python -c "import numpy, matplotlib, seaborn, pyabf; print('All imports OK!')"
```

If no errors appear, the environment is ready to use.

5. Launch Jupyter Lab or Notebook

```
# Launch Jupyter Lab
jupyter lab

# or launch classic Jupyter Notebook
jupyter notebook
```

In the notebook (top right), select Kernel → `Python (jupyter-env)`.



6. Updating the environment – If you modify environment.yml later (e.g., adding packages), update the environment:

```
conda env update -f environment.yml --prune
```

`--prune` removes packages no longer listed in `environment.yml`.

Notes:

- Keep all project-specific packages inside the virtual environment; do not install them in base.
- For reproducibility, commit `environment.yml` to your repository.

CHAPTER

SIX

INTERACTIVE COMPUTING WITH JUPYTER BAZAAR

Jupyter, Jupyter Notebook, JupyterLab, Binder / MyBinder, Jupyter{book}, Colab, Deepnote

6.1 Jupyter and around

The Jupyter ecosystem evolved to make computing **interactive, reproducible, and shareable**. It began with IPython, an enhanced Python shell for rapid experimentation, and expanded into Jupyter to support multiple languages through a decoupled kernel-interface model. Jupyter Notebook introduced a web-based environment combining code, outputs, and narrative text, ideal for teaching, research, and data analysis. JupyterLab provides a modern, flexible workspace for managing notebooks, scripts, and data in complex projects. Finally, Jupyter{book} allows structured, publication-quality books and websites to be built from notebooks and Markdown, fully executable and shareable. Together, these tools address the need for **interactive coding, reproducibility, multi-language support, and clear communication of computational results**, even beyond Python:

- **Jupyter** is an open-source project that evolved from IPython. IPython originally provided an enhanced interactive Python shell, with powerful features like introspection, rich media, and tab-completion, making Python more user-friendly for experimentation and data analysis. Jupyter extended this idea to **multiple programming languages** (Python, R, Julia, and more) by separating the **kernel** (which executes code) from the **interface** (which displays results interactively).
- **Jupyter Notebook** builds on this foundation, providing a web-based interactive environment where users can write and execute code, display rich outputs (plots, images, LaTeX, widgets), and combine them with narrative text. This allows notebooks to serve as **reproducible documents** for data analysis, teaching, and scientific communication.
- **JupyterLab** is the next-generation interface for Jupyter, offering a **modular, flexible environment** where users can work with notebooks, text editors, terminals, and data files all in one workspace. It improves productivity for complex projects, supports extensions, and makes multi-document workflows smoother than classic notebooks.
- **Binder/MyBinder** is a cloud service that allows users to **launch fully executable Jupyter environments** directly from GitHub repositories. It lets anyone run notebooks or Jupyter Books without installing anything locally, making content fully interactive and reproducible online.
- **Jupyter{book}** extends the Jupyter ecosystem by allowing users to turn **collections of notebooks and Markdown files into interactive, publication-quality books and websites**. Unlike standalone notebooks, Jupyter Books provide structured chapters, table-of-contents, cross-references, and can be executed to show live outputs, making them ideal for teaching, tutorials, and reproducible scientific publications.

```
Jupyter Ecosystem Evolution (approximate years)
===============================================

IPython (2001) - interactive Python shell
        |
        ▼
Jupyter (2014) - multi-language kernel architecture
        |
```

(continues on next page)

(continued from previous page)

```
        ├── Jupyter Notebook (2015) - web-based interactive notebooks
        │      - web-based interactive notebooks
        │
        ├── JupyterLab (2018)
        │      - modern IDE-like workspace for notebooks & files
        │
        ├── Binder / MyBinder (2017)
        │     - cloud service for executable notebooks (and later books)
        │
        └── Jupyter{book} (2018+)
               - structured, publication-quality books from notebooks & Markdown
```

6.2 Colab

Colab is Google's cloud version of Jupyter Notebook, fully integrated with the Jupyter ecosystem, making it easy to run, share, and collaborate on notebooks online.

Colab (2017) sits in this ecosystem as a **cloud-hosted Jupyter Notebook environment**:

- Runs notebooks **without local setup**.
- Provides **free CPU/GPU/TPU resources**.
- Enables **real-time collaboration** like Google Docs.
- Fully compatible with **.ipynb notebooks**, so you can open notebooks from GitHub or Jupyter Book in Colab.

Colab is designed:

- For **students, researchers, or teams** who don't want to install Python locally.
- To **share interactive notebooks** with reproducible results.
- To **test notebooks from GitHub** quickly.

6.3 Jupyter {book}

jupyter {book} and two important files:

- _config.xml
- _toc.yml

Launch into interactive computing interfaces

6.3.1 Jupyter backends

the Jupyter inline backend is what converts your Matplotlib figures into embedded images inside notebooks, and `%config InlineBackend...` is how you control their quality and format.

```
%config InlineBackend.figure_format = 'pdf'     # fine but could be combersome
%config InlineBackend.figure_format = 'svg'     #           "
%config InlineBackend.figure_format = 'retina'  #

%config InlineBackend.figure_format = 'png'
%config InlineBackend.rc = {'figure.dpi': 200}  # or 300 for print quality
```

with `retina` the Jupyter inline backend actually tells Matplotlib to render the figure at double the standard DPI (so if your default is 100 dpi, it makes a 200 dpi PNG). Then the notebook displays it at the normal on-screen size, so it looks sharper on high-density displays (like MacBooks) and also when the image gets embedded in the LaTeX/PDF build.

Other backends:

```
%matplotlib inline    # use inline backend
%matplotlib notebook  # interactive plots inside notebook
%matplotlib widget    # interactive with ipywidgets
```

CHAPTER

SEVEN

CSV IN THE CONTEXT OF PANDA AND CSVKIT

7.1 Basics about csv

CSV, for Comma-Separated Values, is a plain text format where each row represents a record, and columns are separated by commas (,). Example:

```
name,age,city
Alice,34,Paris
Bob,29,London
Charlie,41,Rome
```

CSV is a plain text (ASCII) format that makes data human-readable, easy to generate or parse in virtually any programming language (Python, R, SQL, Excel, etc.), and portable across operating systems (Windows, Unix, macOS). Because it's just text with no dependencies, CSV files are simple to share—whether by email, version control systems like GitHub, or quick inspection with tools such as `less` or `cat`. They work well for small-to-medium datasets, even with millions of rows, but come with limitations: all values are treated as text until explicitly parsed, and the format lacks support for data types or nested/complex structures.

With python there are 2 ways to deal with CSV:

- **Built-in support:** Python's standard library includes the `csv` module, which lets you read and write CSV files without installing anything extra. It handles splitting rows into fields, quoting, delimiters, and more.
- **With pandas:** For data analysis, the `pandas` library makes CSV handling much more powerful. `pandas.read_csv("file.csv")` loads data directly into a `DataFrame`, automatically inferring types (numbers, strings, dates) and offering options for missing values, encodings, and delimiters. Saving back is just as easy with `.to_csv()`.

Here we will use this second solution. Later we will also present a non-python handy tool called `csvkit` which regroups command-line utilities for quick inspection and transformation of CSV files, often faster than writing a Python script.

7.2 Pandas DataFrame

pandas DataFrame (df) is far better than built-in Python tools like lists, dictionaries, or arrays for the following reasons:

- Tabular structure built-in
  - A DataFrame behaves like a table or spreadsheet: rows = records, columns = fields.
  - You don't have to manually manage parallel lists or nested dictionaries.
- Easy indexing and filtering: doing the same with lists/dicts would require loops and conditionals — much more verbose
- Powerful aggregation & grouping: compute counts, averages, sums, or custom statistics without writing loops
- Built-in handling of missing data

  - Pandas understands `NaN` values automatically.
  - Built-in functions handle missing data gracefully.
- Integration with plotting and analysis
- Easy I/O; Load/save CSV, Excel, SQL, JSON, and more with a single command.

See infra.

Main methods on df:

- Exploring the structure

```
df.shape          # dimensions (rows, columns)
df.columns        # list of column names
df.dtypes         # data types of each column
df.info()         # concise summary
df.head(5)        # first 5 rows
df.tail(5)        # last 5 rows
```

- Inspecting the data

```
df.describe()          # statistics (mean, std, min, max, quartiles)
df.value_counts()      # count frequency of values (for a Series)
df.unique()            # unique values (for a Series)
df.isnull().sum()      # count missing values per column
df.sample(5)           # random sample of rows
```

- Selecting and filtering

```
df['col']              # select one column
df[['col1','col2']]    # select multiple columns
df.loc[0]              # select by label
df.iloc[0]             # select by index
df[df['col'] > 10]     # filter rows
```

- Sorting and grouping

```
df['col']              # select one column
df[['col1','col2']]    # select multiple columns
df.loc[0]              # select by label
df.iloc[0]             # select by index
df[df['col'] > 10]     # filter rows
```

- Modifying

```
df.rename(columns={'old':'new'}, inplace=True)  # rename columns
df.drop(columns=['col'], inplace=True)          # drop column
df.dropna()                                     # drop rows with NaN
df.fillna(0)                                    # fill NaN with 0
df.assign(newcol=df['col']*2)                   # add new column
```

- Exporting

```
df.to_csv('file.csv', index=False)  # save as CSV
df.to_excel('file.xlsx', index=False) # save as Excel
```

7.3 `csvkit` the command-line Swiss Army knife

7.3.1 Installation under you favorite conda environment

```
conda activate python-dsspikes-env
conda install -c conda-forge csvkit # That updates the live environment.
```

Export the environment back to `environment.yml`: Now that your environment has `csvkit`, you need to reflect it in your YAML file:

```
conda env export --from-history > environment.yml
```

`--from-history` ensures only the packages you explicitly installed are recorded (cleaner file, avoids tons of build hashes). Now your `environment.yml` will include csvkit in dependencies.

7.3.2 Basics command-line tools

`in2csv`: Excel Slayer

converts various tabular data formats—like Excel (.xls, .xlsx), DBF, fixed-width, or even Google Sheets—into clean, standard CSV.

```
# Convert Excel to CSV and print to stdout
in2csv data.xlsx

# Convert Excel to CSV and save to a file
in2csv data.xlsx > data.csv

# List sheet names in an Excel file
in2csv -n data.xlsx

# Convert a specific sheet
in2csv -s "Sheet1" data.xlsx > data.csv
```

Convert formats, here CSV to JSON:

```
in2csv data/records_fake_metadata.csv | csvjson > metadata.json
```

`csvlook`: Data Periscope

Quickly inspect your CSV in the terminal. Allows you to display CSV files in a nicely formatted, readable table in the terminal — almost like a "pretty-printed" view of your data. Pipe to `less -S` to scroll horizontally:

```
csvlook data/records_fake_metadata.csv | less -S
```

Preview the first few rows:

```
csvlook data/records_fake_metadata.csv | head -n 12
```

`csvcut`: Data Scalpel

Select columns by name or index:

```
# By name
csvcut -c cell_id,protocol,prot-opt data/records_fake_metadata.csv

# By index (first column is 1)
csvcut -c 1,4,6 data/records_fake_metadata.csv
```

`csvgrep`: Data Filter

Filter rows based on column values:

```
# Keep only rows where protocol is IC
csvgrep -c protocol -m IC data/records_fake_metadata.csv

# Keep only rows where prot-opt contains "ramp"
csvgrep -c prot-opt -m ramp data/records_fake_metadata.csv
```

`csvsort`: Data Organizer

Sort your CSV by one or more columns:

```
# Sort by date
csvsort -c date data/records_fake_metadata.csv

# Sort by protocol, then by date
csvsort -c protocol,date data/records_fake_metadata.csv
```

`csvstat`: Quick Stats

Get basic stats and info about the CSV:

```
csvstat data/records_fake_metadata.csv
```

Combining commands

You can pipe all the line commands. For example, get cell_id and file_name for all IC steps:

```
csvgrep -c protocol -m IC data/records_fake_metadata.csv \
  | csvgrep -c prot-opt -m steps \
  | csvcut -c cell_id,file_name \
  | csvlook
```

CHAPTER

EIGHT

DIAGNOSTIC

8.1 Myst

This page tests **MyST Markdown extensions**.

8.1.1 Task list

- [x] Done
- [ ] Not yet done

8.1.2 Admonitions

Note

This is a *note* admonition.

Warning

This is a *warning* admonition.

8.2 Emoji Test Page

This page demonstrates using emojis in a Jupyter Book PDF.

8.2.1 Inline Emoji

Here is a lightbulb emoji inline: :emoji:`1F4A1`

Here is a rocket emoji inline: :emoji:`1F680`

8.2.2 Emoji with text

- :emoji:1F4A1 **Idea:** Always document your data!
- :emoji:1F680 **Launch:** Start your analysis.
- :emoji:1F680 **Rocket + Math:** $E = mc^2$:emoji:1F680

8.2.3 Emoji in a list

1. :emoji:1F4DA Read the documentation
2. :emoji:1F4BB Use Python
3. :emoji:1F4A1 Generate ideas

8.2.4 Emoji in headers

:emoji:1F680 Getting Started

:emoji:1F4A1 Tips & Tricks

INDEX