The kaobook class

Use this document as a template

My PhD Thesis

Customise this page according to your needs

Tobias Hangleiter*

April 3, 2025

* A LaTeX lover/hater

The kaobook class

Disclaimer
You can edit this page to suit your needs. For instance, here we have a no copyright statement, a colophon and some other information. This page is based on the corresponding page of Ken Arroyo Ohori's thesis, with minimal changes.

Colophon
This document was typeset with the help of KOMA-Script and LaTeX using the kaobook class.
The source code of this book is available at:
https://github.com/fmarotta/kaobook
(You are welcome to contribute!)

Publisher
First printed in May 2019 by

The harmony of the world is made manifest in Form and Number, and the heart and soul and all the poetry of Natural Philosophy are embodied in the concept of mathematical beauty.

– D’Arcy Wentworth Thompson

Contents

Part I

A Flexible Python Tool For Fourier-Transform Noise Spectroscopy

The `python_spectrometer` software package 1

In this chapter, I will lay out the design and functionality of the `python_spectrometer` Python package.[1]

1: The package repository is hosted on GitLab. Its documentation is automatically generated and hosted on GitLab as well. Releases are automatically published to PyPI and allow the package to be installed using `pip install python-spectrometer`.

1.1 Package design and implementation

The `python_spectrometer` package provides a central class, `Spectrometer`, that users interact with to perform data acquisition, spectrum estimation, and plotting. It is instantiated with an instance of a child class of the `DAQ` base class that implements an interface to various data acquisition device (DAQ) hardware devices. New spectra are obtained by calling the `Spectrometer.take()` method with all acquisition and metadata settings.

In the following, I will go over the the design of these aspects of the package in more detail.

Table 1.1: Variable names used in ?? and their corresponding parameter names as used in `python_spectrometer` and `scipy.signal.welch()` [1].

Variable	Parameter
L	`n_pts`
f_s	`fs`
K	`noverlap`
N	`nperseg`
M	`n_seg`
f_{min}	`f_min`
f_{max}	`f_max`

1.1.1 Data acquisition

The `daq` module contains on the one hand the declaration of the `DAQ` abstract base class and its child class implementations, and on the other the `settings` module, which defines the `DAQSettings` class. This class is used in the background to validate data acquisition settings both for consistency (c.f. ??) and hardware constraints.

To better understand the necessity of this functionality, consider the typical scenario of a physicist[2] in the lab. Alice has wired up her experiment, performed a first measurement, and to her dismay discovered that the data is too noisy to see the sought-after effect. She sets up the `python_spectrometer` code to investigate the noise spectrum of her measurement setup. From her noisy data she could already estimate the frequency of the most harrowing noise, so she knows the frequency band $[f_{min}, f_{max}]$ she is most interested in. But because she is lazy,[3] she does not want to do the mental gymnastics to convert f_{min} to the parameter that her DAQ device understands, L (see Table 1.1), especially considering that L depends on the number of Welch averages and the overlap. Furthermore, while she could just about do the conversion from f_{max} to the other relevant DAQ parameter, f_s, in her head, her device imposes hardware constraints on the allowed sample rates she can select! The `DAQSettings` class addresses these issues. It is instantiated with any subset of the parameters listed in Table 1.1[4] and attempts to resolve the parameter interdependencies lined out in ?? upon calling `DAQSettings.to_consistent_dict()`.[5] This either infers those parameters that were not given from those that were or, if not possible, uses a default value. Child classes of the `DAQ` class can subclass `DAQSettings` to implement hardware constraints such as a finite set of allowed sampling rates or a maximum number of samples per data buffer.

2: Let's call her Alice.

3: Physicists generally are.

4: `DAQSettings` inherits from the builtin `dict` and as such can contain arbitrary other keys besides those listed in Table 1.1. However, automatic validation of parameter consistency is only performed for these special keys.

5: Since the graph spanned by the parameters is not acyclic, this only works *most* of the time.

For instance, Alice might want to measure the noise spectrum in the frequency band [1.5 Hz, 72 kHz]. Although she would not have to do this explicitly,[6] she could inspect the parameters after resolution using the code shown in Listing 1.1.

6: Settings are automatically parsed when passed to the `take()` method of the `Spectrometer` class.

Listing 1.1: `DAQSettings` example showcasing automatic parameter resolution. `n_avg` determines the number of outer averages, *i. e.*, the number of data buffers acquired and processed individually.

```
>>> from python_spectrometer.daq import DAQSettings
>>> settings = DAQSettings(f_min=1.5, f_max=7.2e4)
>>> settings.to_consistent_dict()
{'f_min': 1.5,
 'f_max': 72000.0,
 'fs': 144000.0,
 'df': 1.5,
 'nperseg': 96000,
 'noverlap': 48000,
 'n_seg': 5,
 'n_pts': 288000,
 'n_avg': 1}
```

```
{'f_min': 14.30511474609375,
 'f_max': 72000.0,
 'fs': 234375.0,
 'df': 14.30511474609375,
 'nperseg': 16384,
 'noverlap': 0,
 'n_seg': 1,
 'n_pts': 16384,
 'n_avg': 1}
```

Listing 1.2: Resolved settings for the same input parameters as in Listing 1.1 but for the `ZurichInstrumentsMFLIScope` backend with hardware constraints on `n_pts` and `fs`.

If the instrument she'd chosen for data acquisition had been a Zurich Instruments MFLI's "Scope" module [2], the same requested settings would have resolved to those shown in Listing 1.2.[7] This is because the Scope module constrains $L \in [2^{12}, 2^{14}]$ and $f_s \in 60\,\mathrm{MHz} \times 2^{[-16,0]} \approx \{915.5\,\mathrm{Hz}, \dots, 30\,\mathrm{MHz}, 60\,\mathrm{MHz}\}$.

[2]: (n.d.), *Scope Module - LabOne API User Manual*

7: And issued a warning to inform the user their requested settings could not be matched.

8: Which might differ from the requested settings as outlined above.

As already mentioned, the `DAQ` base class implements a common interface for different hardware backends, allowing the `Spectrometer` class to be hardware agnostic. That is, changing the instrument that is used to acquire the data does not necessitate adapting the code used to interact with the instrument. To enable this, different instruments require small wrapper drivers that map the functionality of their actual driver onto the interface dictated by the `DAQ` class. This is achieved by subclassing `DAQ` and implementing the `DAQ.setup()` and `DAQ.acquire()` methods. Their functionality is best illustrated by the internal workflow. When acquiring a new spectrum, all settings supplied by the user are first fed into the `setup()` method where instrument configuration takes place. The method returns the actual device settings,[8] which are then forwarded to the `acquire()` generator function. Here, the instrument is armed (if necessary), and subsequently data is fetched from the device and yielded to the caller `n_avg` times, where `n_avg` is the number of outer averages. Listing 1.3 represents the data acquisition workflow as pseudocode.

DAQ pseudocode?

1.1.2 Data processing

Introduce Bob?

Once time series data has been acquired using a given `DAQ` backend, it could in principle immediately be used to estimate the PSD following ??. However, it is often desirable to transform, or process, the data in

Listing 1.3: DAQ workflow pseudocode. A `SomeDAQ` object (representing the instrument `Some`) is instantiated with a driver object (for instance a `QCoDeS Instrument`). The instrument is configured with the given `user_settings`. Calling the generator function `daq.acquire()` with the actual device settings returns a generator, iterating over which yields one data buffer per iteration. The data buffers can then be passed to further processing functions (the power spectral density (PSD) estimator in our example).

```
daq = SomeDAQ(driver)
parsed_settings = daq.setup(**user_settings)
acquisition_generator = daq.acquire(**parsed_settings)
for data_buffer in acquisition_generator:
    do_something_with(data_buffer)
```

some fashion. This can include simple transformations such as accounting for the gain of a transimpedance amplifier (TIA) and convert the voltage back to a current,[9] or more complex ones such as applying calibrations. In particular, since the process of computing the PSD already involves Fourier transformation, the processing can also be performed in frequency space.

9: Although it is of course less than trivial to discriminate between current and voltage noise in a TIA.

In `python_spectrometer`, this can be done using a `procfn` (in the time domain) or `fourier_procfn` (in the Fourier domain). The former is specified as an argument directly to the `Spectrometer` constructor. It is a callable with signature `(x, **kwargs) -> xp`, that is, takes the time series data as its first (positional) argument and arbitrary settings that are passed through from the `take()` method as keyword arguments, and returns the processed data. Listing 1.4 shows a simple function that accounts for the gain of an amplifier.

```
def compensate_gain(x, gain=1.0):
    return x / gain
```

Listing 1.4: A simple `procfn`, which converts amplified data back to the level before amplification.

```
def derivative(xf, f, n=0):
    return xf / (2j * pi * f)**n
```

Listing 1.5: A simple `fourier_procfn`, which calculates the (anti-)derivative.

The latter is specified in the `psd_estimator` argument of the `Spectrometer` constructor. This argument allows the user to specify a custom estimator for the PSD, in which case a callable is expected. Otherwise, it should be a mapping containing parameters for the default PSD estimator, `scipy.signal.welch()` [1]. Here, the keyword `fourier_procfn` should be a callable with signature `(xf, f, **kwargs) -> (xfp, fp)`.[10] That is, it should take the frequency-space data, the corresponding frequencies, and arbitrary keyword arguments and return a tuple of the processed data and the corresponding frequencies. The latter are required in case the function modifies the frequencies.[11] A simple example for a processing function in Fourier space is shown in Listing 1.5, which computes the (anti-)derivative of the data using the fact that

[1]: (n.d.), *Welch — SciPy v1.15.2 Manual*

10: I. e., the `psd_estimator` argument would be `{"fourier_procfn": fn}`.

11: One example is the `octave_band_rms()` function from the `qutil.signal_processing` module [3]

$$\frac{\partial^n}{\partial t^n} \xrightarrow{\text{F.T.}} (\mathrm{i}\omega)^n \tag{1.1}$$

under the Fourier transform. In Part II, I discuss more complex use-cases of the processing functionality included in `python_spectrometer` in the context of vibration spectroscopy.

ref

1.2 Feature overview

überleitung

1.2.1 Sequential spectrum acquisition

Now that we have a basic understanding of the design choices underlying `python_spectrometer`, let us discuss the typical workflow of using the package. The default mode for spectrum acquisition using `python_spectrometer` revolves around the `take()` method. Key to this workflow is the idea that each acquired spectrum can be assigned a comment that allows to easily identify a spectrum in the main plot. For instance, this comment could contain information about the particular settings that were active when the spectrum was recorded, or where a particular cable was placed.

Consider as an example the procedure of "noise hunting", *i. e.*, debugging a noisy experimental setup. The experimentalist,[12] having discovered that his data is noisier than expected, sets up the `Spectrometer` class with an instance of the `DAQ` subclass for the DAQ instrument connected to his sample. Choosing the frequency bounds f_{min} and f_{max}, and using the sensible defaults for the remaining spectrum parameters, Charlie first

12: Let's call him Charlie.

Listing 1.6: Setup and workflow using the `python_spectrometer` package. `session` and `device` are Application Programming Interface (API) objects of the `zhinst.toolkit` driver package. It is therefore possible to simply use the driver objects that are already in use in the measurement setup.

```
from python_spectrometer import Spectrometer, daq

mfli_daq = daq.ZurichInstrumentsMFLIDAQ(session, device)
spect = Spectrometer(mfli_daq)
spect.take('baseline', f_min=f_min, f_max=f_max)
```

grounds the input of his DAQ to record a *baseline* spectrum. Thus far, his code would hence look something like that shown in Listing 1.6.

Part II

Characterization and Improvements of a Millikelvin Confocal Microscope

Part III

Electrostatic Trapping of Excitons in Semiconductor Membranes

Part IV

A Filter-Function Formalism For Quantum Operations

Appendix

Bibliography

[1] *Welch — SciPy v1.15.2 Manual.* URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.welch.html (visited on 03/31/2025) (cited on pages 3, 5).

[2] *Scope Module - LabOne API User Manual.* URL: https://docs.zhinst.com/labone_api_user_manual/modules/scope/index.html (visited on 04/02/2025) (cited on page 4).

[3] *Octave_band_rms — Qutil 2025.3.1 Documentation.* URL: https://qutech.pages.rwth-aachen.de/qutil/_autogen/qutil.signal_processing.fourier_space.octave_band_rms.html (visited on 04/03/2025) (cited on page 5).

Special Terms

A

API Application Programming Interface. 6

D

DAQ data acquisition device. 3–6

P

PSD power spectral density. 4, 5

T

TIA transimpedance amplifier. 5