The kaobook class

Use this document as a template

My PhD Thesis

Customise this page according to your needs

Tobias Hangleiter[*]

April 15, 2025

[*] A LaTeX lover/hater

The kaobook class

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

Introduction 1

Noise is ubiquitous in condensed matter physics experiments, and in mesoscopic systems in particular it can easily drown out the sought-after signal. Hence, characterizing (and subsequently mitigating) noise is an essential task for the experimentalist. But noise comes in as many different forms as there are types of signal sources and detectors, whether it be a voltage source or a photodetector, and while some instruments have built-in solutions for noise analysis, they vary in functionality and capability. Moreover, the measured signal often does not directly correspond to the noisy physical quantity of interest, making it desirable to be able to manipulate the raw data before processing.

Theory of spectral noise estimation 2

THERE exists a multitude of methods for estimating noise properties of a classical signal $x(t)$.[1]

1: We discuss only classical noise here, meaning $x(t)$ commutes with itself at all times. For descriptions of and spectroscopy protocols for quantum noise refer to Refences **Clerk2010** and **Paz-Silva2017**, for example.

lay out some others

If the noisy process $x(t)$ has Gaussian statistics, meaning that the value at a given point in time follows a normal distribution with some mean μ and variance σ^2 over multiple realizations of the process, it can be fully described by the power spectral density (PSD) $S(\omega)$.[2] For the purpose of noise estimation, the assumption of Gaussianity is a rather weak one as the noise typically arises from a large ensemble of individual fluctuators and is therefore well approximated by a Gaussian distribution by the central limit theorem.[3] Even if the process $x(t)$ is not perfectly Gaussian, non-Gaussian contributions can be seen as higher-order contributions if viewed from the perspective of perturbation theory, and therefore the PSD still captures a significant part of the statistical properties. For this reason, the PSD is the central quantity of interest in noise spectroscopy and I will discuss some of its properties in the following.

2: The term *power spectrum* is often used interchangably. I will do so as well, but emphasize at this point that in digital signal processing in particular, the *spectrum* is a different quantity from the *spectral density*. See also sidenote 17 in Chapter 3.

maybe a classical signal processing ref?

3: As an example, consider electronic devices, where voltage noise arises from a large number of defects and other charge traps in oxides being populated and depopulated at certain rates γ. The ensemble average over these so-called two-level fluctuators (TLFs) then yields the well-known $1/f$-like noise spectra (at least for a large density [**Mehmandoost2024**]).

flesh out this sidenote?

For real signals $x(t) \in \mathbb{R}$, $S(\omega)$ is an even function and one therefore distinguishes the *two-sided* PSD $S^{(2)}(\omega)$ defined over $\mathbb{R}$ from the *one-sided* PSD $S^{(1)}(\omega) = 2S^{(2)}(\omega)$ defined only over $\mathbb{R}^+$. Complex signals $x(t) \in \mathbb{C}$ such as those generated by Lock-in amplifiers after demodulation in turn have asymmetric, two-sided PSDs.

flesh out

2.1 Spectrum estimation from time series

To see how the PSD may be estimated from time-series data, consider a continuous wide-sense stationary[4] signal in the time domain $x(t) \in \mathbb{C}$ that is observed for some time T. We define the windowed Fourier transform of $x(t)$ and its inverse by[5]

$$\hat{x}_T(\omega) = \int_0^T \mathrm{d}t\, x(t)\mathrm{e}^{-\mathrm{i}\omega t} \tag{2.1}$$

$$\text{and} \quad x(t) = \int_{-\infty}^{\infty} \frac{\mathrm{d}\omega}{2\pi}\hat{x}_T(\omega)\mathrm{e}^{\mathrm{i}\omega t}, \tag{2.2}$$

4: For a wide-sense stationary (also called weakly stationary) process $x(t)$, the mean is constant and the auto-correlation function $C(t,t') = \langle x(t)^* x(t')\rangle$ is given by $\langle x(t)^* x(t+\tau)\rangle = \langle x(0)^* x(\tau)\rangle$ with $\tau = t' - t$. That is, it is a function of only the time lag τ and not the absolute point in time. For Gaussian processes as discussed here, this also implies stationarity [**Koopmans1995**]. The property further implies that $C(\tau)$ is an even function.

sketch of auto-correlation function?

5: In this chapter we will always denote the Fourier transform of some quantity ξ using the same symbol with a hat, $\hat{\xi}$.

i. e., we assume that outside of the window of observation $x(t)$ is zero. The auto-correlation function of $x(t)$ is given by

$$C(\tau) = \langle x(t)^* x(t+\tau)\rangle \tag{2.3}$$

$$= \lim_{T\to\infty}\frac{1}{T}\int_0^T \mathrm{d}t\, x(t)^* x(t+\tau), \tag{2.4}$$

where $\langle\,\cdot\,\rangle$ is the ensemble average over multiple realizations of the process and the last equality holds true for ergodic processes. Expressing $x(t)$ in terms of its Fourier representation (Equation 2.1) and reordering the integrals, we get[6]

6: Mathematicians might at this point argue the integrability of $x(t)$, but as we deal with physical processes with finite bandwidth (and have no shame), we do not.

$$C(\tau) = \lim_{T\to\infty}\frac{1}{T}\int_0^T \mathrm{d}t \int_{-\infty}^{\infty}\frac{\mathrm{d}\omega}{2\pi}\hat{x}_T(\omega)^* \mathrm{e}^{-\mathrm{i}\omega t}\int_{-\infty}^{\infty}\frac{\mathrm{d}\omega'}{2\pi}\hat{x}_T(\omega')\mathrm{e}^{\mathrm{i}\omega'(t+\tau)} \tag{2.5}$$

$$= \lim_{T\to\infty}\frac{1}{T}\int_{-\infty}^{\infty}\frac{\mathrm{d}\omega}{2\pi}\int_{-\infty}^{\infty}\frac{\mathrm{d}\omega'}{2\pi}\hat{x}_T(\omega)^*\hat{x}_T(\omega')\mathrm{e}^{\mathrm{i}\omega'\tau}\int_0^T \mathrm{d}t\,\mathrm{e}^{\mathrm{i}t(\omega'-\omega)} \tag{2.6}$$

7: Note that, because $x(t)$ is wide-sense stationary, we may shift the limits of integration $\int_0^T \to \int_{-T/2}^{+T/2}$.

The innermost integral approaches a δ-function for large T,[7] allowing us to further simplify this under the limit as

$$C(\tau) = \lim_{T\to\infty}\frac{1}{T}\int_{-\infty}^{\infty}\frac{\mathrm{d}\omega}{2\pi}\int_{-\infty}^{\infty}\frac{\mathrm{d}\omega'}{2\pi}\hat{x}_T(\omega)^*\hat{x}_T(\omega')\mathrm{e}^{\mathrm{i}\omega'\tau}\delta(\omega'-\omega) \tag{2.7}$$

$$= \lim_{T\to\infty}\frac{1}{T}\int_{-\infty}^{\infty}\frac{\mathrm{d}\omega}{2\pi}|\hat{x}_T(\omega)|^2\mathrm{e}^{\mathrm{i}\omega\tau} \tag{2.8}$$

$$= \int_{-\infty}^{\infty}\frac{\mathrm{d}\omega}{2\pi}S(\omega)\mathrm{e}^{\mathrm{i}\omega\tau} \tag{2.9}$$

with the PSD

$$S(\omega) = \lim_{T\to\infty}\frac{1}{T}|\hat{x}_T(\omega)|^2 \tag{2.10}$$

$$= \int_{-\infty}^{\infty}\mathrm{d}\tau C(\tau)\mathrm{e}^{-\mathrm{i}\omega\tau} \tag{2.11}$$

Equation 2.9 is the Wiener-Khinchin theorem that states that the autocorrelation function $C(\tau)$ and the PSD $S(\omega)$ are Fourier-transform pairs [**Koopmans1995**]. Furthermore, defining the latter through Equation 2.10 gives us an intuitive picture of the PSD if we recall Parseval's theorem,

$$\int_{-\infty}^{\infty}\frac{\mathrm{d}\omega}{2\pi}\frac{1}{T}|\hat{x}_T(\omega)|^2 = \frac{1}{T}\int_{-\infty}^{\infty}\mathrm{d}t\,|x(t)|^2. \tag{2.12}$$

That is, the total power P contained in the signal $x(t)$ is given by integrating over the PSD. Similarly, the power contained in a band of frequencies $[\omega_1, \omega_2]$ is given by

$$P(\omega_1,\omega_2) = \mathrm{RMS}_S(\omega_1,\omega_2)^2 \tag{2.13}$$

$$= \int_{\omega_1}^{\omega_2}\frac{\mathrm{d}\omega}{2\pi}S(\omega) \tag{2.14}$$

where $\mathrm{RMS}_S(\omega_1,\omega_2)$ is the root mean square within this frequency band. These relations are helpful when analyzing noise PSDs to gauge the relative weight of contributions from different frequency bands to the total noise power.

Equation 2.10 represents the starting point for the experimental spectrum estimation procedure. Instead of a continuous signal $x(t), t \in [0, T]$, consider its discretized version[8]

8: We only discuss the problem of equally spaced samples here. Variants for spectral estimation of time series with unequal spacing exist [**Lomb1976**, **Scargle1982**].

$$x_n, \quad n \in \{0, 1, \dots, N-1\} \tag{2.15}$$

defined at times $t_n = n\Delta t$ with $T = N\Delta t$ and where $\Delta t = f_s^{-1}$ is the sampling interval (the inverse of the sampling frequency f_s). Invoking the ergodic theorem, we can replace the long-term average in Equation 2.10 by the ensemble average over M realizations i of the noisy signal $x_n^{(m)}$ and write

$$S_n = \frac{1}{M}\sum_{i=0}^{M-1}\left|\hat{x}_n^{(m)}\right|^2 \tag{2.16}$$
$$= \frac{1}{M}\sum_{i=0}^{M-1} S_n^{(m)} \tag{2.17}$$

where $\hat{x}_n^{(m)}$ is the discrete Fourier transform of $x_n^{(m)}$, we defined the *periodogram* of $x_n^{(m)}$ by

$$S_n^{(m)} = \left|\hat{x}_n^{(m)}\right|^2, \tag{2.18}$$

and S_n is an *estimate* of the true PSD sampled at the discrete frequencies $\omega_n = 2\pi n/T \in 2\pi \times \{-f_s/2\,,\dots,\,f_s/2\}$.[9] Equation 2.16 is known as Bartlett's method [**Bartlett1948**] for spectrum estimation.[10]

9: We blithely disregard integer algebra issues occuring here for conciseness and leave it as an exercise for the reader to figure out what the exact bounds of the set of ω_n are.

10: By taking the limit $M \to \infty$ one recovers the true PSD,

$$\lim_{M\to\infty} S_n = S(\omega_n).$$

The continuum limit is as always obtained by sending $\Delta t \to 0, N \to \infty, N\Delta t = \text{const.}$

To better understand the properties of this estimate, let us take a look at the parameters Δt, N, and M. The sampling interval Δt defines the largest resolvable frequency by the Nyquist sampling theorem,

$$f_{\text{max}} = \frac{f_s}{2} = \frac{1}{2\Delta t}. \tag{2.19}$$

In turn, the number of samples N determines the frequency resolution Δf, or smallest resolvable frequency,

$$f_{\text{min}} = \Delta f = \frac{1}{T} = \frac{1}{N\Delta t} = \frac{f_s}{N}. \tag{2.20}$$

Lastly, M determines the variance of the set of periodograms $\{S_n^{(m)}\}_{i=0}^{M-1}$ and hence the accuracy of the estimate S_n.

In practice, the ensemble realizations i are of course obtained sequentially, implying that one acquires a time series of data $x_n, n \in \{0, 1, \dots, NM-1\}$ and partitions these data into M sequences of length N. It becomes clear, then, that the Bartlett average (Equation 2.16) trades spectral resolution (larger N) for estimation accuracy (larger M) given the finite acquisition time $T = NM\Delta t$.

An improvement in data efficiency can be obtained using Welch's method [**Welch1967**]. To see how, we first need to discuss spectral windowing.

2.2 Window functions

Partitioning a signal x_n into M sections $x_n^{(m)}$ of length N is mathematically equivalent to multiplying the signal with the rectangular *window function* given by[11]

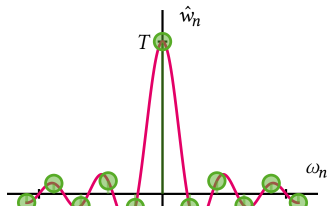
11: This window is also known as the boxcar or Dirichlet window.

$$w_n^{(m)} = \begin{cases} 1 & \text{if} \quad (m-1)N \leq n < mN \quad \text{and} \\ 0 & \text{else} \end{cases} \tag{2.21}$$

so that $x_n^{(m)} = x_n w_n^{(m)}$.

add $\omega_n^{(m)}$, scaled ticks

Now recall that multiplication and convolution are duals under the Fourier



transform, implying that

$$\hat{x}_n^{(m)} = \hat{x}_n * \hat{w}_n^{(m)}. \tag{2.22}$$

where the Fourier representation of the rectangular window[12]

12: $\operatorname{sinc}(x) = \sin(x)/x$.

$$\hat{w}_n^{(m)} = \hat{w}_n \mathrm{e}^{-\mathrm{i}(m-1/2)\omega_n T}, \tag{2.23}$$

$$\hat{w}_n = T \operatorname{sinc}\left(\frac{\omega_n T}{2}\right). \tag{2.24}$$

Figure 2.1 shows the unshifted rectangular window $\hat{w}_n$ in Fourier space. We can hence understand the Fourier spectrum of $x_n^{(m)}$ as sampling $\hat{x}_n$ with the probe $\hat{w}_n^{(m)}$. However, while in the continuum limit (*c.f.* sidenote 10) Equation 2.24 tends towards $\delta(\omega_n)$ and thus will produce a faithful reconstruction of the true spectrum, the finite frequency spacing Δf of discrete signals introduces a finite bandwidth of the probe as well as *sidelobes*. These effects induce what is known as *spectral leakage* [**Harris1978**, **Koopmans1995**] and lead to artifacts and deviations of the spectrum estimator S_n from the true spectrum $S(\omega_n)$.

For this reason, a plethora of *window functions* have been introduced to mitigate the effects of spectral leakage. Key properties of a window are the spectral bandwidth (center lobe width) and sidelobe amplitude between which there typically is a tradeoff.[13]

13: Wikipedia gives a good overview of existing window functions [**WindowFunctionWiki**].

add $\omega_n^{(m)}$, scaled ticks

A window frequently used in spectral analysis is the Hann window [**Nuttall1981**],

$$w_n^{(m)} = \begin{cases} \cos^2\left(\frac{\pi n}{N}\right) & \text{if} \quad (m-1)N \le n < mN \quad \text{and} \\ 0 & \text{else} \end{cases} \tag{2.25}$$

with the Fourier representation of the unshifted window,

$$\hat{w}_n = T \operatorname{sinc}\left(\frac{\omega_n T}{2}\right) \times \frac{1}{2(1-\omega_n T/2\pi)(1+\omega_n T/2\pi)}, \tag{2.26}$$

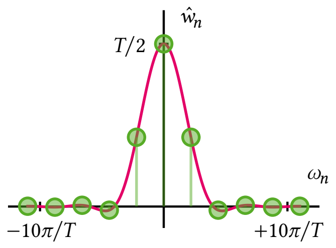


Figure 2.2: The Fourier representation of the Hann window in continuous time.

shown in Figure 2.2. The favorable properties of the Hann window are apparent when compared to the rectangular window in Figure 2.1; the sidelobes are quadratically suppressed while the center lobe is only slightly broadened.

Another favorable property of the Hann window is that $w_0^{(0)} = w_{N-1}^{(0)} = 0$. This suppresses detrimental effects arising from a possible discontinuity ($x_0^{(0)} \neq x_{N-1}^{(0)}$) at the edge of a data segment related to the discrete Fourier transform, which assumes periodic data.[14]

14: Although this can usually also be achieved by detrending the data before performing the Fourier transform, which is a good idea in any case.

2.3 Welch's method

Contemplating Equation 2.25, one might come to the conclusion that using a window such as this is not very data efficient in the sense that a large fraction of samples located at the edge of the window is strongly suppressed and hence does not contribute significantly to the spectrum estimate. To alleviate this lack of efficiency, one can introduce an overlap between adjacent data windows. That is, instead of partitioning the data x_n into M non-overlapping sections of length N, one shifts the mth window forward by $-mK$ with $K > 0$ the overlap. Finally, the periodogram

(Equation 2.18) is computed for each window and subsequently averaged to obtain the spectrum estimator (Equation 2.16).

This method of spectrum estimation is known as Welch's method [**Welch1967**]. One can show [**Welch1967**] that the correlation between the periodograms of adjacent, overlapping windows is sufficiently small to avoid a biased estimate. The overlap naturally depends on the choice of window; a typical value for the Hann window $K = N/2$ with which one would obtain $M = 2L/N - 1$ windows for data of length L.[15]

15: Again neglecting integer arithmetic issues.

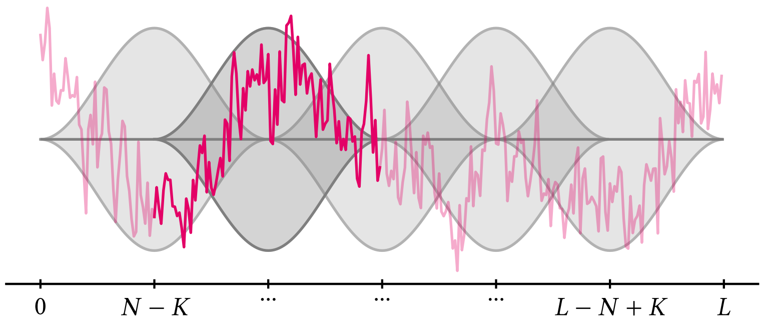


Figure 2.3: Illustration of Welch's method for spectrum estimation. The data (pink) of length L is partitioned into $K = 2L/N - 1$ segments of length N. Each segment is multiplied with a window function (gray) which reduces spectral leakage and other artifacts. A finite overlap K between adjacent windows (gray) ensures efficient sample use.

Figure 2.3 conceptually illustrates Welch's method for a trace of $1/f$ noise with $L = 300$ samples in total. Choosing the Hann window and an overlap of 50 % results in $M = 5$ segments for a window length of $N = 100$. The data in the second window is highlighted.

2.3.1 Parameters

We are now in a position to discuss how the various parameters of a time series relate to both to physical parameters of the resulting spectrum estimate and to each other. To this end, we will go through the typical procedure of acquiring a spectrum estimate using Welch's method chronologically.

To acquire data using some form of (digital) data acquisition device (DAQ), one usually needs to specify two parameters first: the total number of samples to be acquired, L, and the sample rate, f_s. This results in a measurement of duration $T = L\Delta t$ where $\Delta t = f_s^{-1}$ as previously mentioned. The choice of f_s already induces an upper bound on the first parameter characterizing the PSD estimate: the largest resolvable frequency $f_{\mathrm{max}} \leq f_s/2$ (*c.f.* Equation 2.19, but note that we allow f_{max} to be smaller than half the sample rate in anticipation of hardware constraints). Next, we choose a number of Welch averages, M, *i.e.*, data partitions, and their overlap, K. In doing so, one fixes the number of samples per partition N and thereby induces the lower bound on the second parameter characterizing the PSD estimate: the frequency spacing $\Delta f = 1/N \leq f_{\mathrm{min}}$ (*c.f.* Equation 2.20).[16]

Table 2.1: Overview of spectrum estimation parameters. The parameters can be assigned into three groups: 1. DAQ parameters configuring the data acquisition device, 2. Welch parameters specifying the periodogram averaging, and 3. Spectrum properties induced by the above.

1. DAQ parameters	
L	Total number of samples
f_s	Sample rate
2. Welch parameters	
K	Number of overlap samples
N	Number of segment samples
M	Number of Welch segments
3. Spectrum parameters	
f_{min}	Smallest resolvable frequency
f_{max}	Largest resolvable frequency

16: Technically, the smallest resolvable frequency in a fast Fourier transform (FFT) is zero, of course. But as data is typically detrended (a constant or linear trend subtracted) before computation of the periodogram, the smallest *meaningful* frequency is given by f_{min}.

D(A)G for parameter interdependencies?

The python_spectrometer software package 3

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`.

3.1 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 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 3.1: Variable names used in Chapter 2 and their corresponding parameter names as used in `python_spectrometer` and `scipy.signal.welch()` [**WelchScipy**].

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

3.1.1 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.* Subsection 2.3.1) 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 $[f_{\mathrm{min}}, f_{\mathrm{max}}]$ 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 3.1), 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 3.1[4] and attempts to resolve the parameter interdependencies lined out in Subsection 2.3.1 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 3.1. 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 3.1.

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

Listing 3.1: `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 3.2: Resolved settings for the same input parameters as in Listing 3.1 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 [**ScopeModuleZhinst**], the same requested settings would have resolved to those shown in Listing 3.2.[7] This is because the Scope module constrains $L \in [2^{12}, 2^{14}]$ and $f_s \in 60\,\mathrm{MHz} \times 2^{[-16,0]} \approx \{915.5\,\mathrm{Hz}, \dots, 30\,\mathrm{MHz}, 60\,\mathrm{MHz}\}$.

ScopeModuleZhinst

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

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 `Spectrometer`. 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 as representatively shown in Listing 3.3.

Listing 3.3: DAQ workflow pseudocode. A `MyDAQ` object (representing the instrument `My`) 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 PSD estimator in our example).

```
daq = MyDAQ(driver_handle)

parsed_settings = daq.setup(**user_settings)
acquisition_generator = daq.acquire(**parsed_settings)

for data_buffer in acquisition_generator:
    estimate_psd(data_buffer)
```

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.[9] An exemplary implementation of a `DAQ` subclass for a fictitious instrument is shown in Listing 3.4. In addition to the methods to configure the instrument and perform data acquisition, it is possible to override the `DAQSettings` property to implement instrument-specific hardware constraints such as, in this example, the number of samples per buffer being constrained to the discrete interval $[1, 2048]$. Leveraging the `qutil.domains` module, more complex constraints such as sample rates restricted to an internal clock rate divided by a power of two[10] can be specified.

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

9: *I. e.*, the number of time series data batches acquired, as opposed to the number of Welch averages `n_seg` within one batch.

10: *C.f.* the implementation of the AlazarTech ATS9440 digitizer card.

```python
# daq/mydaq.py
import dataclasses
from qutil.domains import DiscreteInterval
from .base import DAQ
from .settings import DAQSettings

@dataclasses.dataclass
class MyDAQ(DAQ):
    handle: mydriver.DeviceHandle

    @property
    def DAQSettings(self) -> type[DAQSettings]:
        class MyDAQSettings(DAQSettings):
            ALLOWED_NPERSEG = DiscreteInterval(1, 2048)
        return MyDAQSettings

    def setup(self, **settings) -> dict:
        settings = self.DAQSettings(settings)
        parsed_settings = settings.to_consistent_dict()
        self.handle.configure(parsed_settings)
        return parsed_settings

    def acquire(self, n_avg: int, *, **settings) -> Generator:
        self.handle.arm(n_avg)
        for _ in range(n_avg):
            self.handle.wait_for_trigger()
            yield self.handle.fetch()
        return self.handle.metadata
```

Listing 3.4: Exemplary code for a `DAQ` implementation of some instrument with given driver class `DeviceHandle` in the package `mydriver`. The `MyDAQ` class is instantiated with a `DeviceHandle` instance. Optionally, the `DAQSettings` property can be overridden to implement hardware constraints or default values for data acquisition parameters. For this, the `qutil.domains` module provides several classes that represent bounded domains and sets. The `setup()` method parses the given acquisition settings and configures the instrument through the external driver interface `handle.configure()`. The `acquire()` method arms the instrument (if necessary) and loops over the number of outer averages, `n_avg`. In the body of the loop, it can wait for external triggers (or send software triggers) before yielding a batch of data fetched from the external driver interface. Once acquisition is done, the method can return arbitrary metadata to the `Spectrometer` object to attach to the stored data.

3.1.2 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 Equation 2.16. However, it is often desirable to transform, or process, the data in 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,[11] 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.

11: 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 3.5 shows a simple function that accounts for the gain of an amplifier.

```python
def comp_gain(x, gain=1.0, **_):
    return x / gain
```

Listing 3.5: A simple `procfn`, which converts amplified data back to the level before amplification. More complex processing chains can concisely be defined with `qutil.functools.FunctionChain` that pipes the output of one function into the input of the next.

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()` [**WelchScipy**]. Here, the keyword `fourier_procfn` should be a callable with signature `(xf, f, **kwargs) -> (xfp, fp)`.[12] 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

WelchScipy

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

in case the function modifies the frequencies.[13] A simple example for a processing function in Fourier space is shown in Listing 3.6, which computes the (anti-)derivative of the data using the fact that

13: One example is the octave_band_rms() function from the qutil.signal_processing module [**OctaveBandRmsQutil**]

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

Listing 3.6: A simple fourier_procfn, which calculates the (anti-)derivative.

$$\frac{\partial^n}{\partial t^n} \xrightarrow{\text{F.T.}} (\mathrm{i}\omega)^n \tag{3.1}$$

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

3.2 Feature overview

überleitung

3.2.1 Serial 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,[14] 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, a Zurich Instruments MFLI.[15] Choosing the frequency bounds, say $f_{\mathrm{min}} = 10\,\mathrm{Hz}$ and $f_{\mathrm{max}} = 100\,\mathrm{kHz}$, and using the sensible defaults for the remaining spectrum parameters, Bob first grounds the input of his DAQ to record a *baseline* spectrum. Thus far, his code would hence look something like that shown in Listing 3.7, which produces the plot shown in Figure 3.1.

14: Let's call him Bob.

15: For the MFLI, DAQ subclasses for both the Scope and the DAQ module are implemented. The former works with data before and the latter with data after demodulation.

```python
from python_spectrometer import Spectrometer, daq
from qutil.functools import scaled

mfli_daq = daq.ZurichInstrumentsMFLIDAQ(session, device)
spect = Spectrometer(mfli_daq, procfn=scaled(1e6),
                     processed_unit='µV')

spect.take('baseline', f_min=1e1, f_max=1e5, daq='grounded')
```

Listing 3.7: Setup and serial 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. The procfn and processed_unit arguments help converting raw data into a more human-friendly unit.

After acquiring the baseline, he next ungrounds the DAQ to obtain a representative spectrum of the noise in an actual measurement. He then proceeds by tweaking things on his setup, testing out different parameters, *etc.* Every time he changes something, he acquires another spectrum using take(), labeling each with a meaningful comment for identification.

This leaves him with the spectrometer plot as shown in Figure 3.2. While working, Bob realizes he'd like see the signal in the time-domain as well. He easily achieves this by setting spect.plot_timetrace = True,

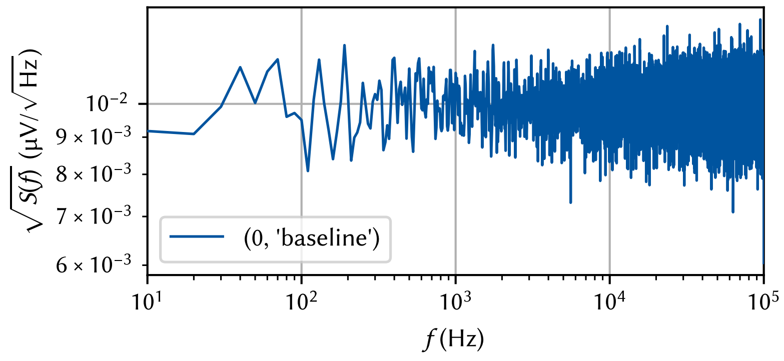


Figure 3.1: The `python_spectrometer` plot after acquiring the (here: synthetic) baseline spectrum. By default, the amplitude spectral density (ASD) = √PSD is displayed in the main plot. Each spectrum is assigned a unique identifier key consisting of an incrementing integer and the user comment, and can be refered to by either (or both) when interacting with the object.

```python
settings = {'f_min': 1e1, 'f_max': 1e5, 'daq': 'connected'}
spect.take('connected', **settings)
spect.take('lifted cable', cable='lifted', **settings)
spect.take('jumped', **settings)
```

Listing 3.8: Code to acquire additional spectra. Arbitrary key-value pairs can be passed to the `take()` method, which are stored as metadata if they do not apply to any functions downstream in the data processing chain.

which adds an oscilloscope subplot to spectrometer figure as shown in Figure 3.3.

Bob now observes that the noise spectra he has recorded display many sharp peaks in particular at high frequencies while the $1/f$ noise floor seems pretty consistent across different measurements. This makes it harder for him to evaluate whether any of his changes are actually an improvement or not. The `python_spectrometer` package allows addressing this by plotting the integrated spectra in another subplot. Bob's spectrometer figure after setting `spect.plot_cumulative = True` is shown in Figure 3.4. In the case that `spect.plot_amplitude == True`, this new subplot shows the root mean square (RMS) in the band $[f_{\mathrm{min}}, f]$,

$$\mathrm{RMS}_S(f) \equiv \mathrm{RMS}_S(f_{\mathrm{min}}, f), \tag{3.2}$$

and the band power (Equation 2.13) otherwise.

The cumulative RMS plot already helps, but Bob would like a more quantitative comparison of relative spectral powers. Therefore, he rescales the spectra in terms of their relative powers expressed in dB[16] by applying the following settings, which produces the plot shown in Figure 3.5:

16: Recall that the decibel is defined by the ratio L_P of two powers P_1, P_2 as [**Pozar2005**]

$$L_P = 10\log_{10}\left(\frac{P_1}{P_2}\right)\ \mathrm{dB}.$$

```python
spect.plot_dB_scale = True
spect.plot_amplitude = False
spect.plot_density = False
```

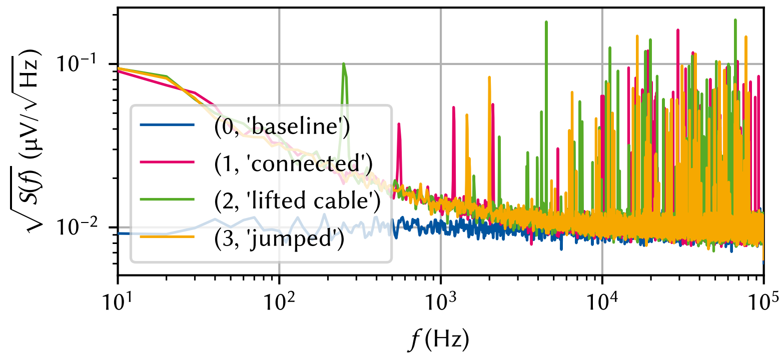


Figure 3.2: The `python_spectrometer` plot after acquiring additional (synthetic) spectra. Each spectrum is uniquely identified by a two-tuple of `(index, comment)`.

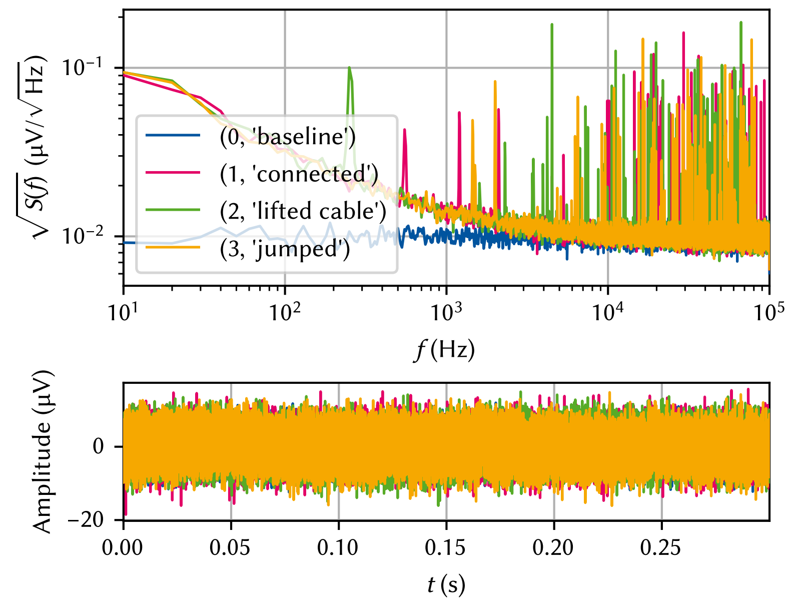


Figure 3.3: The `python_spectrometer` plot shown in Figure 3.2 when setting `spect.plot_timetrace = True`. This adds a subplot that shows the time series data from which the PSD was computed akin to what an oscilloscope would show. Note that this is the entire time series, *i. e.*, the data of length *L*, which is (by default, using Welch's method) segmented for spectrum estimation.

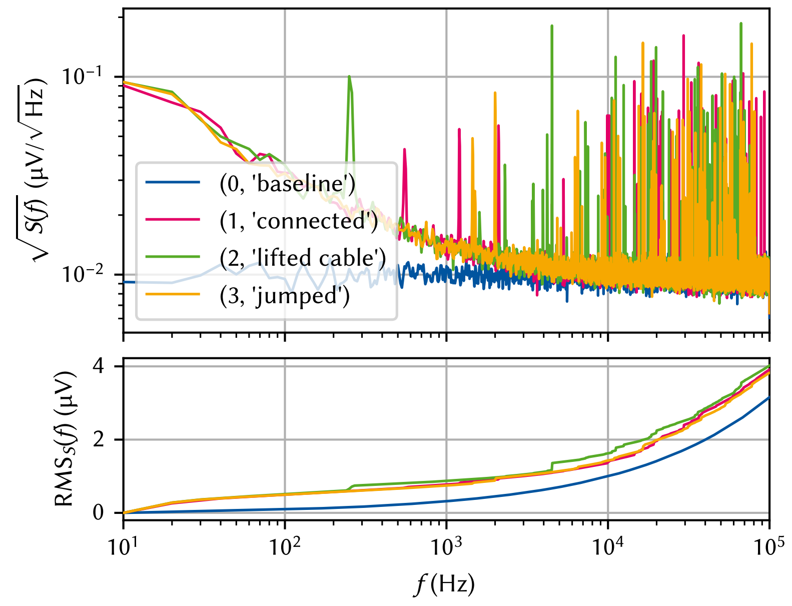


Figure 3.4: The `python_spectrometer` plot shown in Figure 3.2 when setting `spect.plot_cumulative = True`. This adds a subplot that shows the RMS (*c. f.* Equation 2.13) which can be helpful in evaluating the contribution of individual peaks in the spectrum to the total noise power. Both the oscilloscope subplot (Figure 3.3) and the RMS subplot can also be shown at the same time.

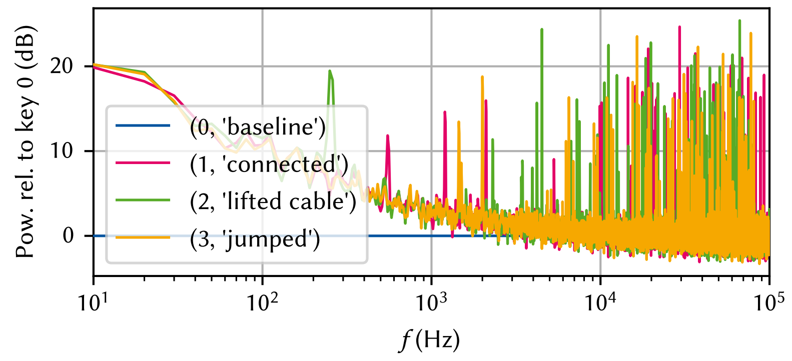


Figure 3.5: The `python_spectrometer` plot in relative mode. Starting from the state in Figure 3.2, we set `spect.plot_dB_scale = True` as well as `spect.plot_amplitude = False` and `spect.plot_density = False` to compare the relative noise powers with respect to the baseline.

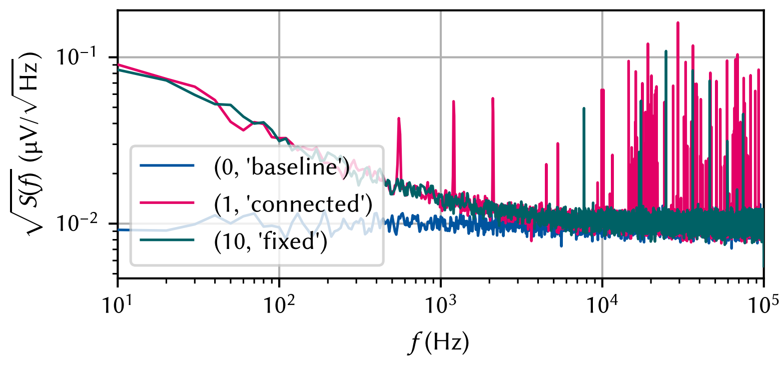


Figure 3.6: The `python_spectrometer` plot after multiple additional spectra were acquired and hidden. Hiding spectra that one is not interested in anymore is achieved through `spect.hide(*range(2, 10))`. This is reversed by `spect.show(*range(2, 10))`. Data can also be dropped (`spect.drop(key)`) or deleted (`spect.delete(key)`) from the internal cache and disk, respectively.

The attribute `plot_density` controls whether the *power spectral density* or the *power spectrum*.[17] Scaling the data to the power spectrum instead of the density, Bob can get an estimate of the RMS at a single frequency by reading off the peak height. Additionally displaying the data in dB then gives insight into relative noise powers of different spectra.

17: At this point, we *do* need to distinguish between the PSD and the power spectrum counter to sidenote 2 in Chapter 2. The PSD and power spectrum are related by the equivalent noise bandwidth (ENBW),

$$\text{Spectral density} \xrightarrow{\times\text{ENBW}} \text{Spectrum},$$

which is itself a function of the sampling rate and the properties of the spectral window [**Harris1978**],

$$\text{ENBW} = f_s \frac{\sum_n \hat{w}_n^2}{\left[\sum_n \hat{w}_n\right]^2}. \qquad (3.3)$$

Bob carries on with his enterprise and continues to acquire spectra until, finally, he finds the source of his noise! Alas, his spectrometer plot is now overflowing with plotted data when really he just wants to compare the baseline, the original, noisy state, and the final, clean spectrum. He simply calls

```
spect.hide(*range(2, 10))
```

to hide the eight spectra of unsuccessful debugging, leaving him with a plot as shown in Figure 3.6.

Finally, happy with the results, Bob serializes the state of the spectrometer to disk, allowing him to pick up where he left off at a later point in time:

```
spect.serialize_to_disk('2032-12-24_noise_hunting')
```

The next week, Bob is asked by his team about his progress on debugging the noise in their setup. Even though he is working from home that day and does not have access to the lab computer, Bob simply uses his laptop computer and pulls up the `Spectrometer` session stored on the server, allowing them to interactively discuss the spectra:

```
file = '2032-12-24_noise_hunting'
# Read-only instance because no DAQ attached
spect = Spectrometer.recall_from_disk(savepath / file)
```

This opens up the plot shown in Figure 3.6 again. While they cannot acquire new spectra in this state,[18] they can still use all the plotting features like showing or hiding spectra, or changing plot types as discussed above.

18: They could of course always attach a `DAQ` instance to the spectrometer and continue as they were.

3.2.2 Live spectrum acquisition

Manually recording spectra in the workflow outlined in Subsection 3.2.1 becomes tedious at some point, and experimenters tend to become negligent with keeping metadata and comments up to date as they continue to change settings. Once a certain number of spectra has been obtained, the spectrometer plot also becomes crowded, and hiding old spectra manually is cumbersome. Moreover, each time a spectrum is captured, data is saved to disk, potentially accruing large amounts of disk space. For these

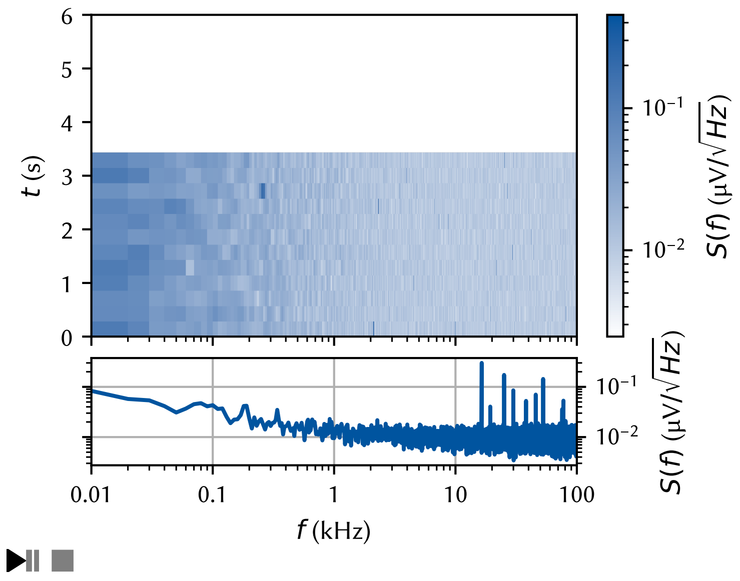


Figure 3.7: Spectrometer live view. The bottom plot shows the most recently acquired spectrum, while the top plot shows a waterfall plot of the most recent ones. Data acquisition runs in a background thread, keeping the interpreter responsive and available to interact with instruments, for example. The icons in the bottom left and right corner allow interacting with the live view.

reasons, the `python_spectrometer` package also offers a non-persistent live mode for displaying spectra continuously.

This mode is facilitated by the `qutil.plotting.live_view` module that provides asynchronous plotting functionality based on `matplotlib`. The `live_view` module supports both the multithreading and multiprocessing paradigms for concurrency in order to keep the interpreter responsive. In the former case, the window hosting the figure runs in the main thread and is kept responsive using `matplotlib`'s GUI event loop mechanisms. In the latter, the plotting takes place in a separate process, resulting in true parallelism.

The live mode is started with the `Spectrometer.live_view()` method. Data is continuously acquired[19] in a background thread using the same `DAQ` interface as the serial mode. Instead of saving the data on disk and managing plotting from `python_spectrometer`, however, the data is fed into a queue.[20] A `live_view.IncrementalLiveView2D` object then retrieves the data from the queue and handles plotting in a GUI event loop. To start a spectroscopy session with the same parameters as in Subsection 3.2.1 with a given `Spectrometer` object (*c.f.* Listing 3.7), we would call

19: Technically, a very large n_avg is used.

20: A queue is a concurrency mechanism for exchanging data between multiple threads or processes.

```
view = spect.live_view(f_min=1e1, f_max=1e5, in_process=True)
```

which would open a figure window such as that shown in Figure 3.7. Similar to `take()`, the data acquisition parameters are passed to `live_view()` as keyword arguments.[21] The `in_process` argument specifies if multiprocessing (`True`) or multithreading (`False`) is used. Dictionaries with customization parameters for the `live_view` object can further be passed to the method.

21: The difference is, of course, that we do not need to specify a comment since no data is retained.

Conclusion and outlook 4

In this part, I presented the `python_spectrometer` Python package for interactive, backend-agnostic spectrum acquisition.

Outlook. There are several possible avenues for future development of the `python_spectrometer` package. An obvious case is adding support for more DAQ hardware instruments by implementing `DAQ` interfaces. Modular devices such as those offered by QBLOX[1] and Quantum Machines[2] are on track to become the new standard in quantum technology labs. Implementing drivers for these instruments would benefit the adoption of both the instruments and the `python_spectrometer` package.

1: https://www.qblox.com/research

2: https://www.quantum-machines.co/

Next, incorporating noise spectroscopy into the standard measurement workflow of quantum device experiments would allow experimentalists to quickly gauge noise levels as they are performing measurements. If for some reason the noise changed[3] the experimentalist could quickly obtain insight into the noise by analyzing the spectrum. In a client-server architecture, which is inherently asynchronous, such as Zurich Instrument's LabOne[4] software, this is already possible using the web interface. But of course the strength of the `python_spectrometer` package stems from its capacity to be utilized in conjunction with any hardware instrument.

3: As it happens often, unfortunately.

4: https://www.zhinst.com/ch/en/instruments/labone/labone-instrument-control-software

One way to implement such functionality would be to introduce a proxy `DAQ` subclass to be used together with the live mode presented in Subsection 3.2.2. This proxy class would serve as an interface to external measurement software and expose two attributes; first, a data queue, into which the external code could place arbitrary time series data that was obtained during some measurement, and second, a shared dictionary to hold acquisition parameters as these might change between measurements. Because the live view mode runs in the background, the external measurement framework could push data to the queue whenever new data was taken without obstructing the measurement workflow.

```
# daq/tee.py
import dataclasses
import threading
import multiprocessing as mp

from .base import DAQ

@dataclasses.dataclass
class TeeDAQ(DAQ):
    settings:
      ↪ mp.managers.DictProxy
    data_queue: mp.JoinableQueue
    stop_event: threading.Event

    def setup(self, **_):
        settings = self.⌋
          ↪ DAQSettings(self.⌋
          ↪ settings)
        return settings.⌋
          ↪ to_consistent_dict()

    def acquire(self, **_):
        while not self.⌋
          ↪ stop_event.is_set():
            yield self.⌋
              ↪ data_queue.⌋
              ↪ get(block=True)
```

Listing 4.1: Template design for a proxy `DAQ` implementation to stream noise spectra from an external measurement framework. The `settings` attribute is a dictionary proxy shared between processes and used to pass acquisition parameters from the measurement framework to `python_spectrometer`.

Listing 4.1 shows a template design for such a `TeeDAQ` class. The `setup()` method ignores the input parameters and instead obtains the current settings from the shared `settings` proxy. Similarly, instead of fetching data from an instrument itself, the `acquire()` method attempts to fetch data from the shared `data_queue` and blocks the thread if no data is present, thereby efficiently idling and consuming no resources unless triggered by the external caller. A measurement framework would then interact with the `TeeDAQ` object as exemplarized by the following code:

```
daq = TeeDAQ(...)
spect = Spectrometer(daq)
view = spect.live_view()
...
data = measure(fs, n_pts)
daq.settings.update(fs=fs, n_pts=n_pts)
daq.data_queue.put(data)
```

Measurement frameworks integrating with this interface could thus provide experimentalists live feedback on current noise levels with negligible overhead and minimal code adaptation.

Finally, it might be useful to not only allow estimating PSDs but also cross power spectral densitys (CSDs) or *cross-spectra*. The cross-spectrum[5] is

5: Again, we use the two terms interchangably unless otherwise indicated, see sidenote 2 in Chapter 2.

the Fourier transform of not the auto-correlation but the cross-correlation function $C(\tau)$ (*c.f.* Equation 2.3) between two random processes. Take a set of processes $\{x_1(t), x_2(t), \ldots, x_n(t)\}$ that correspond to noise measured at different locations in a sample. The cross-correlation function between variables x_i and x_j is then given by[6]

6: Again assuming wide-sense stationary processes.

$$C_{ij}(\tau) = \left\langle x_i(t)^* x_j(t+\tau) \right\rangle . \tag{4.1}$$

This function (and its Fourier pair the cross-spectrum $S_{ij}(\omega)$) quantifies the degree of correlation between noise at site i and noise at site j. Unlike the *auto*-spectrum (or self-spectrum), the cross-spectrum is always a complex quantity, even for real $x_i(t)$. It is not hard to see that for quantum processors, for example, these kinds of correlations could have significant impact on operation, and on error correction in particular [**Aharonov2006**, **Nickerson2019**, **Clader2021**]. To incorporate cross-spectra in the `python_spectrometer` package, only small changes should be necessary.

First, the data acquisition logic would need to be adapted. Two possible routes suggest themselves here; first, specialized `DAQ` classes could be implemented that, in place of yielding one batch of time series data, yield two batches each time they are queried. This approach first of all requires instruments with multiple channels,[7] which is not necessarily given. Furthermore, it would incur additional coding efforts by having to re-implement each `DAQ` class for cross-spectra. On the other hand, it would arguably make synchronization between channels easier to achieve.

7: If one sticks to single `DAQ` instances managing single instruments.

A less involved path would adapt the `Spectrometer` class to work with multiple `DAQ`s. This would not involve additional driver work[8] and allow the `Spectrometer` object to ensure synchronization between the different `DAQ`s. The internal workflow shown in Listing 3.3 would then need to be slightly adapted to the code shown in Listing 4.2. The downside of this approach is that synchronization of different instruments or channels would need to be taken care of externally.

8: Except possibly ensuring thread safety and timing synchronization if multiple `DAQ` objects communicate with the same physical instrument.

Listing 4.2: Proposed `DAQ` workflow for estimating cross-spectra. Each hardware channel (same or different instruments) is assigned to a `DAQ` object. After instrument configuration, it is asserted that the parameters match. Finally, data is fetched from both channels and fed into a CSD estimator. Note that triggering would need to be implemented externally.

```python
daq_1 = MyDAQ(driver_handle, channel=1)
# Or MyOtherDAQ(driver_handle_2) if another instrument
daq_2 = MyDAQ(driver_handle, channel=2)
daqs = (daq_1, daq_2)

parsed_settings = [daq.setup(**user_settings) for daq in daqs]
assert all_equal(parsed_settings), "DAQ settings do not match"

acquisition_generators = [daq.acquire(**parsed_settings[0])
                          for daq in daqs]
for data_buffers in zip(*acquisition_generators):
    estimate_csd(*data_buffers)
```

Further code adaptations would involve minor changes such as replacing the spectrum estimator with `scipy.signal.csd()` [**CSDScipy**] for CSD estimation[9] and make the plotting conform to complex data.[10]

CSDScipy

9: In practice, working with the normalized CSD, or correlation coefficient [**Rojas-Arias2023**, **Yoneda2023**]

$$r_{ij}(\omega) = \frac{S_{ij}(\omega)}{\sqrt{S_i(\omega)S_j(\omega)}} \tag{4.2}$$

with $S_i(\omega)$ the PSD of process x_i would likely be more favorable.

10: It could be worthwile to add a subplot to display both the magnitude and phase of the complex quantity $S_{ij}(\omega)$.

improve this part.

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 Unital Quantum Operations

Appendix

Special Terms

A

API Application Programming Interface. 14
ASD amplitude spectral density. 15

C

CSD cross power spectral density. 19, 20

D

DAQ data acquisition device. 9, 11, 12, 14, 19

E

ENBW equivalent noise bandwidth. 17

F

FFT fast Fourier transform. 9

P

PSD power spectral density. 5–7, 9, 12, 13, 15–17, 19, 20

R

RMS root mean square. 6, 15–17

T

TIA transimpedance amplifier. 13
TLF two-level fluctuator. 5