Use this document as a template

My PhD Thesis

Customise this page according to your needs

Tobias Hangleiter*

April 22, 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

1 Introduction

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 exist various methods for estimating the properties of noise in a classical signal $x(t)$.[1] A simple metric quantifying the average noise amplitude of the signal observed from time $t = 0$ to $t = T$ is the root mean square (RMS) [**RMSMathworld**],

$$\mathrm{RMS}_x = \sqrt{\frac{1}{T}\int_0^T \mathrm{d}t\, x(t)^2}. \tag{2.1}$$

While this already tells us *something* about the noise, it is evident that a single number does not provide many clues if we were to attempt to mitigate the noise or say something qualitative about it beyond "small" and "large". In cases such as these, physics has often turned to the *spectral* representation of the function of interest. Knowing the frequency content of a function gives access to a wealth of information about the underlying contributing processes. But how can we learn how a system behaves as a function of frequency?

Consider an electrical black box – some device under test (DUT) – with two leads connected to a lock-in amplifier (LIA) as sketched in **??**. Assuming the DUT is conducting, we could simply measure the conductance $G(t) = I(t)/V(t)$ through the device for some time T with a given lock-in modulation frequency f_i, subtract the constant offset,[2] and calculate the RMS using **??**. Repeating this procedure for different modulation frequencies $\{f_i\}_i$, we would collect a set of RMS-values that we could assign to the modulation frequencies at which they were measured and thus sample the noise amplitude spectrum[3] of the conductance, $S_G(f_i)$.[4] This method has the advantage that we can choose the frequencies at which the spectrum should be sampled—in particular they do not have to be evenly spaced. However, it has two significant shortcomings. First, it is rather inefficient and therefore time-consuming. For N frequency sample points, it takes a total time of NT to acquire all data, where T needs to be chosen such that the variance of RMS_G is sufficiently small. Second, and crucially, it is not always possible to excite the system at a certain frequency and measure its response. For example, it is generally considered hard[5] to – deliberately – excite vibrations of a specific frequency in a cryostat. A related method is available if one has access to a frequency-tunable *probe*, that is, a physical system whose behavior – typically its time evolution or some measurable property after letting the system evolve – under noise is well understood within a given noise model, and that can be controlled to be sensitive to a certain frequency. One example for such methods is dynamical decoupling (DD)-based noise spectroscopy using a qubit [**Alvarez2011**, **Szankowski2017**, **Dial2013**, **Connors2022**], a protocol based on insights gained from the filter-function formalism—see **??**. These protocols are especially well-suited for high but less so for low frequencies as they rely on observing the coherence of a qubit and hence the visibility on long time scales.

In a sense the most straight-forward way of measuring the noise spectrum presents itself as a consequence of the Wiener-Khinchin theorem [**Wiener1930**, **Khintchine1934**] that relates the noise spectrum to the Fourier transform of a time-dependent signal. This makes it possible to estimate the noise spectrum by simply observing a signal for a certain amount of time and performing some post-processing![6] This

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 1 and 2, for example.

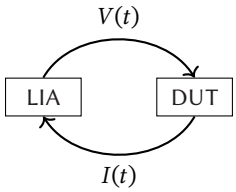

Figure 2.1: Measuring the conductance through a device under test (DUT) using a lock-in amplifier (LIA).

2: The constant part $G_0 = G(t) - \delta G(t)$ of course also holds some information about the system, for example about its bandwidth.

3: We will properly define this term below.

4: See **??** for a discussion on how the RMS at a certain frequency relates to other quantities discussed in this part.

5: And also unpopular with colleagues.

method will be the topic of the present chapter. It is laid out as follows. I will first derive the Wiener-Khinchin theorem for continuous processes and introduce the central quantity of interest in noise spectroscopy; the power spectral density (PSD). Then, I will discretize the method and discuss resulting side-effects before describing an efficient way of obtaining the spectrum from a finite amount of data. Finally, I elaborate on relevant parameters and properties.

2.1 Spectrum estimation from time series

To see how spectral noise properties may be estimated from time-series data, consider a continuous wide-sense stationary[7] 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[8]

$$\hat{x}_T(\omega) = \int_0^T \mathrm{d}t\, x(t) e^{-i\omega t} \tag{2.2}$$

$$\text{and} \quad x(t) = \int_{-\infty}^{\infty} \frac{\mathrm{d}\omega}{2\pi} \hat{x}_T(\omega) e^{i\omega t}, \tag{2.3}$$

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

$$C(\tau) = \langle x(t)^* x(t+\tau) \rangle \tag{2.4}$$

$$= \lim_{T\to\infty} \frac{1}{T} \int_0^T \mathrm{d}t\, x(t)^* x(t+\tau), \tag{2.5}$$

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 (**??**) and reordering the integrals, we get[9]

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

$$= \lim_{T\to\infty} \frac{1}{T} \int_{-\infty}^{\infty} \frac{\mathrm{d}\omega}{2\pi} \int_{-\infty}^{\infty} \frac{\mathrm{d}\omega'}{2\pi} \hat{x}_T(\omega)^* \hat{x}_T(\omega') e^{i\omega'\tau} \int_0^T \mathrm{d}t\, e^{it(\omega'-\omega)} \tag{2.7}$$

The innermost integral approaches a δ-function for large T,[10] allowing us to further simplify this under the limit as

$$C(\tau) = \lim_{T\to\infty} \frac{1}{T} \int_{-\infty}^{\infty} \frac{\mathrm{d}\omega}{2\pi} \int_{-\infty}^{\infty} \frac{\mathrm{d}\omega'}{2\pi} \hat{x}_T(\omega)^* \hat{x}_T(\omega') e^{i\omega'\tau} \delta(\omega' - \omega) \tag{2.8}$$

$$= \lim_{T\to\infty} \frac{1}{T} \int_{-\infty}^{\infty} \frac{\mathrm{d}\omega}{2\pi} |\hat{x}_T(\omega)|^2 e^{i\omega\tau} \tag{2.9}$$

$$= \int_{-\infty}^{\infty} \frac{\mathrm{d}\omega}{2\pi} S(\omega) e^{i\omega\tau} \tag{2.10}$$

with the power spectral density (PSD)[11]

$$S(\omega) = \lim_{T\to\infty} \frac{1}{T} |\hat{x}_T(\omega)|^2 \tag{2.11}$$

$$= \int_{-\infty}^{\infty} \mathrm{d}\tau C(\tau) e^{-i\omega\tau} \tag{2.12}$$

7: For a wide-sense stationary (also called weakly stationary) process $x(t)$, the mean is constant and the autocorrelation 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 [7]. The property further implies that $C(\tau)$ is an even function.

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

9: 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.

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

11: 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 **??** in **??**.

?? is the Wiener-Khinchin theorem [**Wiener1930**, **Khintchine1934**] that states that the autocorrelation function $C(\tau)$ and the PSD $S(\omega)$ are Fourier-transform pairs [7]. Furthermore, defining the latter through ?? gives us an intuitive picture of the PSD if we recall Parseval's theorem,

$$\int_{-\infty}^{\infty} \frac{\mathrm{d}\omega}{2\pi} \frac{1}{T} |\hat{x}_T(\omega)|^2 = \frac{1}{T} \int_{-\infty}^{\infty} \mathrm{d}t\, |x(t)|^2. \tag{2.13}$$

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

$$P(\omega_1, \omega_2) = \mathrm{RMS}_S(\omega_1, \omega_2)^2 \tag{2.14}$$

$$= \int_{\omega_1}^{\omega_2} \frac{\mathrm{d}\omega}{2\pi} S(\omega) \tag{2.15}$$

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.

To become familiar with the quantities $C(\tau)$ and $S(\omega)$, consider the Ornstein-Uhlenbeck process [8], the only stationary Gaussian Markovian stochastic process [9]. The autocorrelation function of the Ornstein-Uhlenbeck process is given by

$$C(\tau) = \sigma^2 e^{-\tau/\tau_c}, \tag{2.16}$$

with σ the RMS and τ_c the correlation time of the process. The PSD in turn is the Lorentzian function

$$S(\omega) = \frac{2\sigma^2 \tau_c}{1 + (\omega\tau_c)^2}. \tag{2.17}$$

For a given discretization time step Δt and hence bandwidth $\omega \in [0, \pi f_s]$, the Ornstein-Uhlenbeck process interpolates between perfectly uncorrelated, white noise ($\Delta t/\tau_c \to 0, S(\omega) = \text{const.}$) and correlated, pink noise ($\Delta t/\tau_c \to \infty, S(\omega) \propto \omega^{-2}$). ?? depicts simulated data and its autocorrelation function and PSD for exemplary parameters: in the white noise limit ($\tau_c = 10^{-2}\Delta t$, blue), in the intermediate regime ($\tau_c = 1\Delta t$, magenta), and in the correlated limit ($\tau_c = 10^2 \Delta t$, green). From the time series plot at the top it becomes clear that the RMS alone is insufficient to describe the properties of noisy signals as the curves differ significantly despite being normalized to their RMS. The autocorrelation functions averaged over 10^3 realizations of the noisy signals as well as their theoretical (continuous) value, ??, are plotted in the middle panel, normalized to $\tau_c = \Delta t$ and $\sigma^2 = 1/4$. For the white noise limit (blue), correlations are too short to be resolved with the given time discretization. The correlations decay to e^{-1} at $\tau/\tau_c = 10^{-2}, 1, 10^2$, respectively. Finally, the bottom panel shows the PSD, ??, and its periodogram estimate, again averaged over 10^3 realizations of the signal and normalized to $\tau_c = \Delta t$ and $\sigma^2 = 1/4$. The cross-over from white to pink PSD occurs at $\omega = \tau_c$. While the simulated data for $\tau_c = 10^{-2}\Delta t$ appears perfectly white, that for $\tau_c = 10^2 \Delta t$ appears perfectly $1/f$-like because the spectrum is only white below the smallest resolvable frequency $\Delta f = T^{-1}$.

Having gotten an intuition for the quantities $C(\tau)$ and $S(\omega)$, let us move on to see how the latter may be obtained from time-series data. ?? represents

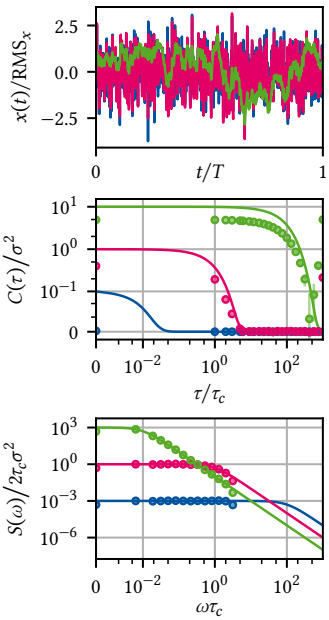

Figure 2.2: Ornstein-Uhlenbeck process Simulated time traces (top), autocorrelation function (middle), PSD (bottom) of the Ornstein-Uhlenbeck process. Top: Simluated time traces using the algorithm presented in ??. The data are normalized to the computed RMS (equal to σ in the continuous case). Middle: Theoretical autocorrelation function (??, solid lines) and computed from the simulated data averaged over 10^3 traces (circles, subset of points). Error bars indicate the standard error of the mean and axes are scaled with respect to the parameters of the magenta data, and data are plotted on an asinh-scale. Bottom: Theoretical PSD (??, solid lines) and periodograms computed from the simulated data averaged over 10^3 traces using `scipy.signal.periodogram()`, *cf.* ?? (circles, subset of points). Axes are again scaled with respect to the parameters of the magenta data and plotted on an asinh-scale. Parameters are $\tau_c = \Delta t \times \{10^{-2}, 1, 10^2\}$ and $\sigma^2 = \sqrt{\tau_c}/4$ for blue, magenta, and green data, respectively.

the starting point for the experimental spectrum estimation procedure. Instead of a continuous signal $x(t), t \in [0, T]$, consider its discretized version[12]

$$x_n, \quad n \in \{0, 1, \dots, N-1\} \tag{2.18}$$

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 **??** by the ensemble average over M realizations i of the noisy signal $x_n^{(m)}$ and write

$$S_n = \frac{1}{M} \sum_{i=0}^{M-1} \left| \hat{x}_n^{(m)} \right|^2 \tag{2.19}$$

$$= \frac{1}{M} \sum_{i=0}^{M-1} S_n^{(m)} \tag{2.20}$$

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

$$S_n^{(m)} = \left| \hat{x}_n^{(m)} \right|^2, \tag{2.21}$$

and S_n is an *estimate* of the true PSD sampled at the discrete frequencies $\omega_n = 2\pi n/T \in 2\pi \times \{-f_s/2, \dots, f_s/2\}$.[13] **??** is known as Bartlett's method [12] for spectrum estimation.[14]

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,

$$f_{\max} = \frac{f_s}{2} = \frac{1}{2\Delta t}. \tag{2.22}$$

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

$$f_{\min} = \Delta f = \frac{1}{T} = \frac{1}{N\Delta t} = \frac{f_s}{N}. \tag{2.23}$$

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 (**??**) 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 [13]. 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[15]

12: We only discuss the problem of equally spaced samples here. Variants for spectral estimation of time series with unequal spacing exist [10, 11].

13: 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.

14: 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 $\Delta t \to 0, N \to \infty, N\Delta t = \text{const}$.

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

$$w_n^{(m)} = \begin{cases} 1 & \text{if} \quad (m-1)N \leq n < mN \quad \text{and} \\ 0 & \text{else} \end{cases} \tag{2.24}$$

so that $x_n^{(m)} = x_n w_n^{(m)}$. Now recall that multiplication and convolution are duals under the Fourier transform, implying that

$$\hat{x}_n^{(m)} = \hat{x}_n * \hat{w}_n^{(m)}, \tag{2.25}$$

where the Fourier representation of the rectangular window[16]

$$\hat{w}_n^{(m)} = \hat{w}_n e^{-i(m-1/2)\omega_n T}, \tag{2.26}$$

$$\hat{w}_n = T\,\mathrm{sinc}\left(\frac{\omega_n T}{2}\right). \tag{2.27}$$

?? 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 (??) ?? tends towards $\delta(\omega_n)$ and thus will produce a faithful reconstruction of the true spectrum, the finite frequency spacing Δf of discrete signals and finite observation length T introduce a finite bandwidth of the probe as well as *sidelobes*. These effects induce what is known as *spectral leakage* and *scalloping loss* [7, 14] 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.[17] A window frequently used in spectral analysis is the Hann window [16],

$$w_n^{(m)} = \begin{cases} \cos^2\left(\frac{\pi n}{N}\right) & \text{if} \quad (m-1)N \leq n < mN \quad \text{and} \\ 0 & \text{else} \end{cases} \tag{2.28}$$

with the Fourier representation of the unshifted window,

$$\hat{w}_n = T\,\mathrm{sinc}\left(\frac{\omega_n T}{2}\right) \times \frac{1}{2(1-\omega_n T/2\pi)(1+\omega_n T/2\pi)}, \tag{2.29}$$

shown in ??. The favorable properties of the Hann window are apparent when compared to the rectangular window in ?? and ??; 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.[18]

2.3 Welch's method

Contemplating ??, 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

16: $\mathrm{sinc}(x) = \sin(x)/x$.

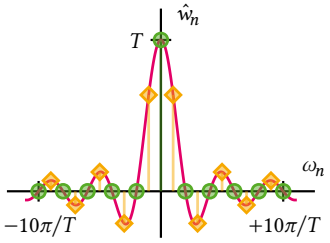

Figure 2.3: The Fourier representation of the rectangular window in continuous time (solid line) and for discrete frequencies $\omega_n = 2\pi n/T$ (circles). Introducing a phase shift, that is, shifting the window with respect to the signal in time, effectively shifts $\omega_n \to \omega_{n+\eta}$ as indicated for $\eta = 1/2$ (diamonds). This incurs scalloping loss.

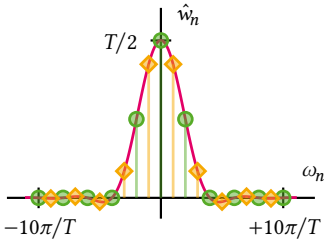

Figure 2.4: The Fourier representation of the Hann window in continuous time (solid line) and for discrete frequencies ω_n (circles). Diamonds indicate discrete sampling when the window completely out of phase with the signal (*cf.* ??).

17: Wikipedia gives a good overview of existing window functions [15].

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

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 (**??**) is computed for each window and subsequently averaged to obtain the spectrum estimator (**??**).

This method of spectrum estimation is known as Welch's method [13]. One can show [13] 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.[19]

19: Again neglecting integer arithmetic issues.

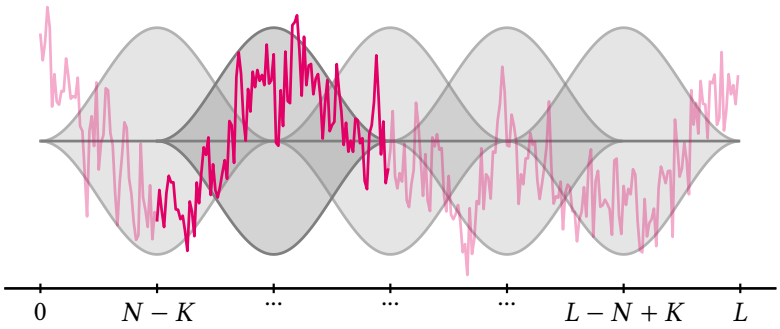

Figure 2.5: Illustration of Welch's method for spectrum estimation. The data (pink) of length L is partitioned into $K = 2L/N-1$ segments of length N. Each segment is multiplied with a window function (gray) which reduces spectral leakage and other artifacts. A finite overlap K between adjacent windows (gray) ensures efficient sample use.

?? 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.

Table 2.1: Overview of spectrum estimation parameters. The parameters can be assigned into four groups: 1. DAQ parameters configuring the data acquisition device, 2. Welch parameters specifying the periodogram averaging, 3. Spectrum properties induced by the above, and 4. External parameters unrelated to the others.

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
4. Miscellaneous parameters	
O	Number of outer averages

2.4 Parameters & Properties of the PSD

We are now in a position to discuss how the various parameters of a time series relate to both the 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 (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 $f_{max} \leq f_s/2$ (*cf.* **??**, 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 $\Delta f = 1/N \leq f_{min}$ (*cf.* **??**). Finally, we can introduce a number of *outer* averages, O, that is, the number of data batches that are acquired. While not directly related to Welch's method, choosing $O > 1$ can, for instance, help achieve a certain variance in the number of samples per batch, L, is limited by the data acquisition hardware, or simply allow for updating the spectrum estimate as data is being acquired. **??** shows

2: Technically, the smallest resolvable frequency 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_1.

the relationships of the various parameters among each other. In **??**, I lay out how these inter-dependencies are implemented in software.

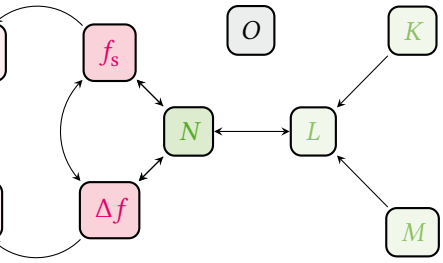

Figure 2.6: Relationships of data acquisition parameters (*cf.* **????**, with arrows indicating dependencies. N, f_s, and Δf are the central quantities defining the estimated spectrum's properties. From f_s and Δf follow (bounds for) f_{max} and f_{min}. From N, together with K and M, follows L, the total number of samples per data batch.

To conclude this chapter, let us discuss some of the properties of stochastic processes and their autocorrelation function and PSD. Consider again the process $x(t)$. We say $x(t)$ is *Gaussian* if $x(t) \sim \mathcal{N}(\mu, \sigma^2)\ \forall t$, meaning that the value of $x(t)$ at a given point in time follows a normal distribution with some mean μ and variance σ^2 over multiple realizations of the process. In this case, its statistical properties are fully described by the autocorrelation function $C(\tau)$ and PSD $S(\omega)$. This is because only the first two cumulants of a Gaussian distribution are nonzero [**Fox1978**]. 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].[21] Even if $x(t)$ 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. Let us just note at this point that techniques to estimate higher-order spectra (or *polyspectra*) exist [**Chandran1994**, **Norris2016**, **Szankowski2017**].

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 LIAs after demodulation in turn have asymmetric, two-sided PSDs. In this chapter so far, we have implicitly employed the two-sided definition, but in the software package presented in **??**, two-sided spectra are used only for complex data.

21: As an example, consider electronic devices, where voltage noise is thought to arise 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 [4, 5] (at least for a large density [6]).

The `python_spectrometer` software package

3

In this chapter, I introduce the `python_spectrometer` Python package[1] and lay out its design and functionality. This software package was developed to make it easier for experimentalists to transfer the mathematical machinery introduced in **??** to the lab. While in principle the entire process of spectrum estimation from a given array of time series data is already covered by the `welch()` routine in `SciPy` [17], obtaining the data array is not standardized. Different DAQ instruments have different capabilities, both on the hardware and the software level, and different driver interfaces to communicate with them. This implies custom data acquisition code is required for every instrument, introducing a significant entry barrier to spectral analysis. The `python_spectrometer` package implements a simple interface to different hardware instruments that allows for changing the hardware backend without having to adapt the user-facing code and also incorporates different hardware constraints.

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

[17]: (n.d.), *`welch` — SciPy v1.15.2 Manual*

What is more, noise spectroscopy tends to be a visual endeavor in practice; it is hard to compare different noise spectra based on quantitative reasoning alone. Data visualization is hence an integral part of noise spectroscopy, but plotting is not just plotting. Do we want the data to be shown on a log-log scale?[2] Do we want to show the relative magnitude of different data sets? Do we want to inspect the time traces as well? The `python_spectrometer` package addresses these questions by allowing users to interactively change features of the main plot window to adapt it to the form best suited to the situation at hand.

2: The short answer is yes, but it comes with visual side-effects that demand other ways of plotting data at times. The long answer is therefore yes, *and* ...

Moreover, when concerned with noise spectrum estimation, we are typically more interested in specifying parameters of the resulting PSD rather than parameters of the underlying time series data. The `python_spectrometer` package approaches data acquisition from the inverse direction: rather than inferring the spectrum properties from the time series data, users specify the properties they would like the resulting spectrum to have and the package chooses the correct parameters for data acquisition accordingly.

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.[3] 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.

3.1.1 Data acquisition

?? shows the directory structure of the source code. The `daq` subpackage contains on the one hand the declaration of the `DAQ` abstract base class (`base.py`) and its child class implementations (`qcodes.py`, *etc.*), and on the other the `settings.py` module, which defines the `DAQSettings` class.

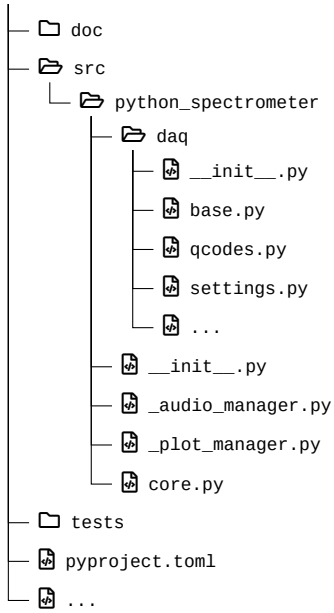

Figure 3.1: Source tree structure of the `python_spectrometer` package. Driver wrappers are placed in the `daq` subpackage. `core.py` exports the `Spectrometer` class.

3: Actually *drivers* to be mo

This class is used in the background to validate data acquisition settings both for consistency (see **??**) and hardware constraints.

Table 3.1: Variable names used in **??** and their corresponding parameter names as used in `python_spectrometer` and `scipy.signal.welch()` [17].

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

To better understand the necessity of this functionality, consider the typical scenario of a physicist[4] 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,[5] she does not want to do the mental gymnastics to convert f_{min} to the parameter that her DAQ device understands, L (see **??**), 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 **??**[6] and attempts to resolve the parameter interdependencies lined out in **??** upon calling `DAQSettings.to_consistent_dict()`.[7] 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.

For instance, Alice might want to measure the noise spectrum in the frequency band $[1.5\,\text{Hz}, 72\,\text{kHz}]$. Although she would not have to do this explicitly,[8] she could inspect the parameters after resolution using the code shown in **??**.

4: Let's call her Alice.

5: Physicists generally are.

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

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

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

```
>>> 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}
```

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.

```
{'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 **??** 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,[9] the same requested settings would have resolved to those shown in **??**.[10] This is because the Scope module constrains $L \in [2^{12}, 2^{14}]$ and $f_s \in 60\,\text{MHz} \times 2^{[-16,0]} \approx \{915.5\,\text{Hz}, \ldots, 30\,\text{MHz}, 60\,\text{MHz}\}$.

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

9: https://docs.zhinst.com/labone_programming_manual/python_user_manual/modules/scope/index.html

10: `DAQ` issued a warning to inform the user their requested settings could not be matched

functionality is best illustrated by the internal workflow as representatively shown in **??**.

```
AQ(driver_handle)

ttings = daq.setup(**user_settings)
on_generator = daq.acquire(**parsed_settings)

buffer in acquisition_generator:
ate_psd(data_buffer)
```

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).

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,[11] 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.[12] An exemplary implementation of a `DAQ` subclass for a fictitious instrument is shown in **??**. 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[13] can be specified.

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

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

13: See for example the implementation of the AlazarTech ATS9440 digitizer card.

3.1.2 Data processing

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 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,[14] 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.

14: 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. **??** shows a simple function that accounts for the gain of an amplifier. 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()` [17]. Here, the keyword `fourier_procfn` should be a callable with signature `(xf, f, **kwargs) -> (xfp, fp)`.[15] 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.

[17]: (n.d.), *`welch` — SciPy v1.15.2 Manual*

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

```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.

```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. Note the token `**_` variable keyword argument that ensures no errors arise from other parameters being passed to the function. 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 are required in case the function modifies the frequencies.[16] A simple example for a processing function in Fourier space is shown in **??**, which computes the (anti-)derivative of the data using the fact that

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

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

16: One example is the `octave_band_rms()` function from the `qutil.signal_processing` subpackage [18] that decimates frequencies to fractions of octave bands. See **??** for a use-case.

```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.

3.2 Feature overview

Now that we have a basic understanding of the design choices underlying `python_spectrometer`, let us discuss the typical workflow of using the package. Two modes of operation are to be distinguished: first, "serial" mode, in which users record new spectra manually, and second, "live" mode, in which new data is continuously being acquired. The former is well suited to a structured approach to noise spectroscopy, where data is retained persistently and discrete changes are made to the system between data acquisitions. The latter is aimed at a more fluent workflow in which data is not retained and data acquisition runs in the background.

blabla

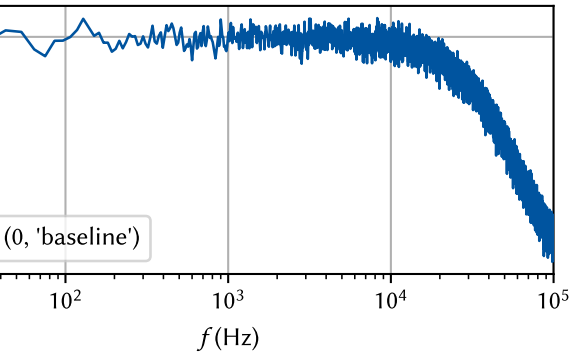

Figure 3.2: The `python_spectrometer` plot after acquiring the (here: synthetic) baseline spectrum. By default, the amplitude spectral density (ASD) = √PSD is displayed in the main plot. Each spectrum is assigned a unique identifier key consisting of an incrementing integer and the user comment, and can be refered to by either (or both) when interacting with the object.

3.2.1 Serial spectrum acquisition

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 be assigned a comment that allows to easily identify it 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,[17] 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.[18] Choosing to work with the demodulated data to benefit from the corresponding DAQ modules larger flexibility, he recognizes that the resulting PSD will be the two-sided version because the data returned by the LIA is complex. Since he is interested in the physical frequencies[19] he sets the lock-in's modulation frequency to 0 Hz and disables plotting the negative part of the frequency spectrum as it contains only redundant data in this case. Selecting the frequency bounds, say $f_{\text{min}} = 10\,\text{Hz}$ and $f_{\text{max}} = 100\,\text{kHz}$, 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 **??**, which produces the plot shown in **??**.

17: Let's call him Bob.

18: For the MFLI, `DAQ` subclasses for both the Scope and the DAQ module are implemented. The former gives access to the signal before and the latter to the signal after demodulation.

19: A discussion of lock-in amplification is beyond the scope of this chapter. Here I will simply note that, for finite modulation frequencies f_m, LIAs will measure the PSD in the up-converted frequency band $[-f_{\text{max}} + f_m, f_m + f_{\text{max}}]$ rather than the baseband.

```
on_spectrometer import Spectrometer, daq
l.functools import scaled

= daq.ZurichInstrumentsMFLIDAQ(session, device)
pectrometer(mfli_daq, procfn=scaled(1e6),
            plot_negative_frequencies=False,
            processed_unit='µV')

= {'f_min': 1e1, 'f_max': 1e5', 'freq': 0}
e('baseline', daq='grounded', **settings)
```

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.

The noise spectrum he obtains is white up until approximately 20 kHz where it starts falling off $\propto f^{-n}$. This is because the LIA low-pass filters the signal to suppress aliasing[20] 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

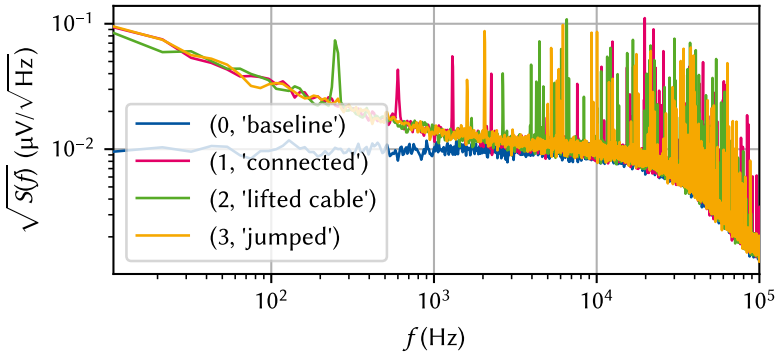

Figure 3.3: The `python_spectrometer` plot after acquiring additional (synthetic) spectra. Each spectrum is uniquely identified by a two-tuple of `(index, comment)`.

comment for identification. The code shown in **??** would then leave him

```
settings['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.

with the spectrometer plot as shown in **??**. 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`, which adds an oscilloscope subplot to spectrometer figure as shown in **??**. In case the DAQ returns complex data, the absolute value $R = X + iY$ is plotted.

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 **??**. In the case that `spect.plot_amplitude == True`, this new subplot shows the RMS in the band $[f_{\min}, f]$,

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

and the band power (**??**) 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[21] by applying the following settings, which produces the plot shown in **??**:

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

The attribute `plot_density` controls whether the *power spectral density* or the *power spectrum*[22] is plotted. 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.

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

21: Recall that the decibel is defined by the ratio L_p of two powers P_1, P_2 as [19]
$$L_p = 10 \log_{10}\left(\frac{P_1}{P_2}\right) \text{dB}.$$

22: At this point, we *do* need to distinguish between the PSD and the power spectrum counter to **??** in **??**. The PSD and power spectrum are related by the equivalent noise bandwidth (ENBW),
$$\text{Spectral density} = \frac{1}{\text{ENBW}} \times \text{Spectrum},$$
which is itself a function of the sampling rate and the properties of the spectral window [14],
$$\text{ENBW} = f_s \cdot \frac{\sum_n \hat{w}_n^2}{\left[\sum_n \hat{w}_n\right]^2}. \tag{3.3}$$

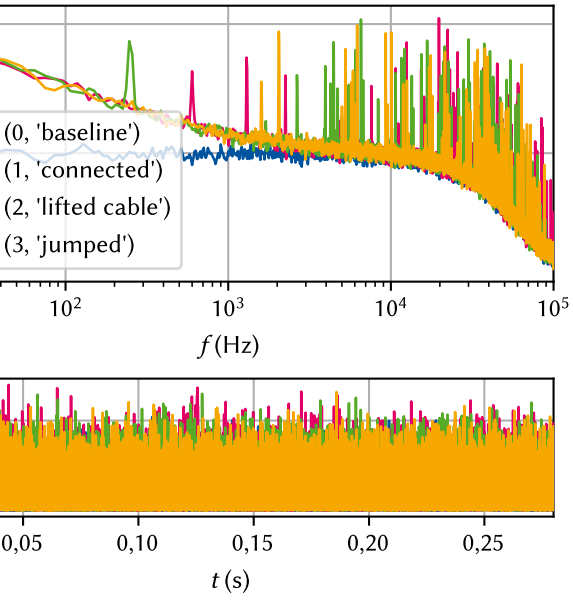

Figure 3.4: The `python_spectrometer` plot shown in **??** when setting `spect.plot_timetrace = True`. This adds a subplot that shows the time series data from which the PSD was computed akin to what an oscilloscope would show. For complex time series, the absolute value $R = X + iY$ is plotted. Note 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.

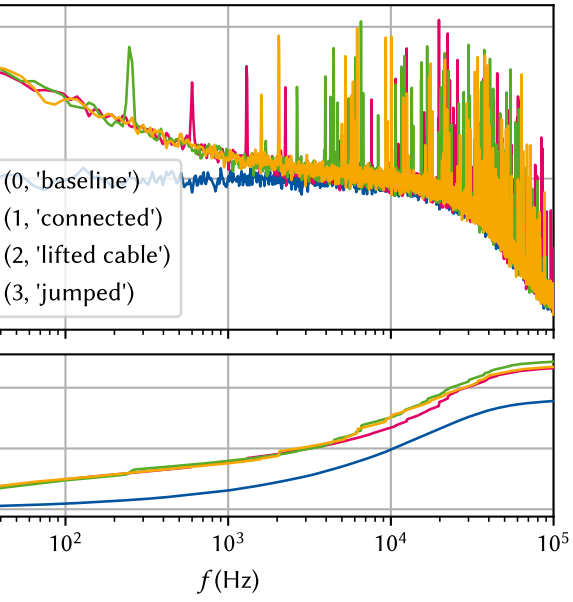

Figure 3.5: The `python_spectrometer` plot shown in **??** when setting `spect.plot_cumulative = True`. This adds a subplot that shows the RMS (see **??**) which can be helpful in evaluating the contribution of individual peaks in the spectrum to the total noise power. Both the oscilloscope subplot (**??**) and the RMS subplot can also be shown at the same time.

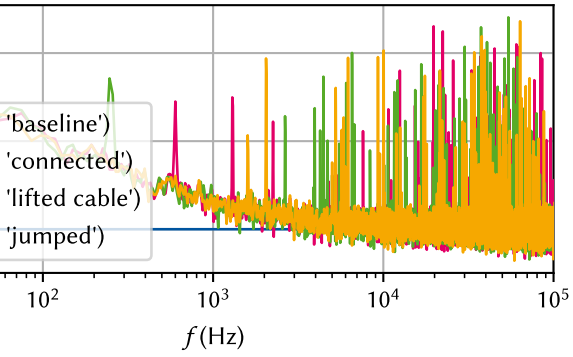

Figure 3.6: The `python_spectrometer` plot in relative mode. Starting from the state in **??**, we set `spect.plot_dB_scale = True` as well as `spect.plot_amplitude = False` and `spect.plot_density = False` to compare the relative noise powers with respect to the baseline.

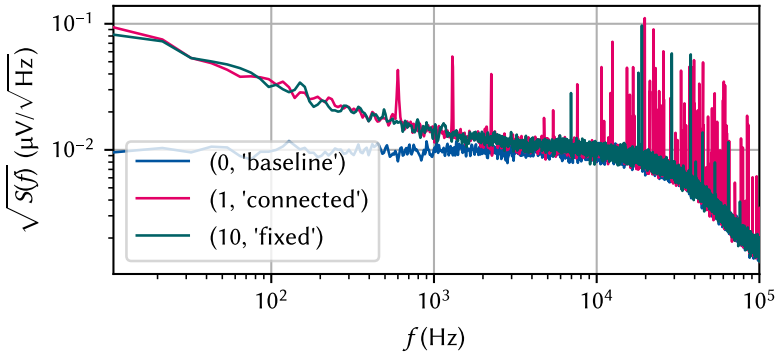

Figure 3.7: The `python_spectrometer` plot after multiple additional spectra were acquired and hidden. Hiding spectra that one is not interested in anymore is achieved through `spect.hide(*range(2, 10))`. This is reversed by `spect.show(*range(2, 10))`. Data can also be dropped (`spect.drop(key)`) or deleted (`spect.delete(key)`) from the internal cache and disk, respectively.

the baseline, the original, noisy state, and the final, clean spectrum. He simply calls

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

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: