The kaobook class

**Use this document as a template**

# Millikelvin Confocal Microscopy of Semiconductor Membranes and Filter Functions for Unital Quantum Operations

**Customise this page according to your needs**

Tobias Hangleiter*

July 14, 2025

* A LaTeX lover/hater

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

# List of Figures

# Publications


[1] Yaiza Aragonés-Soria, René Otten, Tobias Hangleiter, Pascal Cerfontaine, and David Gross. "Minimising Statistical Errors in Calibration of Quantum-Gate Sets." June 7, 2022. DOI: 10.48550/arXiv.2206.03417. (Visited on 06/08/2022). Pre-published.

[2] Pascal Cerfontaine, Tobias Hangleiter, and Hendrik Bluhm. "Filter Functions for Quantum Processes under Correlated Noise." In: *Physical Review Letters* 127.17 (Oct. 18, 2021), p. 170403. DOI: 10.1103/PhysRevLett.127.170403.

[3] Thomas Descamps, Feng Liu, Sebastian Kindel, René Otten, Tobias Hangleiter, Chao Zhao, Mihail Ion Lepsa, Julian Ritzmann, Arne Ludwig, Andreas D. Wieck, Beata E. Kardynał, and Hendrik Bluhm. "Semiconductor Membranes for Electrostatic Exciton Trapping in Optically Addressable Quantum Transport Devices." In: *Physical Review Applied* 19.4 (Apr. 28, 2023), p. 044095. DOI: 10.1103/PhysRevApplied.19.044095. (Visited on 04/28/2023).

[4] Thomas Descamps, Feng Liu, Tobias Hangleiter, Sebastian Kindel, Beata E. Kardynał, and Hendrik Bluhm. "Millikelvin Confocal Microscope with Free-Space Access and High-Frequency Electrical Control." In: *Rev. Sci. Instrum.* 95.8 (Aug. 9, 2024), p. 083706. DOI: 10.1063/5.0200889. (Visited on 08/12/2024).

[5] Denny Dütz, Sebastian Kock, Tobias Hangleiter, and Hendrik Bluhm. "Distributed Bragg Reflectors for Thermal Isolation of Semiconductor Spin Qubits." In preparation.

[6] Sarah Fleitmann, Fabian Hader, Jan Vogelbruch, Simon Humpohl, Tobias Hangleiter, Stefanie Meyer, and Stefan van Waasen. "Noise Reduction Methods for Charge Stability Diagrams of Double Quantum Dots." In: *IEEE Transactions on Quantum Engineering* 3 (2022), pp. 1–19. DOI: 10.1109/TQE.2022.3165968.

[7] Fabian Hader, Jan Vogelbruch, Simon Humpohl, Tobias Hangleiter, Chimezie Eguzo, Stefan Heinen, Stefanie Meyer, and Stefan van Waasen. "On Noise-Sensitive Automatic Tuning of Gate-Defined Sensor Dots." In: *IEEE Transactions on Quantum Engineering* 4 (2023), pp. 1–18. DOI: 10.1109/TQE.2023.3255743.

[8] Tobias Hangleiter, Pascal Cerfontaine, and Hendrik Bluhm. "Filter-Function Formalism and Software Package to Compute Quantum Processes of Gate Sequences for Classical Non-Markovian Noise." In: *Physical Review Research* 3.4 (Oct. 18, 2021), p. 043047. DOI: 10.1103/PhysRevResearch.3.043047. (Visited on 01/19/2022).

[9] Tobias Hangleiter, Pascal Cerfontaine, and Hendrik Bluhm. "Erratum: Filter-function Formalism and Software Package to Compute Quantum Processes of Gate Sequences for Classical Non-Markovian Noise [Phys. Rev. Research 3, 043047 (2021)]." In: *Physical Review Research* 6.4 (Oct. 16, 2024), p. 049001. DOI: 10.1103/PhysRevResearch.6.049001. (Visited on 10/16/2024).

[10] Isabel Nha Minh Le, Julian D. Teske, Tobias Hangleiter, Pascal Cerfontaine, and Hendrik Bluhm. "Analytic Filter-Function Derivatives for Quantum Optimal Control." In: *Physical Review Applied* 17.2 (Feb. 2, 2022), p. 024006. DOI: 10.1103/PhysRevApplied.17.024006. (Visited on 02/03/2022).

[11] Paul Surrey, Julian D. Teske, Tobias Hangleiter, Pascal Cerfontaine, and Hendrik Bluhm. "Data-Driven Qubit Characterization and Optimal Control Using Deep Learning." In preparation.

[12] Kui Wu, Sebastian Kindel, Thomas Descamps, Tobias Hangleiter, Jan Christoph Müller, Rebecca Rodrigo, Florian Merget, Hendrik Bluhm, and Jeremy Witzens. "An Efficient Singlet-Triplet Spin Qubit to Fiber Interface Assisted by a Photonic Crystal Cavity." In: *The 25th European Conference on Integrated Optics*. Ed. by Jeremy Witzens, Joyce Poon, Lars Zimmermann, and Wolfgang Freude. Cham: Springer Nature Switzerland, 2024, pp. 365–372. DOI: 10.1007/978-3-031-63378-2_60.

[13] Kui Wu, Sebastian Kindel, Thomas Descamps, Tobias Hangleiter, Jan Christoph Müller, Rebecca Rodrigo, Florian Merget, Beata E. Kardynal, Hendrik Bluhm, and Jeremy Witzens. "Modeling an Efficient Singlet-Triplet-Spin-Qubit-to-Photon Interface Assisted by a Photonic Crystal Cavity." In: *Physical Review Applied* 21.5 (May 24, 2024), p. 054052. DOI: 10.1103/PhysRevApplied.21.054052. (Visited on 08/21/2024).

# Software

The following open-source software packages were developed (at least partially) during the work on this thesis.


[1] Tobias Hangleiter, Isabel Nha Minh Le, and Julian D. Teske, *Filter_functions* version v1.1.3, May 14, 2024. Zenodo. DOI: 10.5281/ZENODO.4575000.

[2] Tobias Hangleiter, *Lindblad_mc_tools*.

[3] Tobias Hangleiter, *Mjolnir*.

[4] Tobias Hangleiter, Simon Humpohl, Max Beer, and René Otten, *Python-Spectrometer* version 2024.11.1, Nov. 21, 2024. Zenodo. DOI: 10.5281/ZENODO.13789861.

[5] Tobias Hangleiter, Simon Humpohl, Paul Surrey, and Han Na We, *Qutil* version 2024.11.1, Nov. 21, 2024. Zenodo. DOI: 10.5281/ZENODO.14200303.

# Part I

# A Flexible Python Tool For Fourier-Transform Noise Spectroscopy

# The python_spectrometer software package 1

In this chapter, I introduce the `python_spectrometer` Python package[1] [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` [2], obtaining the data array is not standardized. Different data acquisition (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`.

[2]: (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 power spectral density (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.

## 1.1 Package design and implementation

The `python_spectrometer` package provides a central class, `Spectrometer`, that users interact with to perform data acquisition, spectrum estimation, and plotting. It is instantiated with an instance of a child class of the `DAQ` base class that implements an interface to various 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: Actually *drivers* to be more precise.

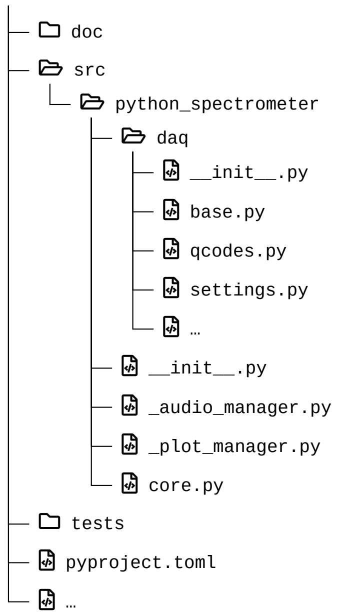


**Figure 1.1:** Source tree structure of the `python_spectrometer` package. Driver wrappers are placed in the `daq` subpackage. `core.py` exports the `Spectrometer` class.

### 1.1.1 Data acquisition

Figure 1.1 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. This class is used in the background to validate data acquisition settings both for consistency (see ??) and hardware constraints.

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 $[f_{\mathrm{min}}, f_{\mathrm{max}}]$ she is most interested in. But because she is lazy,[5] she does not want to do the mental gymnastics to convert $f_{\mathrm{min}}$ to the parameter that her DAQ device understands, $L$ (see Table 1.1), especially considering that $L$ depends on the number of Welch averages and the overlap. Furthermore, while she could just about do the conversion from $f_{\mathrm{max}}$ to the other relevant DAQ parameter, $f_{\mathrm{s}}$, in her head, her device imposes hardware constraints on the allowed sample rates she can select! The `DAQSettings` class addresses these issues. It is instantiated with any subset of the parameters listed in Table 1.1[6] and attempts to resolve the parameter interdependencies laid out in ?? and depicted 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.

**Table 1.1:** Variable names used in ?? and their corresponding parameter names as used in `python_spectrometer` and `scipy.signal.welch()` [2].

| Variable | \protect\textsc{Parameter} |
|---|---|
| $L$ | n_pts |
| $N$ | nperseg |
| $K$ | noverlap |
| $M$ | n_seg |
| $O$ | n_avg |
| $f_s$ | fs |
| $f_{\mathrm{max}}$ | f_max |
| $f_{\mathrm{min}}$ | f_min |

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 Table 1.1. 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.

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,[8] she could inspect the parameters after resolution using the code shown in Listing 1.1.

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

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 Listing 1.2.[10] This is because the Scope module constrains $L \in [2^{12}, 2^{14}]$ and $f_{\mathrm{s}} \in 60\,\mathrm{MHz} \times 2^{[-16,0]} \approx \{915.5\,\mathrm{Hz}, \ldots, 30\,\mathrm{MHz}, 60\,\mathrm{MHz}\}$.

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

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

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

10: 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 1.3.

```
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)
```

**Listing 1.3:** 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 Listing 1.4. 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[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.

### 1.1.2 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. Listing 1.5 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()` [2]. Here, the keyword `fourier_procfn` should be a callable with signature `(xf, f, **kwargs) -> (xfp, fp)`.[15] That is, it should take the frequency-

[2]: (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 1.4:** 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.

space data, the corresponding frequencies, and arbitrary keyword arguments and return a tuple of the processed data and the corresponding frequencies.

A simple example for a processing function in Fourier space is shown in Listing 1.6, which computes the (anti-)derivative of the data using the fact that

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

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.

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

**Listing 1.5:** 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.

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

**Listing 1.6:** A simple `fourier_procfn`, which calculates the (anti-)derivative.

## 1.2 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 in between subsequent data acquisitions. The latter is aimed at a more fluent workflow in which data is not retained and data acquisition runs in the background.

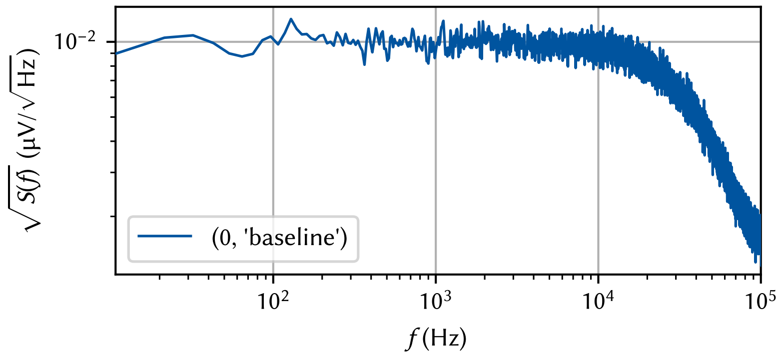


**Figure 1.2:** The `python_spectrometer` plot after acquiring the (here: synthetic) baseline spectrum. By default, the amplitude spectral density (ASD) = $\sqrt{\text{PSD}}$ is displayed in the main plot. Each spectrum is assigned a unique identifier key consisting of an incrementing integer and the user comment, and can be refered to by either (or both) when interacting with the object.

### 1.2.1 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,[16] 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.[17] Choosing to work with the demodulated data to benefit from the corresponding DAQ module's larger flexibility, he recognizes that the resulting PSD will be the two-sided version because the data returned by the lock-in amplifier (LIA) is complex. Since he is interested in the physical frequencies[18] 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 $f_{\text{min}} = 10$ Hz and $f_{\text{max}} = 100$ 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 1.7, which produces the plot shown in Figure 1.2.

16: Let's call him Bob.

17: 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.

18: 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.

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

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

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

**Listing 1.7:** 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.[19] 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

19: Aliasing effects arise from finite sampling according to the Nyquist-Shannon sampling theorem [3–5]. It states that for a given physical bandwidth, the sampling rate $f_s$ must be at least twice as large to faithfully reconstruct the signal in order to avoid aliasing, *i. e.*, the reverse of the argument we have made for the largest resolvable frequency $f_{\text{max}}$. Some DAQ devices perform internal aliasing rejection while others do not.

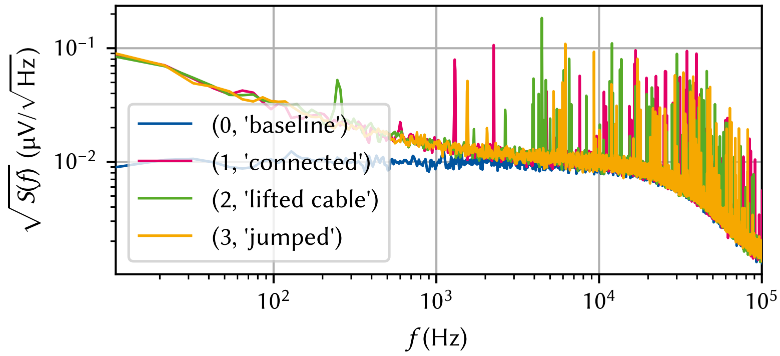


**Figure 1.3:** The `python_spectrometer` plot after acquiring additional (synthetic) spectra. Each spectrum is uniquely identified by a two-tuple of `(index, comment)`.

meaningful comment for identification. The code shown in Listing 1.8 would then leave him with the spectrometer plot as shown in Figure 1.3. 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 the spectrometer figure as shown in Figure 1.4. Since his DAQ returns complex data, the absolute value $R = X + iY$ is plotted.

```
settings['daq'] = 'connected'
spect.take('connected', **settings)
spect.take('lifted cable', cable='lifted', **settings)
spect.take('jumped', **settings)
```

**Listing 1.8:** 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.

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 1.5. In the case that `spect.plot_amplitude == True`, this new subplot shows the root mean square (RMS) in the band $[f_{\mathrm{min}}, f]$,

$$\mathrm{RMS}_S(f) \equiv \mathrm{RMS}_S(f_{\mathrm{min}}, f), \tag{1.2}$$

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[20] by applying the following settings, which changes the plot to the layout shown in Figure 1.6:

```
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* is plotted.[21] 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

20: Recall that the decibel is defined by the ratio $L_P$ of two powers $P_1, P_2$ as [6]

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

21: 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),

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

which is itself a function of the sampling rate and the properties of the spectral window [7],

$$\mathrm{ENBW} = f_s \frac{\sum_n \hat{w}_n^2}{\left[\sum_n \hat{w}_n\right]^2}. \tag{1.3}$$

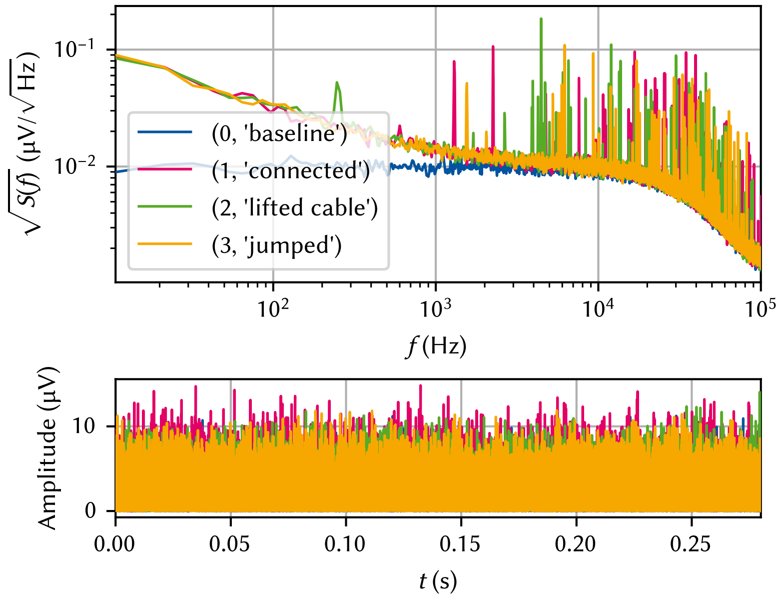


**Figure 1.4:** The `python_spectrometer` plot shown in Figure 1.3 when setting `spect.plot_timetrace = True`. This adds a subplot that shows the time series data from which the PSD was computed akin to what an oscilloscope would show. For complex time series, the absolute value $R = X + iY$ is plotted. 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.

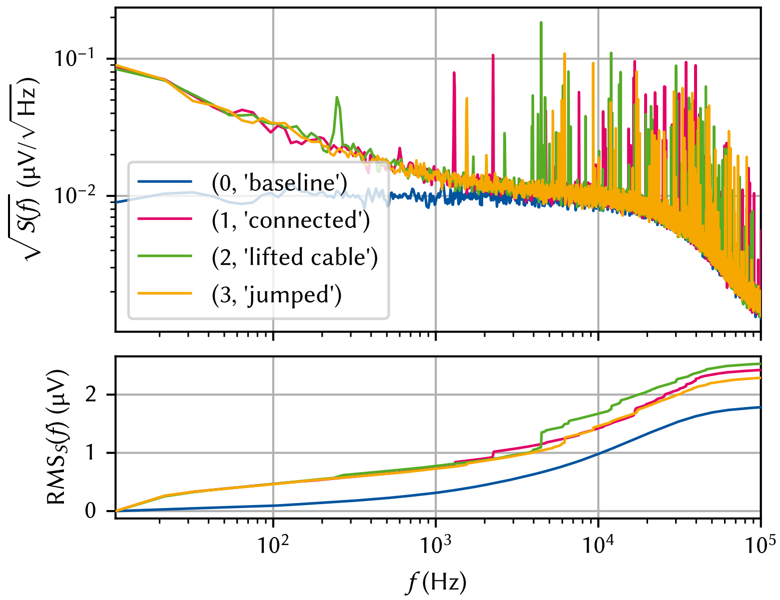


**Figure 1.5:** The `python_spectrometer` plot shown in Figure 1.3 when setting `spect.plot_cumulative = True`. This adds a subplot that shows the RMS (see **??**) which can be helpful in evaluating the contribution of individual peaks in the spectrum to the total noise power. Both the oscilloscope subplot (Figure 1.4) and the RMS subplot can also be shown at the same time.

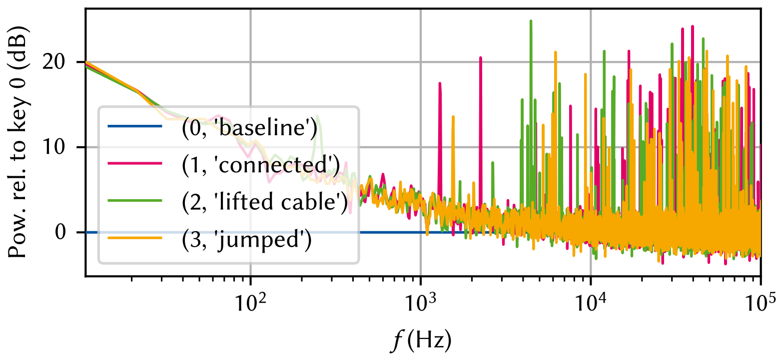


**Figure 1.6:** The `python_spectrometer` plot in relative mode. Starting from the state in Figure 1.3, we set `spect.plot_dB_scale = True` as well as `spect.plot_amplitude = False` and `spect.plot_density = False` to compare the relative noise powers with respect to the baseline.

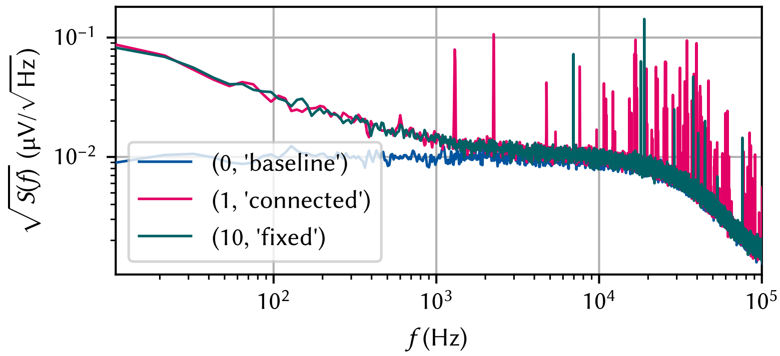


**Figure 1.7:** The `python_spectrometer` plot after multiple additional spectra were acquired and hidden. Hiding spectra that one is not interested in anymore is achieved through `spect.hide(*range(2, 10))`. This is reversed by `spect.show(*range(2, 10))`. Data can also be dropped (`spect.drop(key)`) or deleted (`spect.delete(key)`) from the internal cache and disk, respectively.

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 1.7.

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 1.7 again. While they cannot acquire new spectra in this state,[22] they can still use all the plotting features like showing or hiding spectra, or changing plot types as discussed above.

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

### 1.2.2 Live spectrum acquisition

Manually recording spectra in the workflow outlined in 1.2.1 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 storage space. For these 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.[23] In the former case, the window hosting the figure runs in the

23: Note that technically, data is also recorded in a background thread in serial mode by default.

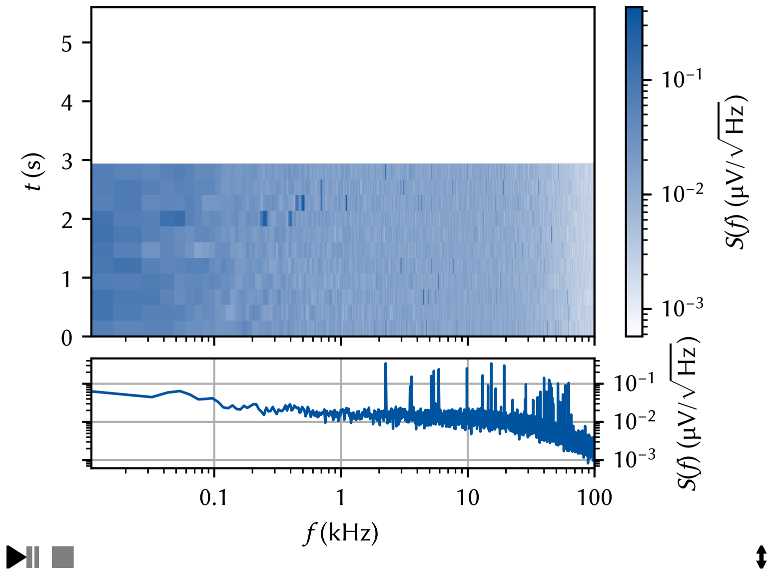


**Figure 1.8:** Spectrometer live view. The bottom plot shows the most recently acquired spectrum, while the top plot shows a waterfall plot of the most recent ones. Data acquisition runs in a background thread, keeping the interpreter responsive and available to interact with instruments, for example. The icons in the bottom left and right corners allow interacting with the live view.

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[24] 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.[25] A `live_view.IncrementalLiveView2D` object then retrieves the data from the queue and handles plotting in the graphical user interface (GUI) event loop. To start a spectroscopy session with the same parameters as in 1.2.1 with a given `Spectrometer` object (see Listing 1.7), we would call

```
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 1.8. Similar to `take()`, the data acquisition parameters are passed to `live_view()` as keyword arguments.[26] 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.

24: Technically, a very large `n_avg` is used.

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

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

# Part II

# Characterization and Improvements of a Millikelvin Confocal Microscope

# Part III

# Optical Measurements of Electrostatic Exciton Traps in Semiconductor Membranes

# Introduction 2

CATS

# 3 The `mjolnir` measurement framework

MJÖLNIR

```
├ doc
├ src
│ └ mjolnir
│   ├ config
│   │ ├ physical_[…].yaml
│   │ ├ optical_path.yaml
│   │ ├ fig_F10.yaml
│   │ └ …
│   ├ instruments
│   │ ├ __init__.py
│   │ ├ logical_[…].py
│   │ └ physical_[…].py
│   ├ measurements
│   │ ├ __init__.py
│   │ ├ handler.py
│   │ ├ measures.py
│   │ └ sweeps.py
│   ├ parameters
│   ├ plotting
│   │ ├ __init__.py
│   │ ├ live_view.py
│   │ └ plot_nd.py
│   ├ __init__.py
│   ├ calibration.py
│   └ …
├ main.py
├ pyproject.toml
└ …
```

**Figure 3.1:** Source tree structure of the `mjolnir` package. Logical `QCoDeS` instruments and parameters are defined in the `instruments` and `parameters` modules, respectively. Instruments are configured using `yaml` files located in the `config` directory. The `measurements` module provides classes for the abstraction of measurements using `QCoDeS` underneath. Live plots of instrument data as well as a plot function for multidimensional measurement data are defined in the `plotting` module. `calibration.py` contains routines for power, CCD, and excitation rejection calibrations. The `main.py` file is a code cell-based script that serves as the entrypoint for measurements.

# Observations 4

PIDGEONS

## 4.1 Transfer-matrix method simulations of the membrane structure

The transfer-matrix method (TMM) is a computationally efficient method of obtaining the electric field in layered structures. In this section, I perform simulations of the heterostructure membranes investigated in this part of the present thesis using the `PyMoosh` package [8] to elucidate the observed quenching of photoluminescence (PL) when illuminating gate electrodes as well as the overall optical efficiency.[1] I will first briefly recap the simulation method following Reference 8. For more details, refer to *ibid.* and references therein.

1: Strictly speaking, the term TMM only refers to one of the several formalisms implemented in the `PyMoosh` package. While fast, it not the most numerically stable, and other methods may be preferred if wall time is not a limiting issue.

Consider a layered structure along $z$ with interfaces at $z_i, i \in \{0, 1, \dots, N+1\}$ that is translationally invariant along $x$ and $y$. Each layer $i$ may consist of a different dielectric material characterized by a (complex) relative permittivity $\epsilon_{r,i}$.[2] The electric field component along $y$ of an electromagnetic wave transverse electric (TE) mode originating in some far away point satisfies the Helmholtz equation

2: We disregard magnetic materials with relative permeability $\mu_r \neq 1$ for simplicity.

$$\frac{\partial^2 E_y}{\partial z^2} + \gamma_i^2 E_y = 0, \tag{4.1}$$

where $\gamma_i = \sqrt{\epsilon_{r,i} k_0^2 - k_x^2}$ with $k_0 = \omega/c$ the wave vector in vacuum and $k_x$ the component along $x$. In layer $i$ of the structure, the solution to Equation 4.1 may be written as a superposition of plane waves incident and reflected on the lower and upper interfaces [8],

$$\begin{cases} E_{y,i}(z) = A_i^+ \exp\{i\gamma_i[z - z_i]\} + B_i^+ \exp\{-i\gamma_i[z - z_i]\}, \\ E_{y,i}(z) = A_i^- \exp\{i\gamma_i[z - z_{i+1}]\} + B_i^- \exp\{-i\gamma_i[z - z_{i+1}]\}, \end{cases} \tag{4.2}$$

where the coefficients with superscript + (−) are referenced to the phase at the upper (lower) interface, respectively. Matching these solutions at $z = z_i$ for all $i$ to satisfy the interface conditions imposed by Maxwell's equations gives rise to a linear system of equations, the solution to which can be obtained through several different methods.

A particularly simple method is the transfer-matrix method ($T$-matrix formalism), which corresponds to writing the interface conditions at $z = z_i$ as the matrix equation

$$\begin{pmatrix} A_{i+1}^+ \\ B_{i+1}^+ \end{pmatrix} = T_{i,i+1} \begin{pmatrix} A_i^- \\ B_i^- \end{pmatrix} \tag{4.3}$$

with

$$T_{i,i+1} = \frac{1}{2\gamma_{i+1}} \begin{pmatrix} \gamma_i + \gamma_{i+1} & \gamma_i - \gamma_{i+1} \\ \gamma_i - \gamma_{i+1} & \gamma_i + \gamma_{i+1} \end{pmatrix} \tag{4.4}$$

the transfer matrix for interface $i$. Connecting the coefficients for adjacent interfaces within a layer of height $h_i = z_{i+1} - z_i$ requires propagating the phase,

$$\begin{pmatrix} A_i^- \\ B_i^- \end{pmatrix} = C_i \begin{pmatrix} A_i^+ \\ B_i^+ \end{pmatrix}, \tag{4.5}$$

with

$$C_i = \exp\left\{\mathrm{diag}(-i\gamma_i h_i, i\gamma_i h_i)\right\}. \tag{4.6}$$

Iterating Equations 4.4 and 4.6, the total transfer matrix $T = T_{0,N+1}$ then reduces to the matrix product

$$T = T_{N,N+1} \prod_{i=0}^{N-1} T_{i,i+1} C_i. \tag{4.7}$$

From $T$, the reflection and transmission coefficients can be obtained as $r = A_0^- = -T_{01}/T_{00}$ and $t = B_{N+1}^+ = rT_{10} + T_{11}$. Taking the absolute value square of reflection and transmission coefficients then yields the reflectance $\mathscr{R}$ and the transmittance $\mathscr{T}$, which correspond to the fraction of total incident power being reflected and transmitted, respectively. To obtain the absorptance $\mathscr{A}$, the fraction of power being absorbed, in layer $i$, one can compute the difference of the $z$-components of the Poynting vectors (*cf.* **??**) at the top of layers $i$ and $i+1$. In the TE case considered here, **??** reduces to [8]

$$\mathrm{S}_i = \mathrm{Re}\left[\frac{\gamma_i^*}{\gamma_0}\left(A_i^+ - B_i^+\right)^*\left(A_i^+ + B_i^+\right)\right] \tag{4.8}$$

and is hence straightforward to extract from the calculation of either the $S$ or $T$ matrices.

Equation 4.7 is simple to evaluate on a computer, making this method attractive for numerical applications. However, the opposite signs in the argument of the exponentials in Equation 4.6 can lead to instabilities for evanescent waves ($\gamma_i \in \mathbb{C}$) due to finite-precision floating point arithmetic [9]. Rewriting Equation 4.4 to have incoming and outgoing fields on opposite sides of the equality alleviates this issue while sacrificing the simple matrix-multiplication composition rule in what is known as the scattering matrix ($S$-matrix) formalism.

Beyond the calculation of the aforementioned coefficients, the TMM formalism also allows to compute the full spatial dependence of the fields. Two cases are implemented in `PyMoosh`: irradiation of the layered structured with a Gaussian beam rather than plane waves of infinite extent, and a current line source inside the structure. In the first case, the previously assumed translational invariance along $x$ leading to a plane-wave spatial dependence is replaced by a superposition of plane waves weighted with a normally distributed amplitude,[3]

3: *I. e.*, the inverse Fourier transform of $\mathcal{E}_0(k_x)E_{y,i}(k_x, z)$.

$$E_{y,i}(x,z) = \exp(ik_x x) \to \int \frac{\mathrm{d}k_x}{2\pi}\mathcal{E}_0(k_x)E_{y,i}(k_x,z)\exp(ik_x x), \tag{4.9}$$

with (*cf.* **??**)

$$\mathcal{E}_0(k_x) = \frac{w_0}{2\sqrt{\pi}}\exp\left\{-ik_x x_0 - \left[\frac{w_0 k_x}{2}\right]^2\right\} \tag{4.10}$$

and

$$E_{y,i}(k_x,z) = A_i^- \exp\{i\gamma_i(k_x)[z - z_{i+1}]\} + B_i^+ \exp\{-i\gamma_i(k_x)[z - z_i]\}, \tag{4.11}$$

and where we considered only normal incidence for simplicity.

In the second case, Langevin et al. [8] consider an AC current $I$ flow-

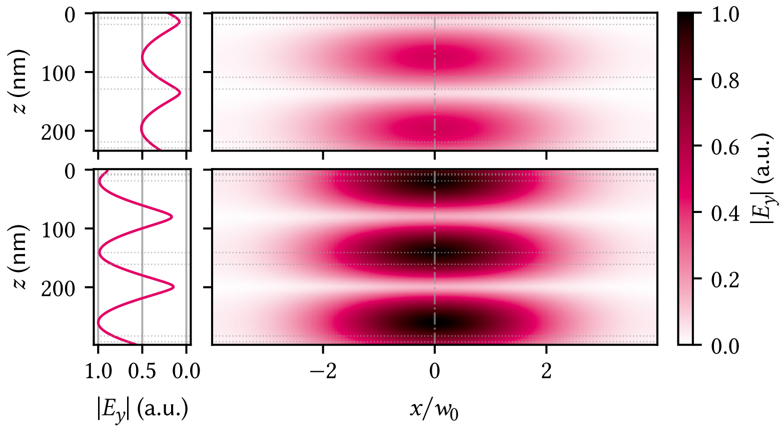


**Figure 4.1:** Absolute value of the electric field inside the double-gated heterostructure under illumination with a Gaussian beam at $\lambda = 825\,\mathrm{nm}$ from the top. Top (bottom) panels show the structure with the default (optimized) barrier thickness of 90 nm (122 nm), respectively. Dotted horizontal lines indicate interfaces between different materials while the vertical dash-dotted line indicates the position of the line cuts shown in the left column. Increasing the thickness of the barrier has two beneficial effects; first, the overall field intensity inside the structure is higher by a factor of two, and second, there is a peak rather than a knot in the quantum well (QW) at a depth of ~ 120 nm (~ 150 nm), leading to enhanced absorption.

ing through a translationally invariant, one-dimensional wire along $y$ at $x = x_{\mathrm{s}}$. This introduces a source term into the Helmholtz equation Equation 4.1 which, upon Fourier transforming in $x$ direction, leads to

$$\frac{\partial^2 \hat{E}_y}{\partial z^2} + \gamma_i^2 \hat{E}_y = -i\omega\mu_0 I\delta(z)\exp(ik_x x_{\mathrm{s}}). \tag{4.12}$$

The electric field $\hat{E}_{y,i}(k_x, z)$ is thus proportional to the Green's function of Equation 4.12 and can be obtained using a similar procedure as in the case of a distant source incident on the structure by matching the interface conditions. Performing the inverse Fourier transform by means of Equation 4.9 with constant weights, $\mathcal{E}_0(k_x) \equiv 1$, then yields the two-dimensional spatial distribution of the electric field, $E_{y,i}(x, z)$.

**Table 4.1**

| | $\mathcal{A}$ (%) | $\mathcal{R}$ (%) |
|---|---|---|
| Bare | 2.93 | 22.43 |
| TG | 1.79 | 41.98 |
| BG | 0.50 | 82.72 |
| TG+BG | 0.41 | 84.78 |

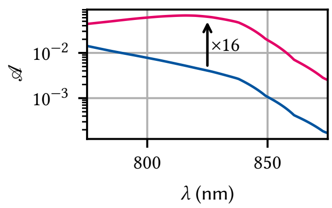


**Figure 4.2:** QW absorptance $\mathscr{A}$ in a heterostructure with default (blue) and optimized (magenta) barrier thickness and top and bottom gates as function of wavelength. Optimization was performed at 825 nm using the differential evolution algorithm implemented in `PyMoosh`, resulting in a barrier thickness of 122 nm and an absorptance better by a factor of 16 at 6.3 %.

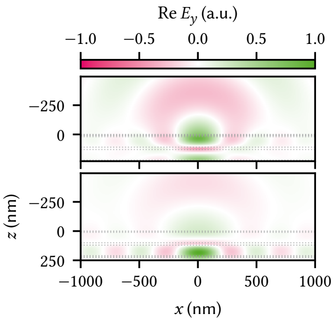


**Figure 4.3:** Real part of the electric field emitted by a current line located in the QW (black point) for different cases of the unoptimized structure. From top to bottom: bare heterostructure, top gate, bottom gate, top and bottom gate. The half space $z < 0$ is the air above the membrane in the direction of the objective lens and the dotted lines indicate interfaces between materials. Evidently, the bottom gate reduces the amplitude in the upper half of the membrane and thereby the outcoupling efficiency compared to the structures with just a top gate, consistent with what is observed in the experiment.

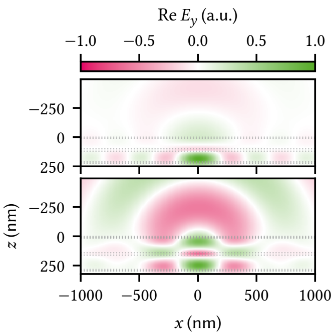


**Figure 4.4:** Real part of the electric field emitted by a current line located in the QW (black point) for the default (top) and optimized (bottom) structures with top and bottom gates. Optimizing the barrier thickness for absorption in the QW evidently also drastically improves the outcoupling efficiency into the halfspace $z < 0$.

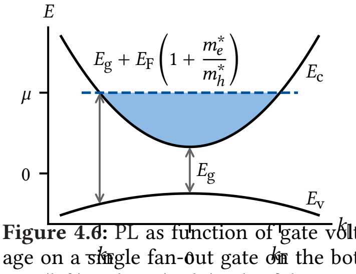


**Figure 4.5**

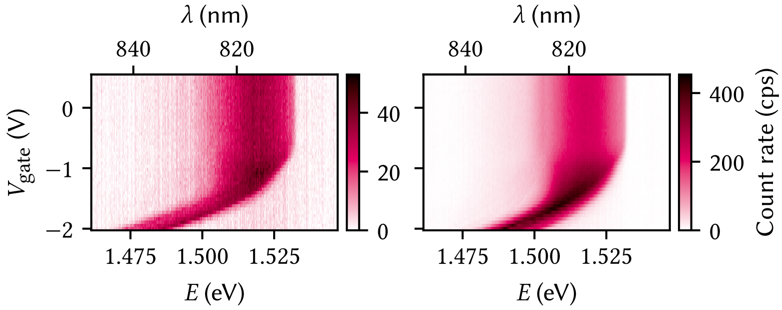


**Figure 4.6:** PL as function of gate voltage on a single fan-out gate on the bottom (left) and top (right) side of the membrane. The behavior is qualitatively similar but the overall quantum efficiency lower by an order of magnitude for gates on the bottom (as-grown buried) side.

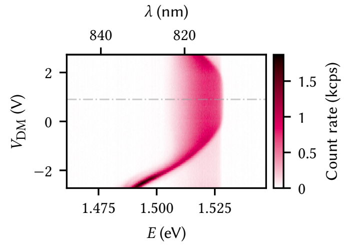


**Figure 4.7:** PL as function of difference-mode voltage on a large exciton trap. The observed Stark shift follows the expected quadratic dispersion, but is offset by 0.9 V with respect to zero bias (dash-dotted gray line). Remnant PL of the two-dimensional electron gas (2DEG) from outside the trap region is faintly visible below −1 V.

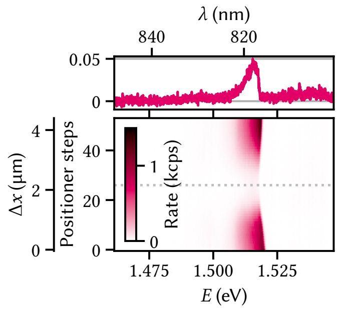


**Figure 4.8**

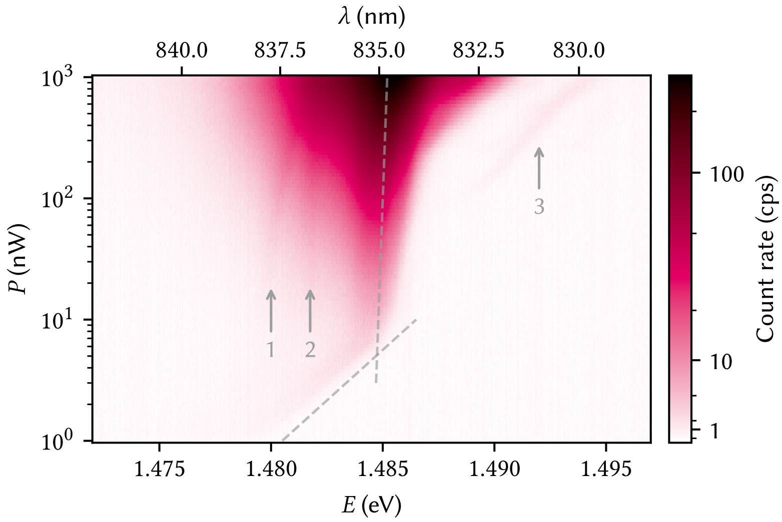


**Figure 4.9**

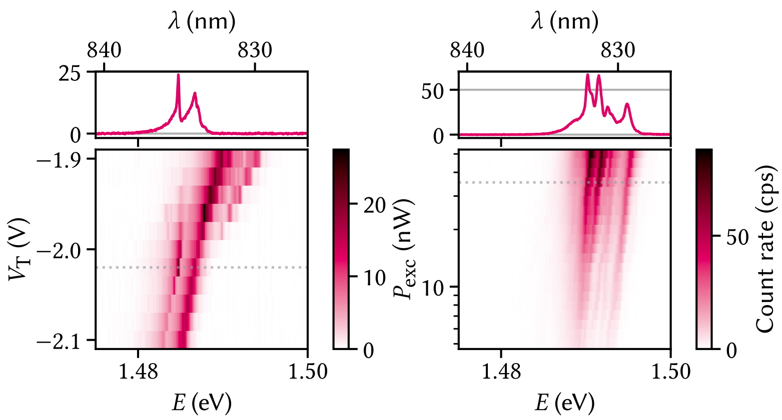


**Figure 4.10**

# 5 Conclusion & outlook

DOGS

# Part IV

# A Filter-Function Formalism For Unital Quantum Operations

# Appendix

# Additional TMM simulations

## A.1 Dependence on epoxy thickness

## A.2 Optimization of the barrier thickness

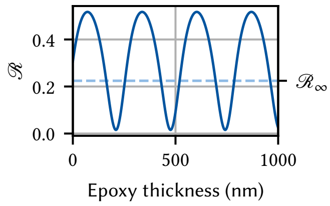


**Figure A.1**

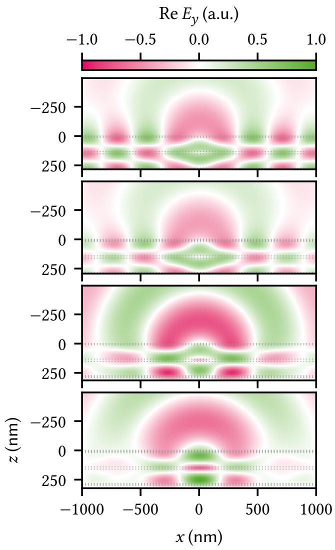


**Figure A.2**

# Bibliography


[1] Tobias Hangleiter et al., *Python-Spectrometer* version 2024.11.1, Nov. 21, 2024. Zenodo. DOI: 10.5281/ZENODO.13789861.

[2] `welch` — *SciPy v1.15.2 Manual.* URL: https://docs.scipy.org/doc/scipy-1.15.2/reference/generated/scipy.signal.welch.html (visited on 03/31/2025).

[3] E. T. Whittaker. "XVIII.—On the Functions Which Are Represented by the Expansions of the Interpolation-Theory." In: *Proc. R. Soc. Edinb.* 35 (Jan. 1915), pp. 181–194. DOI: 10.1017/S0370164600017806. (Visited on 04/22/2025).

[4] H. Nyquist. "Certain Topics in Telegraph Transmission Theory." In: *Trans. Am. Inst. Electr. Eng.* 47.2 (Apr. 1928), pp. 617–644. DOI: 10.1109/T-AIEE.1928.5055024. (Visited on 04/22/2025).

[5] C.E. Shannon. "Communication in the Presence of Noise." In: *Proc. IRE* 37.1 (Jan. 1949), pp. 10–21. DOI: 10.1109/JRPROC.1949.232969. (Visited on 04/22/2025).

[6] David M. Pozar. *Microwave Engineering*. Fourth edition. Hoboken, NJ: John Wiley & Sons, Inc, 2012. 1 p.

[7] F.J. Harris. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." In: *Proc. IEEE* 66.1 (Jan. 1978), pp. 51–83. DOI: 10.1109/PROC.1978.10837. (Visited on 03/27/2025).

[8] Denis Langevin et al. "PyMoosh: A Comprehensive Numerical Toolkit for Computing the Optical Properties of Multilayered Structures." In: *J. Opt. Soc. Am. B* 41.2 (Feb. 1, 2024), A67. DOI: 10.1364/JOSAB.506175. (Visited on 11/11/2024) (cited on pages 15, 16).

[9] Denny Dütz et al. "Distributed Bragg Reflectors for Thermal Isolation of Semiconductor Spin Qubits." In preparation (cited on page 16).

# Special Terms

**Numbers**
**2DEG** two-dimensional electron gas. 20

**A**
**API** Application Programming Interface. 6
**ASD** amplitude spectral density. 6

**C**
**CCD** charge-coupled device. 14

**D**
**DAQ** data acquisition. 2–4, 6, 7

**E**
**ENBW** equivalent noise bandwidth. 7

**G**
**GUI** graphical user interface. 10

**L**
**LIA** lock-in amplifier. 6

**P**
**PL** photoluminescence. 15, 19, 20
**PSD** power spectral density. 2, 4, 6–8

**Q**
**QW** quantum well. 17, 18

**R**
**RMS** root mean square. 7, 8

**T**
**TE** transverse electric. 15, 16
**TIA** transimpedance amplifier. 4
**TMM** transfer-matrix method. iii, 15, 16, 25

# Declaration of Authorship

I, Tobias Hangleiter, declare that this thesis and the work presented in it are my own and has been generated by me as the result of my own original research.

I do solemnly swear that:

1. This work was done wholly or mainly while in candidature for the doctoral degree at this faculty and university;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this university or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others or myself, this is always clearly attributed;
4. Where I have quoted from the work of others or myself, the source is always given. This thesis is entirely my own work, with the exception of such quotations;
5. I have acknowledged all major sources of assistance;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. Parts of this work have been published before as:


[1] Pascal Cerfontaine, Tobias Hangleiter, and Hendrik Bluhm. "Filter Functions for Quantum Processes under Correlated Noise." In: *Phys. Rev. Lett.* 127.17 (Oct. 18, 2021), p. 170403. DOI: 10.1103/PhysRevLett.127.170403.

[2] Thomas Descamps, Feng Liu, Tobias Hangleiter, Sebastian Kindel, Beata E. Kardynał, and Hendrik Bluhm. "Millikelvin Confocal Microscope with Free-Space Access and High-Frequency Electrical Control." In: *Rev. Sci. Instrum.* 95.8 (Aug. 9, 2024), p. 083706. DOI: 10.1063/5.0200889. (Visited on 08/12/2024).

[3] Tobias Hangleiter, Pascal Cerfontaine, and Hendrik Bluhm. "Filter-Function Formalism and Software Package to Compute Quantum Processes of Gate Sequences for Classical Non-Markovian Noise." In: *Phys. Rev. Research* 3.4 (Oct. 18, 2021), p. 043047. DOI: 10.1103/PhysRevResearch.3.043047. (Visited on 01/19/2022).

[4] Tobias Hangleiter, Pascal Cerfontaine, and Hendrik Bluhm. "Erratum: Filter-function Formalism and Software Package to Compute Quantum Processes of Gate Sequences for Classical Non-Markovian Noise [Phys. Rev. Research 3, 043047 (2021)]." In: *Phys. Rev. Res.* 6.4 (Oct. 16, 2024), p. 049001. DOI: 10.1103/PhysRevResearch.6.049001. (Visited on 10/16/2024).


Aachen, July 14, 2025.