`yaq`: Yet Another Acquisition

A modular approach to spectroscopy software and instrumentation

By
Kyle Foster Sunden

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy
(Chemistry)

at the
UNIVERSITY OF WISCONSIN - MADISON
2022

Date of final oral examination: 10 October 2022
This dissertation is approved by the following members of the Final Oral Committee:

John C. Wright, Professor, Analytical Chemistry
John F. Berry, Professor, Inorganic Chemistry
Robert J. Hamers, Professor, Analytical Chemistry
Etienne Garand, Professor, Physical Chemistry
Clark R. Landis, Professor, Inorganic Chemistry
Blaise J. Thompson, Instrumentation Scientist, Analytical Chemistry

Contents

List of Figures

List of Tables

Acknowledgments

To John, thank you for being the kind, patient advisor any graduate student could hope for.

When I asked John to join the group, he asked me why I wanted to join. My answer at that time was "Honestly, it's the people in the group". That answer rings true even as the composition of the group has changed. For that, this dissertation is dedicated to all of the Wright Group members.

To Blaise, thank you for your persistent pursuit of technical excellence. Thank you for starting so many of the projects that I got to work on in the Wright Group. But most of all, thank you for your friendship.

To Hannah, thank you for always knowing how to make me smile.

To Liz, thank you for your steadfast friendship.

To Laura, thank you for encouraging my interest in research.

To Tom and Dan, thank you for welcoming me into a community.

To my parents and Casey, thank you for being the kind, supportive family you are.

To all of those who contribute to and maintain Open Source scientific software, your work is appreciated and enables many great discoveries.

Abstract

Custom scientific instruments are often comprised of many smaller components, some purchased and some built by the researchers. As such, each component has unique command structure and available functionality which must be surmounted. In most scientific instruments, this challenge is overcome for a single particular instrument, performing a single task. The interface must be learned anew to incorporate the same hardware on a different instrument or for a new experiment. yaq is a hardware interface layer designed to decouple the deep knowledge of the component interface from the deep knowledge of the scientific application. This modular approach allows hardware components to be reused more easily without requiring that the direct interface is reimplemented. To accomplish this task, yaq uses a small background program, called a "daemon" which manages the direct hardware interface. A consistent set of self describing methods are then exposed for a "client" application to use. yaq standardizes common operations, making hardware that logically behaves similarly often interchangeable, while still enabling access to specialized functionality that may necessitate a particular component. While yaq was initially designed with the needs of the Wright Group in mind to control multiple laser systems and collect Coherent Multidimensional Spectroscopy data, it has found utility in multiple experimental fields, including collecting data from high pressure gas phase reactions and controlling flow rates of kinetics reactors using syringe pumps. While yaq can be used by small individual experiment specific client programs, its self description and interface consistency promises lend themselves to integration with larger orchestration layers. Herein, one such integration is described in detail as it is utilized by the Wright Group. Bluesky is a collaboration among several National Laboratory facilities to provide a powerful generic experiment orchestration interface. yaq provides an additional hardware interface layer for Bluesky. yaq is an open source project with an ethos of sharing the burden of developing hardware support across the community so that all can benefit. This practice provides a foundation of yaq hardware support that can be relied upon by experimentalists, and expanded when new hardware is desired. yaq is a growing ecosystem of tools built for experimentalists to interact with their instruments.

Part I

Background

Chapter 1

Spectroscopy

1.1 Introduction

Spectroscopy is the study of light interacting with matter. Light can be manipulated easily and provides a useful analytical tool for interrogating materials. There are readily available detectors to quantify the amount of light all the way from single photons to powerful bursts. Simple experiments can be performed with commodity equipment. The simplest spectroscopic experiments, called linear spectroscopies, are common analytical techniques which are taught to all budding chemists. Absorbance, fluorescence, reflectance, and raman spectroscopies are the most common techniques. By observing these four simple techniques, much information about the quantum states of a material can be obtained. Different techniques are sensitive to modes with different characteristics of the transitions, observing different selection rules.

Nonlinear techniques expand on the available information by interrogating coupling between multiple states. This allows for resolution of congested spectra [1, 2], and performing experiments which rely on states that would otherwise be disallowed by selection rules[3, 4]. These techniques, broadly categorized as Coherent Multidimensional Spectroscopy (CMDS), have many parallels to common NMR measurements[5, 6, 7]. In the same ways that multiple NMR pulses reveal coupling between nuclear spin states, so do multiple frequencies of light reveal coupling between optical and vibrational states of molecules. CMDS relies on creating coherences, or superpositions, of multiple states. For this reason, they are sometimes more colloquially referred to as "Shrödinger cat state spectroscopy"[8].

Most CMDS experiments are performed in the time domain, using broadband pulses which are used to create interferograms by scanning in time[9, 10]. This is similar in principle to Fourier Transform infrared absorbance spectroscopy. Time domain CMDS experiments inherit from the tradition of pump-probe and photon echo spectroscopies[11]. Time domain experiments are limited by the pulse bandwidth of a single excitation pulse. While much progress has been made in reliably producing broadband light sources[12], the available range remains small in comparison to the range of many electronic and vibrational states. Additionally, the heterodyne detection of time domain techniques requires a local oscillator to produce the interferometric measurement. This limits the mixing processes available for interrogation, as an output color equal to one of the input colors is required.

By contrast, frequency domain approaches are not limited in the same ways time domain approaches are. There is no need for a broadband light source, as in frequency domain approaches, each data point is collected independently with the incident light having a different combination of colors. This is sometimes referred to as "multi-resonant" CMDS or MR-CMDS[13]. Multiple light sources are used and independently controlled to produce the incident light. The output occurs at the sum or difference frequencies of the incoming light. I like to tell people that the layman's explanation of MR-CMDS is that "I do math with light". These techniques use homodyne detection, and therefore can detect light at new frequencies, opening the door to additional pathways to interrogate the coupling between quantum states. Frequency domain techniques grew out of the tradition of Raman spectroscopy, with one of the earliest examples, Coherent Anti-stokes Raman Spectroscopy (CARS) being comprised of two successive raman transitions[14].

While frequency domain experiments do address some of the shortcomings of the time domain experiments, they are not without their own challenges. First, there are many moving parts to frequency domain experiment. Each light source has multiple motors which result in light having slightly different optical path lengths at different colors, which means that the resultant delay must be externally compensated. This becomes primarily a challenge of orchestration. A complicated instrument requires sophisticated software to control the motors and present a useful parameterization to users. Additionally, there are artifacts of the data collection are present, and must be accounted for and understood. Common examples of such artifacts include: absorptive effects[15], window effects[16, 17], pulse effects[18], and group/phase velocity mismatch[19].

Experimentally, a frequency domain instrument requires two or more tunable light sources. These light sources are typically Optical Parametric Amplifiers (OPAs) or Optical Parametric Oscillators (OPOs). Additionally, the frequency domain experiment requires a controllable time delay to both ensure temporal overlap and act as a discriminating axis for experiments. The delays are not used in the same fashion as time domain experiments, but are still a crucial part of the instrument. All of the incident laser pulses must arrive at the sample overlapped in time and space, and high intensities are required to elicit nonlinear response, so focusing optics are important. Introducing a small time delay between successive pulses can often greatly increase signal to noise ratios because non-resonant background signal is diminished to a greater degree than the resonant material response.

Part II

Development

Chapter 2

WrightTools

This chapter was originally published in the Journal of Open Source Software [20]. WrightTools is a project that was started by Dr. Blaise Thompson circa April of 2015. Upon joining the Wright Group in late 2016 I began taking part in the maintenance and improvement of the library. In the summer of 2017, we undertook a major refactor of the library, which introduced the HDF5-based storage format. Upon Dr. Thompson's graduation in 2018, I have been the primary maintainer of the library. I would also like to recognize the contributions of other Wright Group members to WrightTools, particularly Dr. Daniel Kohler and Dr. Darien Morrow.

2.1 Introduction

"Multidimensional spectroscopy" (MDS) is a family of analytical techniques that record the response of a material to multiple stimuli—typically multiple ultrafast pulses of light. This approach has several unique capabilities:

- resolving congested states [21][2],
- extracting spectra that would otherwise be selection-rule disallowed [3][4],
- resolving fully coherent dynamics [22],
- measuring coupling [23],
- and resolving ultrafast dynamics [24][25].

In our view, the most exciting aspect of these techniques is the vast number of different approaches that scientists can take to learn about material quantum states. Often, a number of these experiments can be accomplished with a single instrument. The diversity of related-but-unique approaches to interrogating quantum systems is an important strength of MDS.

Advancements in optics and laser science are bringing ultrafast multidimensional spectroscopy to more and more laboratories around the world. At the same time, increasing automation and computer control are allowing traditionally "one-dimensional" spectroscopies to be recorded against other dimensions.

Due to its diversity and dimensionality, MDS data is challenging to process and visualize. The tools that scientists develop to process one experiment may not work when different experimental variables

are explored. Historically, MDS practitioners have developed custom, one-off data processing workflows that need to be radically changed when new experiments are undertaken. These changes take time to implement, and can become annoyances or opportunities for error. Even worse, the challenge of designing a new processing workflow may dissuade a scientist from creatively modifying their experimental strategy, or comparing their data with data taken from another instrument. This limit to creativity and flexibility defeats one of the main advantages of the MDS "family approach".

`WrightTools` is a new Python package that is made specifically for multidimensional spectroscopy. It aims to be a core toolkit that is general enough to handle all MDS datasets and processing workloads. Being built for and by MDS practitioners, `WrightTools` has an intuitive, high-level, object-oriented interface for spectroscopists. To our knowledge, `WrightTools` is the first MDS-focused toolkit to be freely available and openly licensed.

2.2 Challenges and Implementation

There are several recurring challenges in MDS data processing and representation:

- There are no agreed-upon file formats. Files generated by researchers may have inconsistent internal conventions, and they often fail to be fully self-describing.
- There is a great diversity of dataset types. The same instrument is capable of producing datasets with many different combinations of scanned hardware.
- Dataset size may be large enough to run into computer memory limits.
- Dataset dimensionality is large enough to represent challenges in human interaction and visualization.

The excellent Scientific Python ecosystem is well suited to address all of these challenges [26]. Numpy supports interaction with and manipulation of multidimensional arrays [27]. Matplotlib supports one, two, and even three-dimensional plotting [28]. h5py [29] interfaces with hdf5 [30], allowing for storage and memory-safe access to large multidimensional arrays in a binary format that can be accessed from a variety of different popular languages, including MATLAB and Fortran. `WrightTools` does not intend to replace or reimplement these core libraries. Instead, `WrightTools` offers an interface that impedance-

matches multidimensional spectroscopy and Scientific Python.

`WrightTools` defines a universal MDS data format: the `wt5` file. These are simply hdf5 files with certain internal conventions that are designed for MDS. These internal conventions enable the flexibility and ease-of-use that we discuss in the rest of this section. Instances of WrightTools's classes dynamically interact with the multidimensional spectroscopic data within these files. These classes are children of h5py classes. `WrightTools` offers a variety of functions that try hard to convert data stored in various other formats to `wt5`.

`WrightTools` defines a unique and flexable strategy of storing and manipulating MDS datasets. A single instance of the `WrightTools.Data` class is implemented as a group containing many separate arrays. There are two principle multidimensional array classes: `Channel` and `Variable`. Conceptually, these correspond to independent (scanned) dimensions—"variables"—and dependent (measured) signals—"channels". Channels typically contain measured signals from all of the different sensors that are employed simultaneously during a MDS experiment. Variables contain coordinates of different light manipulation hardware that are scanned against each-other to make up an MDS experiment. All variables are recorded, including coordinates for hardware that are not actually moved during that experiment (an array with one unique value) or other independent variables, such as lab time.

There can be many variables that change in the context of a single MDS experiment. The typical spectroscopist only really cares about a small subset of these variables, but exactly what subset matters may change as different strategies are used to explore the dataset. Furthermore, it is often useful to "combine" multiple variables using simple algebraic relationships to exploit the natural symmetry of many MDS experiments and to draw comparisons between different members of the MDS family [31]. In light of these details, `WrightTools` provides a high-level `Axis` class that allows users to transparently define which variables, variable relationships, and unit conventions are important to them for representation and manipulation. Each `Axis` contains an `expression`, which dictates its relationship with one or more variables. Given 5 variables with names `['w1', 'w2', 'wm', 'd1', 'd2']` , example valid expressions include `'w1'`, `'w1=wm'`, `'w1+w2'`, `'2*w1'`, `'d1-d2'`, and `'wm-w1+w2'`. Users may treat axes like multidimensional arrays, using `__getitem__` syntax and slicing, but axes do not themselves contain arrays. Instead, the appropriate axis value at each dataset coordinate is computed on-the-fly

using the given expression. Users may at any time change their axes by simply calling `transform` with new expressions.

`WrightTools` offers a suite of data manipulation tools with MDS in mind. Users can access portions of their data using high-level methods like `chop`, `split`, and `clip`. They can process their data using simple mathematical operations or more specific tools like `level`, `gradient`, `collapse`, and `smooth`. Users can even `join` multiple datasets together, creating higher-dimensional datasets when appropriate. All of these operations refer to the self-describing internal structure of the `wt5` file wherever possible. Users are not asked to refer to the specific shape and indicies of their data arrays. Instead, they deal with simple axis expressions and unit-aware coordinates.

`WrightTools` offers a set of "artists" to quickly draw typical representations. These make it trivial to make beautiful Matplotlib representations of MDS datasets. Again, the self-describing internal structure is capitalized upon, auto-filling labels (including units, symbols, and expressions) and auto-scaling axes. For higher-than-two dimensional datasets, `WrightTools` makes it easy to plot many separate figures that can be looped through using an image viewer or stitched into a looping animated gif. A convenience function, `interact2D`, allows users to explore a complete dataset using matplotlib's built-in widgets.

2.3 Availability

`WrightTools` is hosted on GitHub[32] and archived on Zenodo [33]. `WrightTools` is distributed using pip[34] and conda[35] (through conda-forge). Documentation is available at https://wright.tools[36].

2.4 Impact

`WrightTools` has directly enabled no fewer than eleven publications [25][37] [31][38] [19][39] [40][41] [42][43] [17]. Many of these publications have associated open datasets and `WrightTools`-based processing scripts which enhance the scientific community's ability to audit and reproduce the published work. Although these publications span several different MDS family members and instruments, the

common usage of `WrightTools` makes it trivial to download and immediately interact with the raw and processed datasets, and (when applicable) simulations that comprise these publications. These practices are not yet common in the MDS community.

Though still relatively uncommon, MDS is an increasingly important family of analytical techniques used by Chemists and Physists to interrogate especially complex systems and to answer especially challenging questions. By abstracting away common array manpulation, file management, and data visualization tasks, `WrightTools` promises to increase the productivity and creativity of MDS practitioners. We hope that `WrightTools`, and the universal `wt5` file format, will become a useful open source core technology for this growing community. We are particularly excited about ongoing projects that build on top of `WrightTools`, including packages for data acquisition and simulation [44][40].

2.5 Acknowledgements

The development of `WrightTools` has been supported by by the National Science Foundation Division of Chemistry under Grant No. CHE–1709060. D.J.M acknowledges support from the Link Foundation.

Chapter 3

Attune

Attune is a library for representing tuning data. This chapter is largely reproduced from the documentation of Attune. Prior to the major refactor of WrightTools in the summer of 2017, many of the concepts behind Attune were incorporated into WrightTools itself. The ideas for algorithms for tuning Optical Parametric Amplifiers derive primarily from the work of Dr. Blaise Thompson and Dr. Schuyler Kain. I have been the primary maintainer of Attune since its split from WrightTools. Wright Group members have provided valuable feedback on the design and continued evolution of Attune. Attune uses WrightTools data objects to generate and update motor mappings for Optical Parametric Amplifiers (OPAs) and for Spectral Delay Correction (SDC).

3.1 Installation

`attune` requires Python 3.7 or newer.

3.1.1 conda-forge

Conda[45] is a multilingual package/environment manager. It seamlessly handles non-Python library dependencies which many scientific Python tools rely upon. Conda is recommended, especially for Windows users. If you don't have Python yet, start by installing Anaconda[46] or miniconda[47].

```
conda config --add channels conda-forge
conda install attune
```
(3.1)

To upgrade:

```
conda update attune
```
(3.2)

3.1.2 pip

pip[48] is Python's official package manager. Attune is hosted on PyPI[49].

```
pip install attune
```
(3.3)

To upgrade:

```
pip install --upgrade attune
```
(3.4)

3.2 Attune Data Structures

The data structures in `attune` take inspiration from drawing parallels to nomenclature in music. The experimenter is the conductor of an orchestra of several `attune.Instrument` objects.

At a high level, the `attune.Instrument` is what users will directly interact with. An `Instrument` represents a collection of motors which must follow interpolated curves to produce one logical "one to many" mapping of logical position to motor positions. This collection can be as simple as a Spectral Delay Correction (SDC) mapping a color of light onto a single motor position to account for arrival time differences due to color of light. Alternatively, it can be a complex collection of several motors as in an Optical Parametric Amplifier (OPA), which requires all motors to be set to produce light of a selected color. When called like a function, the `attune.Instrument` provides a `attune.Note` which maps a given position to underlying motor positions.

An `attune.Instrument` may consist of several different modes (`attune.Arrangement` s) which can allow for things like different mixing processes in OPAs or multiple correction factors for SDC. Each `attune.Arrangement` in turn consists of `attune.Tune` objects, which provide individual mappings for input to output. By default a `attune.Tune` maps an input to a motor position or `attune.Setable`, however `attune.Arrangement` s may be nested allowing for references to lower level arrangements. This behavior allows the tuning curve for OPA mixing processes (such as Second Harmonic of Signal) to be built by adding one (or more) additional `attune.Setable` to an existing Signal `attune.Arrangement`. Parent `attune.Arrangement` s may override the position of `attune.Setable` s in the child arrangement.

These data structures are treated as "immutable" objects. This means that once created the values and the relationships of the objects are not changed. Instead, we have a system of `transitions` which provide *new*, updated `attune.Instrument` instances. This allows the context of how instruments were created to be preserved.

3.2.1 Setable

A `attune.Setable` consists of only two pieces of information: a name (a string) and a default position (None or string or float).

If the default is set, then any `attune.Note` which does not explicitly set that `attune.Setable` will inherit the default position. If there is no default, then the `attune.Note` will simply not specify the `attune.Setable` at all.

In most cases, `attune.Setable` objects are not required to be explicitly created, unless you wish to take advantage of default behavior.

`attune.Setable` provides an `attune.Setable.as_dict` method to allow for serialization.

```
no_default = attune.Setable("no_default")
default = attune.Setable("default", default=1.2)
```
(3.5)

3.2.2 Tune

A `attune.Tune` represents a continuous transformation from an independent variable to a dependent variable.

Currently the `attune.Tune` class assumes the independent variable is in units of `nm` to simplify the code. The dependent variable units can be specified using the `dep_units` kwarg to `attune.Tune.__init__`.

The `attune.Tune` object can be called as a function, which returns the linear interpolation of the independent to dependent variable mapping. The units of the input and/or desired output can be

specified using keyword arguments.

`attune.Tune` provides an `attune.Tune.as_dict` method to allow for serialization. `attune.Tune` also provides convenience attributes to access the limits of the tune: `attune.Tune.ind_min` and `attune.Tune.ind_max`.

```
tune = attune.Tune([400, 500, 600, 700], [0, 1, 4, 9], dep_units="mm")
val = tune(555) # returns 2.65
val = tune(555, dep_units="cm") # returns 0.265
val = tune(20555, ind_units="wn") # returns 0.86499635
```
(3.6)

3.2.3 DiscreteTune

A `attune.DiscreteTune` represents a discrete transform from a continuous independent variable to discrete string output dependent values.

Currently the `attune.DiscreteTune` class assumes the independent variable is in units of `nm` to simplify the code.

The outputs are stored as a dictionary of output key string to 2-tuple of ranges (min, max), and a default value as fallback. The dictionary is ordered, and the first valid range (inclusive of endpoints) is the value returned. Notably, this construction does limit each potential output to a single range, thus limiting (though not eliminating) the ability to have non-consecutive ranges which evaluate to the same output value. You can, however, place higher priority (earlier) ranges inside of other ranges to allow for some cases of non-consecutive ranges, as well as using default to get a similar effect.

`attune.DiscreteTune` provides an `attune.DiscreteTune.as_dict` method to allow for serialization.

```
dt = attune.DiscreteTune({"hi": (100, 200), "lo": (10, 20), "inner": (50, 60),
↪  "med": (20, 100)}, default="def")
dt(5) == "def"
dt(15) == "lo"
dt(20) == "lo"
dt(30) == "med"
dt(55) == "inner"
dt(70) == "med"
dt(100) == "hi"
dt(150) == "hi"
dt(500) == "def"
```

(3.7)

3.2.4 Arrangement

An `attune.Arrangment` provides a dict-like set of string names to `Tune` and `DiscreteTune` objects. The tunes may represent either a `Setable` (the default) or an `Arrangement` (when the `Instrument` contains an `Arrangement` of that name). When it represents an `Arrangement`, the `Instrument` will recursively evaluate for all `Setable` s.

All of the tunes must have the same independent units and must overlap (the former is easy since all tunes currently have `nm` units).

`attune.Arrangement` provides an `attune.Arrangement.as_dict` method to allow for serialization.

```
arr = attune.Arrangement("arr", {"continuous": tune, "discrete": dt})
```

(3.8)

3.2.5 Instrument

An `attune.Instrument` is the top level representation of the system, the one which users most directly interact with. An `attune.Instrument` provides a dict-like access to a set of `attune.Arrangement` s as well as a secondary dict of `attune.Setable` s. Additionally, `attune.Instrument` provide a system of tracking history via `attune.Transition` object (See also `Transitions`).

Most commonly, `attune.Instrument` objects are called like functions to provide `attune.Setable` positions (as a `attune.Note`) for a particular independent value. If the independent value is valid

for only a single arrangement, then the arrangement does not need to be specified. If, however, the independent value is valid for multiple arrangements, it must be specified.

The setables may be ignored if there is no need for defaults.

`attune.Instrument` provides both `attune.Instrument.as_dict` and `attune.Instrument.save` to allow for serialization.

```
tune = attune.Tune([0, 1], [0, 1])
tune1 = attune.Tune([0.5, 1.5], [0, 1])
first = attune.Arrangement("first", {"tune": tune})
second = attune.Arrangement("second", {"tune": tune1})
inst = attune.Instrument({"first": first, "second": second}, {"tune":
↪  attune.Setable("tune")})
inst(0.25)["tune"] == 0.25
inst(1.25)["tune"] == 0.75
inst(0.75) # raises exception because it is valid for both arrangements
inst(0.75, "first")["tune"] == 0.75
inst(0.75, "second")["tune"] == 0.25
```
(3.9)

Loading from files

The native format for an `Instrument` is JSON encodable as provided by `Instrument.save`. To read back an attune JSON file you can use `attune.open`.

```
instr = attune.open("instrument.json")
```
(3.10)

Alternatively, some formats such as Light Conversion TOPAS4 files can be parsed into `Instrument` s. TOPAS4 tuning curves are made up of multiple files which contain the information needed to recreate the `Instrument`, so the method points to a folder which contains the files.

```
instr = attune.io.from_topas4("path/to/topas4/")
```
(3.11)

3.2.6 Note

A `attune.Note` is the type returned when an `attune.Instrument` is called as a function. It is little more than a dict-like mapping of setable names to positions plus an indication of which arrangement was used to generate those positions. A `attune.Note` also contains a dictionary of setables for convenience.

3.3 The Attune Store

The Attune Store provides a timestamped history of saved instrument JSON files.

3.3.1 Saving an instrument

An `attune.Instrument` can be saved to the Attune Store by using `attune.store`. In order to use store, the instrument must have a name.

```
attune.store(instr)
```
(3.12)

If transitions have been applied in memory, the whole chain will be stored with a single call.

3.3.2 Retrieving an instrument

An individual `attune.Instrument` can be retrieved using `attune.load`, given its name.

```
attune.load("instr")
```
(3.13)

If you wish to select the instrument which was active at some time in the past, you can pass either a `datetime.datetime` object or a date string. If passed as a string, either a timestamp such as an ISO8601 format or certain phrasings of natural language can be passed. In general phrasing as "<X> <units> ago" is likely to yield good results.

Similarly if you want to find the *next* instrument object from a certain date, you can pass the date as well as set `reverse` to False to set the search direction to forward.

```
from dateutil import tz
import datetime
# Load the instrument from midnight UTC on July 15, 2022
attune.load("instr", datetime.datetime(2022, 7, 15, tz=tz.UTC))
# Load using a relative and natural language time
attune.load("instr", "3 days ago")
# Load the next instrument created after "3 days ago"
attune.load("instr", "3 days ago", False)
```
(3.14)

3.3.3 Listing available instruments

A list of available instrument histories can be obtained using `attune.catalog`.

```
attune.catalog()
```
(3.15)

By default, this provides a simple list of string names of instruments.

If you pass the argument `full` as `True`, then `attune.catalog` will instead return a dictionary of names to loaded `attune.Instrument` objects.

```
attune.catalog(True)
```
(3.16)

3.3.4 Instrument history

Since the attune store retains a permanent history, we have methods to interact with that history beyond simply loading

restore

`attune.restore` works exactly like `attune.load`, except instead of returning the instrument to use immediately, it returns the older instrument to the head (active) so that it will be retrieved with `attune.load` without additional arguments. In doing so, it applies a `restore` transition indicating the time passed in to restore it. Restoring to the currently active instrument is a no-op and so the time argument is required.

```
attune.restore("instr", "1 week ago")
instr = attune.load("instr")  # Now the same as it was 1 week prior
```
(3.17)

undo

`attune.undo` provides the instrument from prior to the latest transition. If the transitions have occurred in memory (i.e. not stored to the Attune Store) then it simply provides the previous instrument object directly. If instead the Instrument was loaded from the attune store, it retrieves the instrument that was stored just before itself from the attune store.

```
attune.undo(instr)
```
(3.18)

3.4 Transitions

`attune.Transition` is a class which represents a transformation from one `attune.Instrument` to another.

Each `attune.Transition` has a `attune._transition.TransitionType`, a string indicating what was done to create the `attune.Instrument`. If it exists, the `attune.Transition` also references the previous `attune.Instrument` object that was an input to whatever transformation was applied, as well as a JSON encodable dictionary of metadata to represent parameters that may be useful when

analyzing instrument history. Additionally, the `attune.Transition` can reference a WrightTools Data object that was used as an input to the transformation, if it exists. The data object is placed into the attune store when stored, but not included in the JSON serialization.

3.4.1 create transition

`create` is the default transition for a new `attune.Instrument` which is not based on a previous `attune.Instrument`.

This is the transtition that is applied when you create an instrument using the Python constructor (`attune.Instrument`).

3.4.2 map transitions

`map` transitions come in two flavors: `attune.map_ind_points` and `attune.map_ind_limits`. `map` transitions involve interpolating tunes onto new independent values using the outputs of the existing tunes.

`attune.map_ind_points` is given an array of new independent values to use as independent values for a particular tune in a particular arrangement.

```
out = attune.map_ind_points(instr, "arr", "tune", [400, 600, 800], units="nm")
```
(3.19)

`attune.map_ind_limits` is similar, but instead only cares about setting the bounding limits of the tune. Here, the number of points in the tune is preserved, but remapped onto a linear space from `min` to `max`.

```
out = attune.map_ind_limits(instr, "arr", "tune", 12500, 25000, units="wn")
```
(3.20)

Each of these optionally allow specifying units, and in the case of `attune.map_ind_limits` the points will be linearly spaced in the units provided, but converted to the native units of the instrument for

interpolation.

3.4.3 offset transitions

Like `map` transitions, `offset` transitions come in two flavors: `offset_by` and `offset_to`. `offset` transitions apply a static scalar offset to all dependent values in a tune.

For `attune.offset_by`, you provide the relative value of the change which is directly added to the dependent values of the specified tune.

```
## Add pi to the output values of the "tune" tune in the "arr" arrangement
out = attune.offset_by(instr, "arr", "tune", 3.14)
```
(3.21)

For `attune.offset_to`, you instead provide the absolute dependent value of the tune at a specific indepedent value of the specified tune.

```
## Offset by the scalar value which makes instr(532, "arr")["tune"] == 2.71
out = attune.offset_to(instr, "arr", "tune", 2.71, 532)
```
(3.22)

3.4.4 restore transition

`restore` is the transition which is created when you use `attune.restore` to bring an old instrument object back to the head of the Attune Store.

See `Store` for more information.

3.4.5 rename transition

`rename` is the transition created by `attune.rename`. Since the name is the key for the Attune Store, this transition breaks the history tracking, though the old name is provided for reference in the metadata.

```
out = attune.rename(instr, "out")
```
(3.23)

3.4.6 update_merge transition

update_merge is the transition created by attune.update_merge. This transition allows the merging of two instrument objects into a single instrument object. This is useful for operations such as Spectral Delay Correction, where an instrument is generated indpendent of a previous instrument, but must be integrated to logically group arrangements together.

```
out = attune.update_merge(instr1, instr2)
```
(3.24)

If the input instruments do not share any arrangements, then this operation is equivalent to simply creating a new instrument with all of the arrangements of both inputs. If the input instruments do share arrangements, then instr2 will take precedence on a tune by tune basis.

instr1 is considered the previous instrument and is used to determine the name field of the output instrument.

3.4.7 tuning transitions

There are four tuning methods which incorporate measured data to generate or update Instrument objects: attune.tune_test, attune.intensity, attune.setpoint, and attune.holistic.

3.5 Tuning Transitions

The following methods provide mechanisms of updating and generating attune.Instrument objects from measured data in the WrightTools Data format. This documentation assumes a working knowledge of the WrightTools Data format.

There are four generic methods for working up measurements which each serve to optimize for particular

use case. Each section will provide general usage tips as well as provide specific examples of workflows which use the function.

Because all of these methods have many available parameters, all parameters must be specified as keywords, even the required parameters. There are several parameters which are common to all or most of these methods. To avoid repetition, the common parameters are:

data (all) An input WrightTools Data object, properly formatted for the routine.

channel (all, though for holistic it is channels, plural) The channel from the data object to use as inputs to the workup routine.

arrangement (all) The name of the arrangement in the `attune.Instrument`.

tune (intensity, setpoint, and plural `tunes` for holistic) The name of the tune in the arrangement specified to update.

instrument (all, optional for intensity and setpoint) The `attune.Instrument` to update incorporating the data, if not given a new instrument is created.

level (optional for intensity, holistic, and tune_test) If `True`, then `WrightTools.Data.level` is called prior to interpreting the data from the selected channel.

gtol (optional for intensity, holistic, and tune_test) The "Global Tolerance" to ignore data below a multiplicative threshold of the maximum value in the dataset. This is used to cut out noise from the baseline of the data which may pull the selected centers away from their true value.

ltol (optional for intensity, tune_test) The "Local Tolerance" to ignore data below a multiplicative threshold of the maximum value in a particular slice. This is used to cut out side peaks or other artifacts that are above the global noise floor, but smaller than the desired peak to fit.

autosave (optional for all) This is a boolean of whether or not to save the output instrument and graphical representation into a folder or only keep in memory.

save_directory (optional for all) Specifies the location to save the instrument and graphical representation to

The methods all work by generating some spline of best fit through the space that it is optimizing, using the information from the data. As such, each of the methods also take additional keyword arguments which are passed into the `scipy.interpolate.UnivariateSpline`. This can be useful to fine tune the smoothness of the output to ensure desired outcome. One common use is to forgo smoothing all together by passing `s=0, k=1`, which makes the spline equivalent to a 1-D interpolator through each chosen point.

3.5.1 tune_test

A tune test allows for quick and easy evaluation that a given arrangement produces good quality light that is true to the color it says it is.

The scan for a tune test is the tune points of the arrangement vs a spectral (monochromator or array detector) axis, which is transformed to be the difference between the actual color and the expected color.

When the system is well tuned, the tune test will be flat at 0 along the differential axis and have adequate power over the expected usable range of the arrangement.

When the system is not well tuned, there will be deviations from 0 along the differential axis. If the tune test is significantly off, or if the intensity is diminished in the expected usable domain, then it is advisable to address the problem using other tuning strategies.

In general, systems which use a `attune.setpoint` or `attune.holistic` as one of the steps of the tuning procedure will generally not want to apply tune tests results, instead using it only as a diagnostic tool. This mostly applies to shorter pulse duration OPAs which have less separable motor space and wider bandwidth light, in the femtosecond regime. OPAs with longer pulse durations often can get by with simpler optimization routines using only `attune.intensity` to achieve adequate power over the usable tuning range, but `attune.intensity` does not provide any spectral information so `attune.tune_test` is used to ascertain the actual output color correctly.

`attune.tune_test` generates an updated curve by first identifying the actual color that each tune point

produces, using a spline to smoothly interpolate them, then interpolating each tune of the arrangement back onto the original tune points.

`attune.tune_test` accepts one additional parameter `restore_setpoints` which prevents the tune points from being interpolated back if it is set to `False`.

Figure 3.1 shows the plot associated with the following code:

```
data.transform("w3", "wm-w3")
out = attune.tune_test(
    data=data,
    channel="signal_mean",
    arrangement="sfs",
    instrument=instr,
)
```
(3.25)

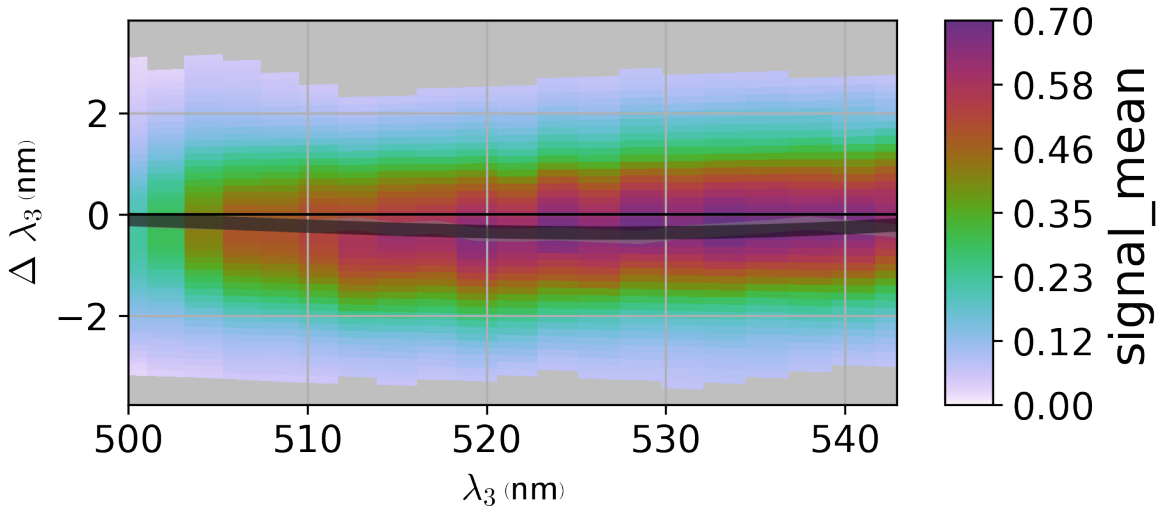

Figure 3.1: An example plot created by `attune.tune_test`.

The optimal tune test plot is flat at 0 deviation from expected color.

3.5.2 intensity

`attune.intensity` provides a mechanism to update a single `attune.Tune` in an instrument by optimizing the tune to provide the most intense position at each tune point.

When passing an `attune.Instrument` to `attune.intensity`, it is treated as updating the existing position by adding to the existing positions. This is the process when updating an OPA motor, which has been scanned as a differential from the previous expected position against the opa tune points. Any tunes other than the one specified as a parameter are ignored and kept the same as the input `attune.Instrument`. This method of tuning is usually sufficient for OPAs in the picosecond regime, where pulse widths allow each motor to be tuned independently. It is also used for later motors in femtosecond tuning procedures such as the `delay_2` motor of a Light Conversion TOPAS-C OPA or any additional mixing process after signal and idler have been generated.

Figure 3.2 shows the plot associated with the following code:

```
data.transform("w1=wm", "w1_Delay_2_points")
new = attune.intensity(
     data=data,
     channel=-1,
     arrangement="sig",
     tune="d2",
     instrument=old,
)
```
(3.26)

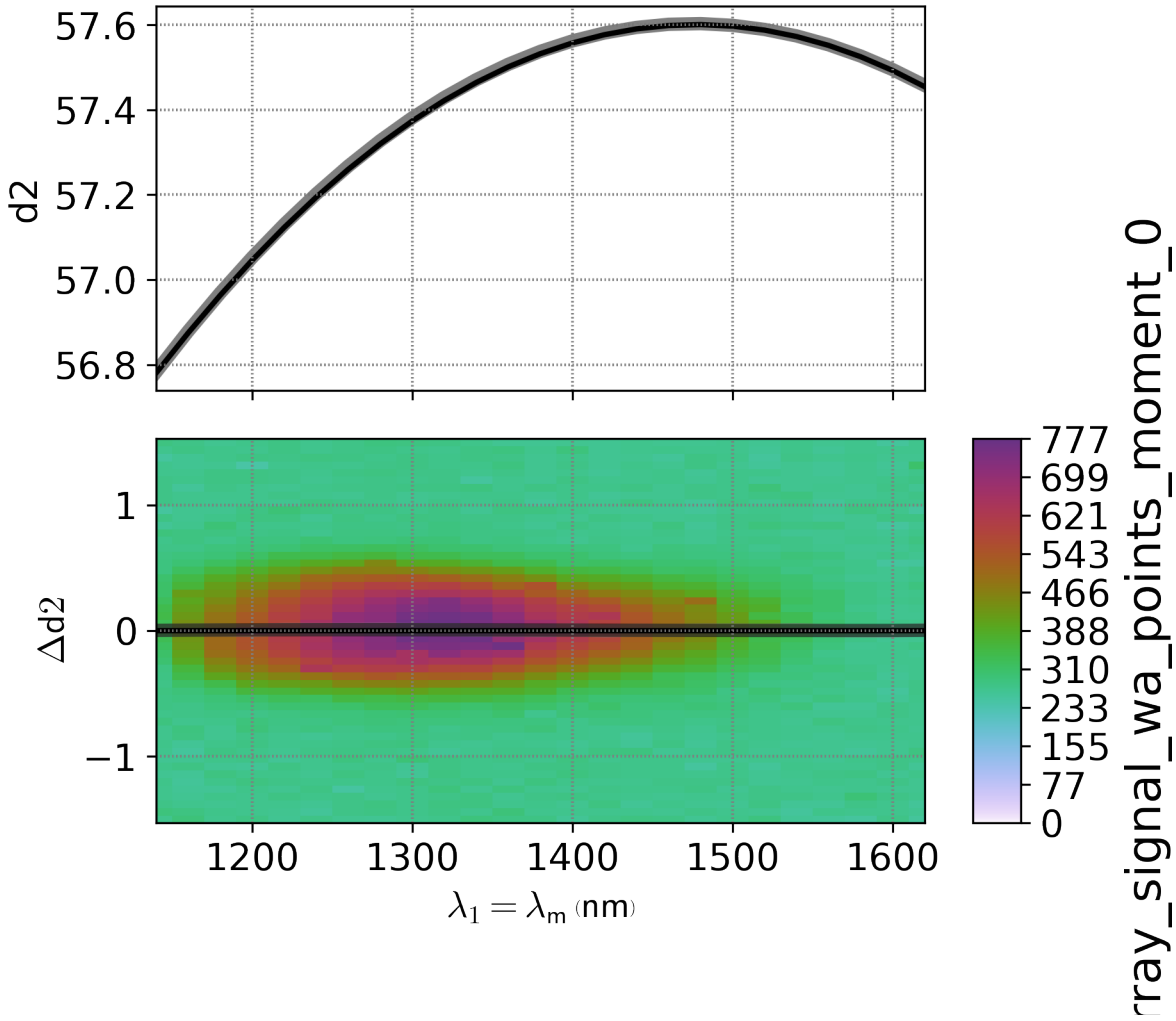

Figure 3.2: An example plot created by `attune.intensity`.

In the plot, the optimal position of the motor would be to follow the ridge of the most intense peak.

When no `Instrument` is provided, `attune.intensity` creates a new `Instrument` containing only the one tune. This is the process for generating a Spectral Delay Correction `Instrument` object. The scan for SDC is a delay position (usually centered around 0) versus the OPA tune points. Since the output instrument contains only a single arrangement with a single tune, `attune.update_merge` is often used to recombine the SDC output into an instrument which contains SDC tunes for alternate OPAs and arrangements of the same OPA.

3.5.3 setpoint

Instead of optimizing for output intensity, `attune.setpoint` optimizes for the correctness of the expected output color. This is useful for motors in femtosecond OPAs which when perturbed change the overall intensity little, but strongly affect the color produced, such as the Light Conversion TOPAS-C `crystal_2` motor.

The scan is nearly identical to the scan required for `attune.intensity`, however since it is optimising color information, a spectral axis (either via scanning a monochromator or via an array detector) must be used. The data must be transformed to (setpoint, differential_motor_position) The channel must be pre-processed to contain the color information, rather than intensity information. This is usually done by taking the `WrightTools.data.Data.moment` with `moment=1` along the spectral axis of the scan.

Figure 3.3 shows the plot associated with the following code:

```
data.transform("w1=wm", "w1_Crystal_2_points", "wa-w1")
data.level(0, 2, 5)
data.array_signal.clip(min=0)
data.transform("w1=wm", "w1_Crystal_2_points", "wa")
data.moment("wa", moment=1, resultant=wt.kit.joint_shape(data.w1,
↪  data.w1_Crystal_2))
data.transform("w1=wm", "w1_Crystal_2_points")
data.channels[-1].clip(min=data.w1.min() - 1000, max=data.w1.max() + 1000)
data.channels[-1].null = data.wa.min()

out = attune.setpoint(
    data=data,
    channel=-1,
    arrangement="sig",
    tune="c2",
    instrument=instr,
)
```

(3.27)

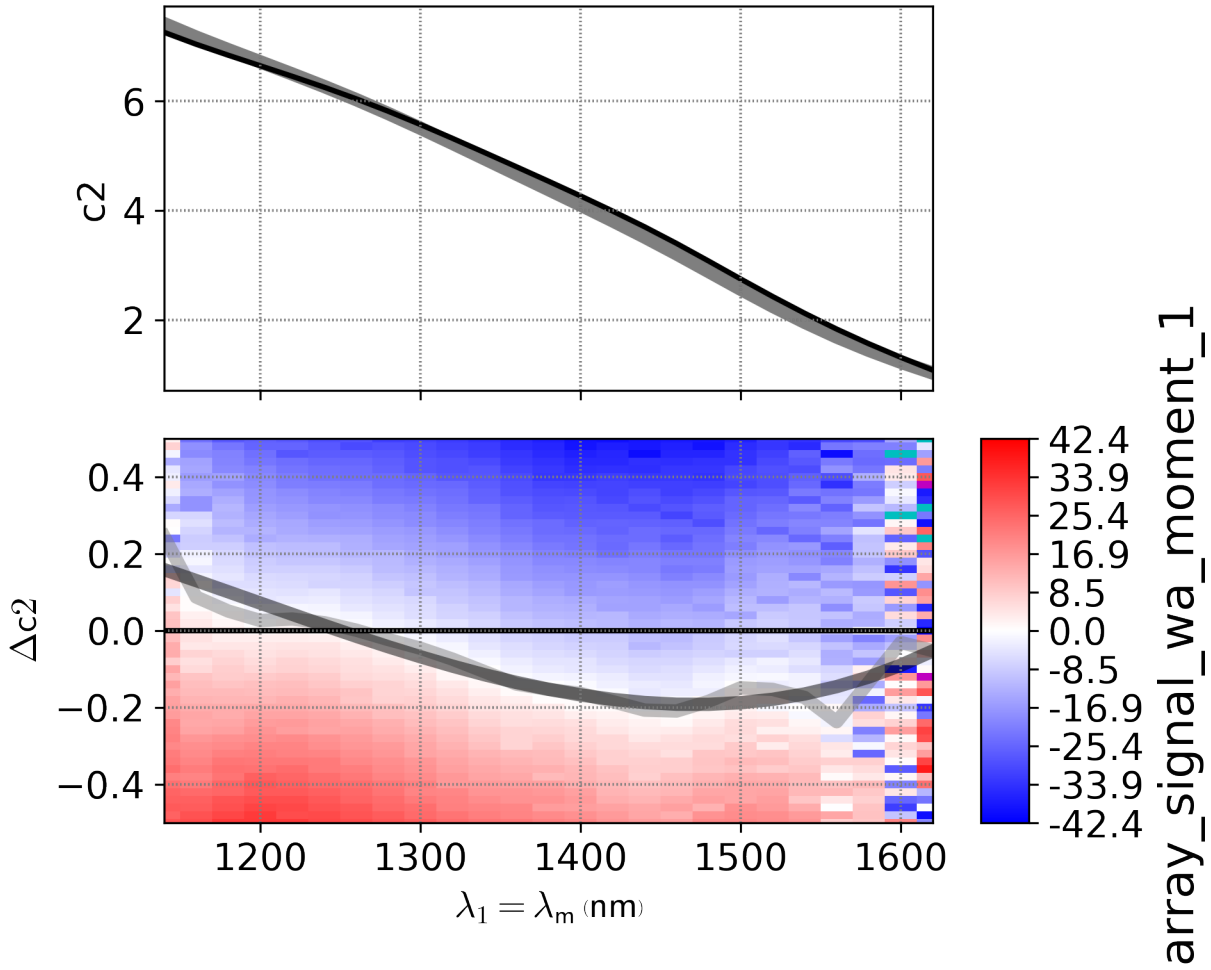

Figure 3.3: An example plot created by `attune.setpoint`.

In the graph, the color represents deviation from the expected color and pure white is the optimal motor position.

3.5.4 holistic

`attune.holistic` takes a multidimensional approach by using both intensity and color information to optimize two motors at once.

This is most useful for femtosecond OPAs where some motors are not separable due to bandwidth of the pulse. As such for the Light Conversion TOPAS-C OPAs, this is used to tune the "preamp" or `crystal_1` and `delay_1`.

The scan for holistic actually looks very similar to the scan for setpoint, including the OPA setpoint axis, a differential motor axis, and a spectral axis (which could be from an array detector). However, instead of being transformed to include the OPA setpoints, the transform is applied such that *two* motors are in the transform.

`attune.holistic` can either be handed separate intensity and spectral channels (as a 2-tuple `channels` argument) if separate preprocessing outside of the scope of the method is required. In this case, it expects each channel to be two dimensional and no spectral axis to be present in the channels or transform of the data. If it is given a single channel it expects that the spectral axis to be present and will take the 0th and 1st `WrightTools.data.Data.moment` to get intensity and spectral information, respectively) The spectral axis is assumed to be the last axis of the transform as provide, but can be overridden using the `spectral_axes` parameter.

The algorithm for `attune.holistic` starts by clipping data below the `gtol` for the amplitude channel, applying the clip to both the amplitude and color channels (as you cannot get a reliable color estimate from values below the noise threshold). It then creates a `LinearNDInterpolator` for each of the amplitude and spectral channels. It finds the point on each edge of the Delaunay interpolation triangles which are the color of each tune point. If enough points are found that are the requested color, it fits the points to a Gaussian function using the intensity information, one Gaussian for each dimension. It then splines each motor against the input color, and replaces the tunes in the input instrument with the

new splined positions.

In principle, this algorithm generalizes to an arbitrary number of dimensions, however the plotting step in particular only works for 2 dimensional data.

Figure 3.4 shows the plot associated with the following code:

```
data.transform("w1_Crystal_1", "w1_Delay_1", "wa")
out = attune.holistic(
    data=data,
    channels="array_signal",
    arrangement="NON-NON-NON-Sig",
    tunes=["c1", "d1"],
    instrument=instr,
    gtol=0.05,
    level=True,
)
```

(3.28)

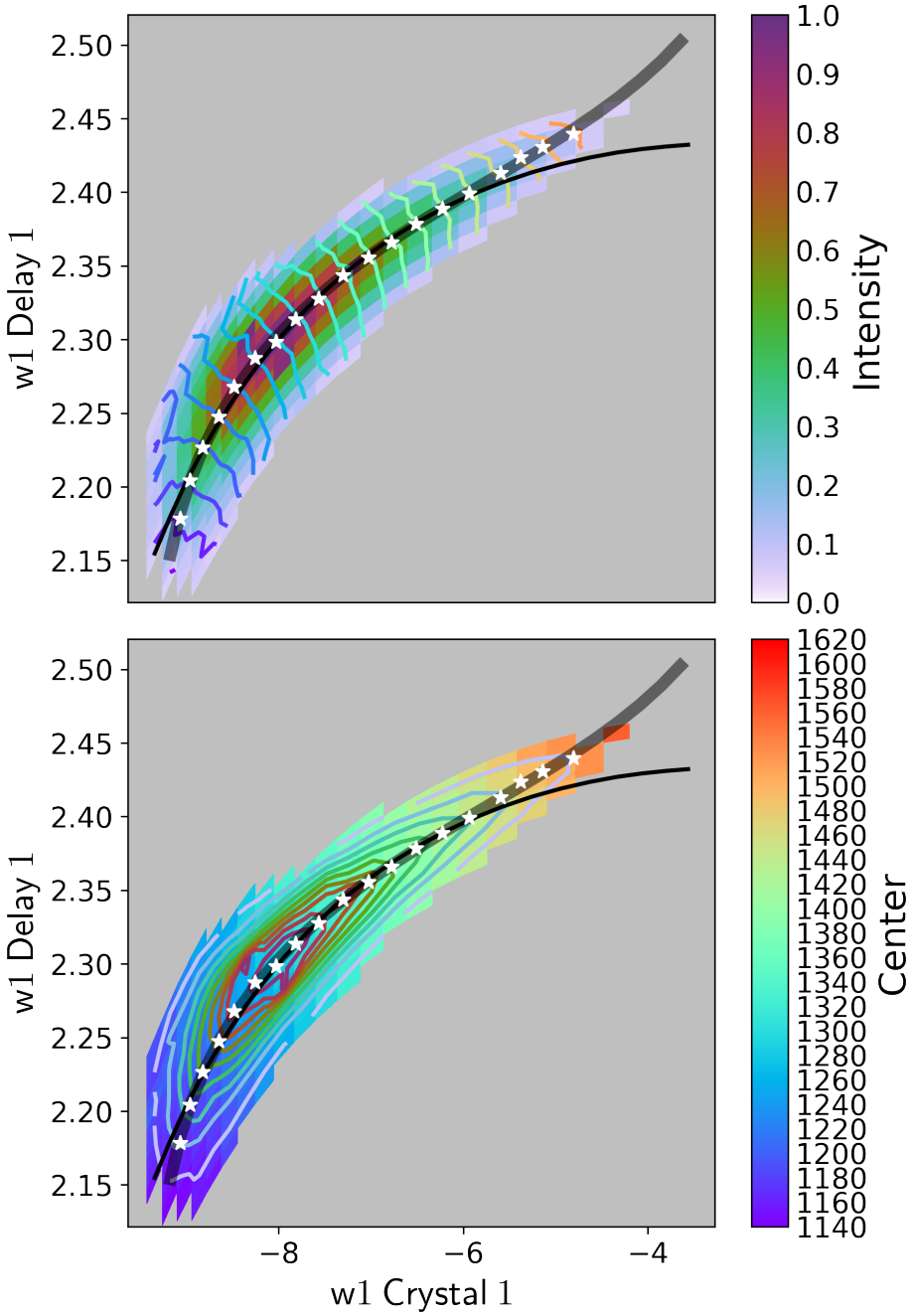

Figure 3.4: An example plot created by `attune.holistic`.

The axes of both plots are in 2D motor-space. The top plot is an intensity plot with contours of constant color overlaid. The bottom plot is the inverse: color plot with contours of constant intensity overlaid. The thin black line is the input path through 2D motor space. The thick semitransparent line is the output selected path through 2D motor space. The stars are the points which were selected by the algorithm for each tune point. Ideally, the thick black line would be along the ridge of the "slug" shape.

Chapter 4

`yaq`

4.1 The yaq Project: A Protocol for Scientific Instrumentation

Modern instrumentation development often involves incorporation of many dissimilar hardware peripherals into a single unified instrument. Increasing availability of modular hardware has brought greater instrument complexity to small research groups. This complexity stretches the capability of traditional, monolithic orchestration software. In many cases, a lack of software flexibility leads creative researchers to feel frustrated, unable to perform experiments they envision. Herein we describe yaq, a software project defining a new standardized way of communicating with diverse hardware peripherals. yaq encourages a highly modular approach to experimental software development, which is well suited to address the experimental flexibility needs of complex instruments. yaq is designed to overcome hardware communication barriers that are insurmountable with typical experimental software. A large number of hardware peripherals are already supported, with tooling available to expand support. The yaq standard enables collaboration among multiple research groups, increasing code quality while lowering development effort.

4.1.1 Introduction

Instrumentation development is a key part of the scientific enterprise. Novel instruments are typically constructed of many individual components that are both purchased and home-built. Orchestration software must communicate with each hardware component in the course of a scientific experiment. This can involve utilization of many interfaces: NI DAQmx[50], SCPI[51], ModBus[52], PICam[53], Thorlabs APT[54], among many others. The challenge of integrating all of these interfaces is a frustrating piece of the modern instrument development process. Weeks can be spent just integrating one new component into an existing project. In small academic labs, these software interfaces are typically created by student researchers without software development experience. Student researchers rarely focus on software reusability, and a lack of maintenance and documentation can make such software more difficult to use as time goes on. Scientists may struggle to rapidly innovate on their experimental design when each hardware addition requires major software development.

Some large user-facilities have addressed interface complexity via the adoption of unified standards,

such as EPICS [55] or TANGO [56]. The unified standards define a network interface for any hardware component. Orchestration software can target these unified standards for reading and writing hardware state. Small background services are written to translate the myriad component interfaces into the standard EPICS IOCs and TANGO Devices. These programs are performant, open source, and have huge libraries of existing hardware interface support, but require expert management to set up and provide descriptions via a separate server program. In our experience EPICS and TANGO do not scale well to single-investigator lab environments.

As smaller research labs have grown in experimental complexity, many individual labs have created domain-specific orchestration software. In the last few years, several open source projects by-and-for small-scale experimentalists have grown in popularity[57, 58, 59, 60, 61, 62, 63, 64]. While this growth is encouraging, many of these are limited by their focus on particular types of hardware or particular experimental domains. Most small custom research instrumentation continues to rely on monolithic software which has hard-coded interface support for each particular connected device. These monolithic applications tend to be inflexible and difficult to develop.

We have created a new network-based communication standard for scientific instrumentation, `yaq`. This standard borrows the most important ideas from established projects used by large user facilities while retaining the simplicity appropriate for small research labs. We have built this standard to be self-describing, portable, and reusable wherever possible. Our primary goal has been to make an interface which simplifies orchestration software development as much as possible.

Here, we discuss the design of the `yaq` standard in the context of four challenges facing instrument designers. First, we discuss how particularly challenging hardware interfaces can become seemingly insurmountable barriers to software control. Next, we discuss how inflexible orchestration software can limit experimental creativity. Then, we focus on challenges that arise when enhancing or modifying existing instruments with new hardware. Finally, we discuss the heavy software maintenance burden that many instrument designers face. In each case, we will highlight how the `yaq` project is designed to alleviate that challenge. Several case studies provide a view into the flexible ways that `yaq` can be applied to perform different scientific experiments. This paper provides an overview of the concepts and motivations behind `yaq`. Refer to the `yaq` website for a formal specification of the `yaq` standard.[65]

4.1.2 Hardware Interface Challenges

In this section we describe the architecture of the yaq framework in light of three major barriers that we have encountered in scientific instrumentation development. First barrier: Multiple interfaces are used to communicate with each component of the system. A fully automated system must be able to use all of these interfaces, a daunting task for scientists who do not specialize in software development. Second barrier: Certain specialty hardware have inconvenient interface requirements. A camera will only work with an obsolete interface card and drivers for Windows XP. A data acquisition manufacturer provides an Application Programmer Interface (API) that only works in Python 3.7. A graduate student wishes to drive several stepper motors using a Raspberry Pi. Third barrier: Some hardware interfaces are blocking. A graphical user interface stalls while waiting for a camera to collect data. Custom orchestration software needs to be closed before the manufacturers configuration software can be used. A graduate student finds themself needing to master advanced concepts in concurrency in order to orchestrate many motors performantly.

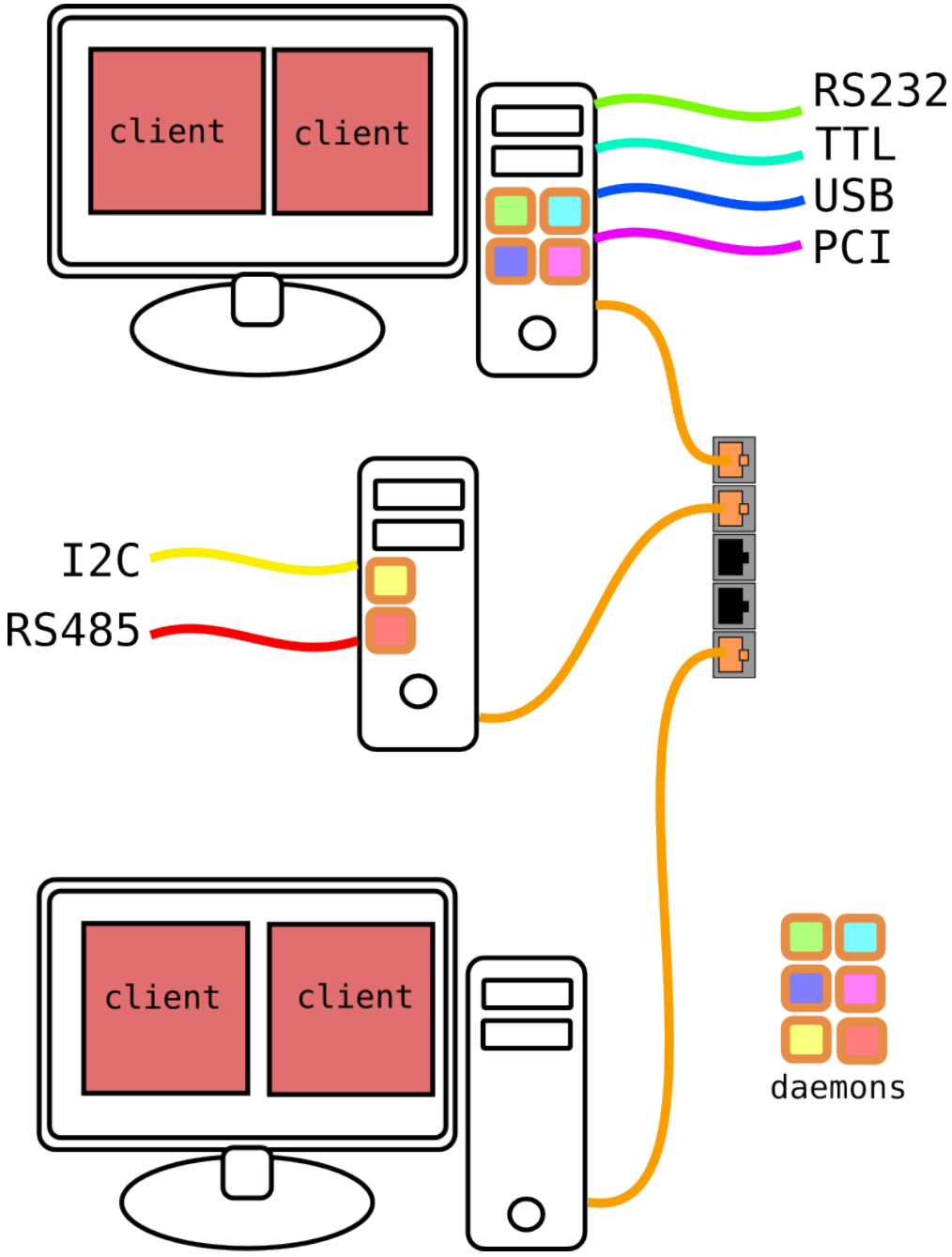

Figure 4.1: Networking diagram

Figure 4.1 diagrams the yaq architecture. Here, we show three different computers connected via an Ethernet network. The top and bottom computer are connected to monitors for interactive use while the middle computer is only accessible via the network. This might represent a complex scientific instrument involving several operator terminals as well as embedded computers. At the top, a single computer is connected to four hardware peripherals through RS232, TTL, USB, and PCI as indicated by the colored lines. That same computer is running four separate programs, one for each peripheral. These small, targeted, programs are managed by the operating system and run in the background. It is conventional to call such programs "daemons"[66]. The middle computer is connected to two additional peripherals, and runs daemons for each. Besides communicating with the hardware peripheral, each daemon can communicate with other programs, "clients", through the network. The four client programs shown in Figure 4.1 can each communicate with all six hardware peripherals shown. As an example, a client running on the bottom computer could communicate with the RS232 peripheral shown in green via the following path: client $\leftrightarrow$ network switch $\leftrightarrow$ top computer $\leftrightarrow$ daemon $\leftrightarrow$ hardware peripheral. This powerful architecture can be used on a single computer or used across many networked computers, including fully remote operator interfaces. This client-server architecture offers similar network capabilities to EPICS and TANGO. As we will show, usage of standards and the creation of tooling makes this architecture accessible to instrument builders outside of large facilities.

In yaq, communication between daemons and clients is performed over TCP/IP using Apache Avro RPC [67]. Avro provides an agreed upon standard for efficient serialization of data and method calls from a remote (client) process. Practically, the yaq interface looks like a collection of methods or functions, which Avro calls "messages". Each message has defined input parameters and output return types. A sensor might implement a message called "`get_measured`" which takes no parameters and returns a dictionary mapping channel names to numeric or array data. A motor would implement a pair of messages for setting and reading back the motor position: "`set_position(float position)` $\rightarrow$ `null`" and "`get_position()` $\rightarrow$ `float`". These self describing messages make up the lowest level functionality of the yaq interface. Each individual communication between client and daemon involves one message being requested by the client and the response returned from the daemon. Each daemon supports a collection of messages for its unique functionality, called the "protocol".

yaq introduces a concept called "Traits" which are collections of related messages that are shared among

multiple protocols. Motors implement the "has-position" trait, which defines "set_position", "get_position", and "get_units". Sensors would implement the "is-sensor" trait which defines "get_measured", "get_channel_names", and "get_channel_units". Protocols which implement a trait must support all of the messages from the trait. Importantly, specific protocols can also implement arbitrary additional messages that are not defined by any trait.

The first hardware interface barrier: Multiple incompatible interfaces are used to communicate with each component of the system. yaq provides a unified TCP/IP interface to all hardware peripherals based on the well-described Avro RPC protocol. The trait system was introduced in pursuit of our primary goal of easing the client development process. Clients can trust that protocols that implement a given trait will behave in similar ways. The standardized yaq interface presents the same set of interactions for client-side scientific code, simplifying the experience of using hardware.

The second hardware interface barrier: Certain specialty hardware have inconvenient interface requirements. Since yaq enables multiple machines, any hardware requirements can be addressed by putting a machine for that specific hardware on the network. The Raspberry Pi which drives several stepper motors can be placed onto a private network to communicate with the primary instrument computer using yaq. Because each yaq daemon is running in its own process, the software environment can be tailored to its needs. A client running up to date Python 3.10 communicates seamlessly with a daemon running Python 3.7.

The third hardware interface barrier: Some hardware interfaces are blocking. Monolithic orchestration software often necessitates separate threads for each component hardware interface, a fragile pattern in which small mistakes become both critical and elusive errors. In yaq, each daemon is only responsible for a single hardware interface. It is expected that each message call over the yaq interface will return rapidly, ensuring that client applications are not blocked for extended periods of time. This principle applies even when the message starts an action that might take several seconds to complete, such as homing a motor or initiating a measurement for a sensor. In these instances, the initial message simply starts the action and returns, with a separate message provided to retrieve results when they exist. In order to know how long to wait, the "is-daemon" trait provides a message called "is_busy" which should return "true" while the long running action is not complete, and "false" once it is finished. Additionally,

multiple clients can communicate with the same daemon simultaneously. A complex instrument may involve multiple operators watching sensor data in real time, while one program is orchestrating the hardware and recording the data.

4.1.3 Experimental Flexibility

Existing experimental orchestration software is often highly inflexible. An experimentalist will spend many hours in lab repeating acquisitions because it is too challenging to add repetition functionality to their software. A laser lab needs to spend weeks on software development when introducing a single new step into their experimental procedure. Researchers are disappointed to realize that they are forced to start from scratch when developing software for a similar instrument built with trivially different hardware.

In our view, software inflexibility is a natural consequence of the typical software development practices used by custom instrument builders. Instrumental software is often built as one monolithic program that does everything from providing a graphical interface, through hardware interfacing, and writing data files. Such software is typically impossible to debug without access to real hardware, often requiring all of the hardware to be available to simply start the program. As such, instrumentation software development time is in conflict with valuable data acquisition time. The hardware interfaces that these programs implement are typically made quickly and without regard to standardization with similar hardware. The orchestration routines are intimately tied to the particular hardware configuration of one instrument.

Unique experiments will always need custom orchestration and user experience. We believe that novel instrumentation development naturally and necessarily includes the creation of targeted software. Developing experimental software is an iterative process tied to the scientific goals of the instrument. Often experimentalists must apply their specialty scientific knowledge to develop this software.[68] `yaq` is architected to encourage better software development practices when creating such programs. In a `yaq` context, orchestration code and graphical control interfaces are implemented as clients. These clients are automatically simpler because they only need to implement the `yaq` standard and do not need to include the vast array of hardware-specific communication interfaces. Beyond this, clients can use traits to interact with similar hardware identically. A client written to perform a two dimensional fluorescence

experiment using two Acton monochromators will also work with Horiba monochromators without any modification.

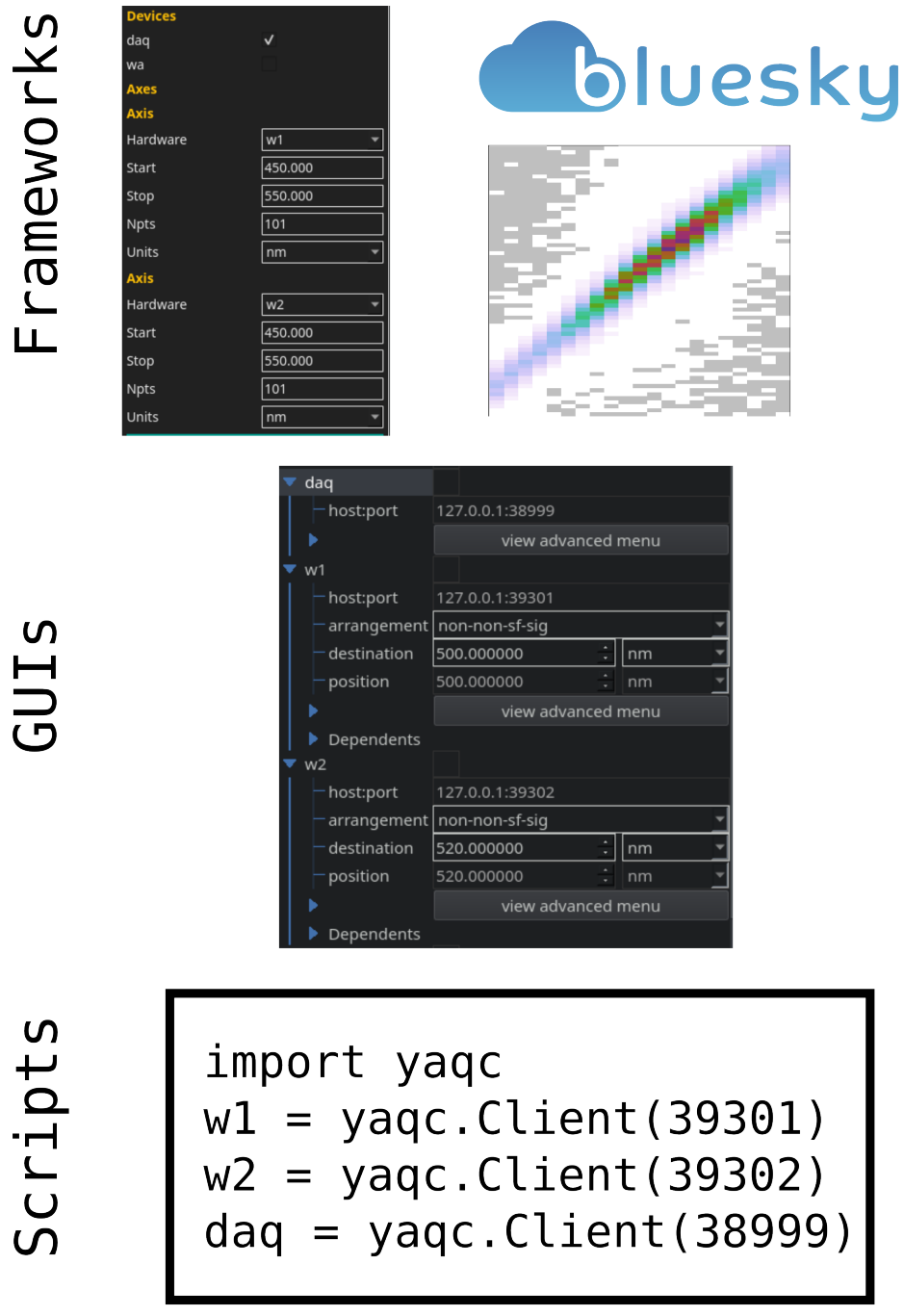

Figure 4.2: yaq scales from simple scripts to integrations with featureful frameworks.

A yaq client can scale in complexity from a small, lightweight script all the way to a sophisticated graphical program, as shown in Figure 4.2. At the bottom, we represent the most lightweight interface to yaq, Python scripting. yaqc[69] is a Python client which is excellent for using in scripts or any other Python program. The code shown creates three client objects which could be used directly through an interactive Python prompt or in a reusable script. Each client object provides Python methods for each Avro message specified by the associated protocol.

In the middle, the same three hardware peripherals are represented in an interactive, graphical form. yaqc-qtpy[70] is a graphical application which builds interactive controls based on traits for any conceivable yaq protocol. The self describing yaq interface is used to provide graphical elements for the most commonly used messages. This is an invaluable tool which provides a "free" graphical user interface (GUI) to any daemon.

At the top, we show several features associated with the integration between yaq and the Bluesky project[71]. Bluesky provides a powerful orchestration layer for conducting and recording data for a wide variety of experimental procedures. Shown in Figure 4.2, we see one such GUI for parameterizing a Bluesky procedure and a representation of the resultant data. yaqc-bluesky[72] provides a bridge to the Bluesky ecosystem. Similar translation layers could be built for a variety of orchestration layers such as PyMoDAQ[57], Instrumental[58], or TRSpectrometer[59].

All three types of clients represented in Figure 4.2 have been shown addressing the same three hardware peripherals. All types of clients can be used simultaneously to interact with the same instrument in different modes. Client sophistication can be introduced naturally, as novel experiments are tested and refined. Different clients can specialize for different requirements of a custom instrument. For example, yaqc-qtpy can provide a quick interface for setting and viewing hardware positions while Bluesky can focus on experimental data acquisition.

The Landis Group at UW-Madison is currently working on a new type of flow reactor: the Wisconsin Quench Kinetics Reactor (WiQK). This reactor incorporates several computer-controlled valves and syringe pumps as well as various sensors. The set of hardware peripherals is rapidly changing as researchers continue to test and refine their design. Only a few researchers are actively using the reactor during this prototyping stage. These researchers are experimentalists who have limited background in software

development. The Landis Group has written basic Python scripts to orchestrate hardware for their reactor. These lightweight scripts can be extensively refactored by the experimentalists as the hardware and orchestration strategy changes dramatically during WiQK development. This approach ensures that the Landis Group is not slowed down by complex, inflexible orchestration software. Once the reactor is complete, more sophisticated graphical clients will be created to accommodate end users who were not involved as the reactor was built.

The Wright Group at UW-Madison needs to orchestrate a large variety of hardware in multidimensional scans for their complex spectroscopy experiments [73, 23]. This need for exquisite hardware control has resulted in several prior attempts at "home-built" orchestration software [74, 75, 76, 13, 77]. Now, using `yaq`, the Wright Group has been able to move to Bluesky rather than inventing their own sophisticated control software "from scratch". The Wright Group uses simulated hardware to enable client development away from the active laboratory computers. This allows Wright Group researchers to create polished client interfaces without interrupting ongoing experiments. Clients developed by the Wright Group have proved flexible enough to be used on four laser systems, each with different complements of hardware. Moving forward, the Wright Group will spend less energy developing control software and more energy developing creative spectroscopy experiments.

4.1.4 Incorporating New Hardware

In a `yaq` context, new hardware can be incorporated into an instrument through the addition of a new daemon. The `yaq` architecture simplifies hardware interface development in several ways. First, because daemons are separate and portable programs, the development effort can be spread across the community of `yaq` users. Often, researchers can download an existing daemon rather than writing a new one. Second, `yaq` daemon development can be performed separated from the particulars of any individual client. Often, this means that initial hardware enablement work can be done on a researchers personal machine before the new hardware peripheral is installed in the instrument. Third, when developing trait-compliant protocols, it becomes easy to design and fully test your hardware interface. Traits are unambiguous and well-described, making an obvious target for development. Tooling exists to verify full trait compliance, for example you can use `yaqc-qtpy` to provide a graphical program to interact with

your hardware immediately. Finally, as discussed in Section II, `yaq` gives you options to design using remote hardware or unusual interfaces when necessary.

We have created several tools to aide in daemon development. First, a Python library, `yaqd-core`[78], which implements shared functionality. Second, `yaq-traits`[79] is a command line application which allows the description of messages provided by a `yaq` protocol to be written in a human-readable fashion and translated into a more fully described machine readable format. The format it generates is an important part of how `yaq` protocols are self describing. This shields developers from the details of Apache Avro, which can be somewhat esoteric.

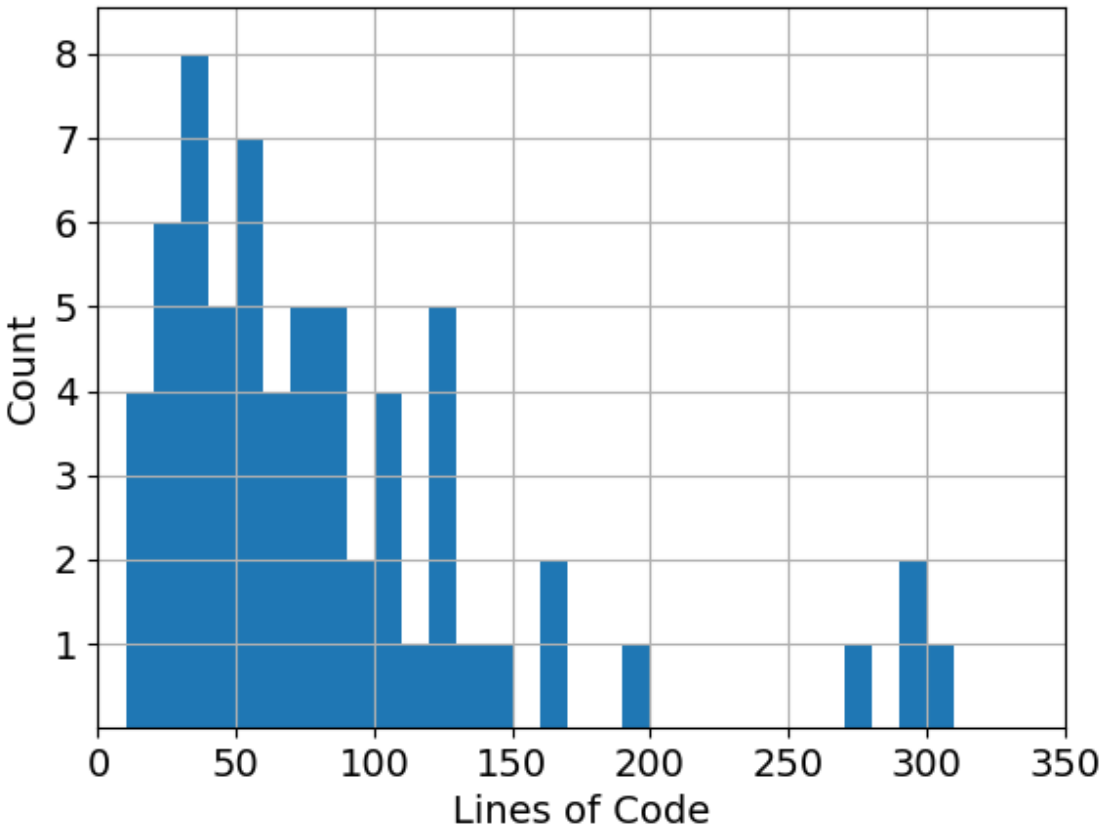

Figure 4.3: Histogram of the number of lines for each implemented daemon. Some daemons are implemented in ways that share code, resulting in apparent line counts less than 10. For example, the Thorlabs APT[54] motor implementation supports at least eight different daemons with each specifying only a handful of constants. These are extreme examples which are not representative of most hardware interfaces, so we omit them here.

Figure 4.3 shows the distribution of unique lines of Python code written to implement each of the daemons in our current ecosystem. While lines of code is an imperfect metric, we use it here to represent the amount of work required to create a daemon for a new hardware peripheral. Most interfaces, such as Brooks MFC[80], have been implemented in fewer than 100 lines of Python code. Even the most complicated daemons are implemented in about 300 lines. Implementing yaq daemons using Python is enabled through our own tooling mentioned above and the large and growing ecosystem of hardware interface tooling libraries that Python now provides[81, 82, 83, 84]. In our experience, the process of creating a new daemon involves about a day of work after communication with the hardware using Python is well understood. There are currently 70 daemons in the yaq project supporting at least 50 kinds of hardware, noting that some daemons support the same hardware and others are software only. Because yaq is standards based, anyone can design and publish new daemons extending our hardware support. A living list of all daemons and supported hardware can be found on the yaq website.[85]

The Stahl Group at UW-Madison created a custom reactor which monitors gasses being produced or consumed in the reaction head-space. [86] This reactor incorporates a collection of sensitive pressure transducers and a single heating process value under computer control. yaq daemons are used to interface with each sensor and the heater controller. Recently, experimentalists have been attempting reactions involving smaller, slower, pressure changes. A fundamental flaw in the initial analog to digital converter board was revealed by these attempts. As a result, a new digitizer has been purchased. This new digitizer will be incorporated into the existing reactor without modifying the existing graphical user interface and data recording program, minimizing downtime.

4.1.5 Technical Debt

Years after the original researchers leave, large monolithic acquisition programs become unknowable, undocumented, and unmaintained. A graduate student discovers a hard-coded conversion factor that is incorrect years after implementation. Scientists resort to sourcing a replacement for an old, broken oscilloscope due to their reliance of their software on that particular interface while newer, cheaper oscilloscopes are readily available. A graduate student is forced to meticulously reverse engineer the LabVIEW codebase that they inherited in order to understand the details of their experiment. Software

developers refer to the extra effort required to modify or fix large unmaintained codebases as "technical debt".[87] Technical debt grows especially fast in academic environments where graduate students are involved in projects for a limited time.

The yaq approach favors many small single-purpose applications above large monolithic ones. For daemons, the purpose of each application is obvious and unambiguous. There is a strict, well defined interface which explicitly limits the kinds of interactions that are provided to the hardware, thus limiting opportunity for unintended consequences. The lack of hardware interface code makes yaq clients much simpler and easier to describe and maintain. Tools like `yaqd-fakes`[88] allow clients to be tested and improved outside of their instrument, including the possibility of fully automated testing. Simple, script-based clients written using the expressiveness of Python can be read and understood in hours rather than weeks. Integrations with communities like Bluesky offer powerful features which are actively maintained across many institutions.

In yaq, each component of an instrument can be developed and distributed separately. For example, two different instruments might happen to use the same temperature sensor. Because the temperature sensor daemon is its own independent program, both instruments can benefit from the same daemon. The growing "ecosystem" of yaq daemons make future instruments easier and easier to develop. Growing this ecosystem is a collaborative effort where many yaq users create portable daemons that they need and share them with the community where they will be used and improved.

Software documentation is famously difficult and thankless work. yaq attempts to automate daemon documentation as much as possible. Our website, https://yaq.fyi, automatically builds generated reference pages for all known protocols. These pages are automatically updated when new versions are published.

yaq is open source software. Anyone can view, install, edit, and suggest changes to our growing collection of daemons and clients. Furthermore, anyone can create their own totally-custom client or daemon software separately by following the specified yaq standard. Thirteen individuals have contributed code to the development of yaq. Open-source development can be a powerful approach for research communities looking to share software-development and maintenance burdens. [89] It is our hope that a vibrant open-source community will form around yaq. While open-source development is not a panacea

[90], we hope that by maintaining a distributed development strategy with a strict focus on only hardware interfaces the yaq project might prove sustainable.

4.1.6 Conclusion

The yaq project defines a new general-purpose standard for hardware control in the context of scientific instrumentation. This standard has some of the powerful features of facility-scale standards while remaining simple enough for feasible implementation and maintenance in small research labs. We have shown how this approach alleviates common problems through discussion and case studies. Designing around self-describing protocols is a productive approach that has great promise in scientific software development.

4.2 Tools

4.2.1 `yaq-traits`

Installation

yaq-traits can be installed via PyPI[79] or conda-forge[91].

```
$ pip install yaq-traits
```
(4.1)

```
$ conda config --add channels conda-forge
$ conda install yaq-traits
```
(4.2)

definitions

Avro Protocol (AVPR)[92] The Avro protocol is the fully specified description of the daemon. It describes the exposed method signatures (names, arguments, defaults to arguments, text description). In addition, yaq avpr files include information about the configuration, state, and traits of a daemon. Avpr is a JSON file.

TOML[93] TOML is a configuration file. Here, we use TOML files to specify the minimum info about a daemon. (i.e. behavior and descriptions inherited via traits is *not* included in the TOML) A full reference is available below.

Trait[94] A trait is simply a collection of exposed methods, configuration, and state used to encourage shared behavior among similar yaq daemons with different implementations

Usage

yaq-traits is a command line application.

Help Help: learn more, right from your terminal.

```
$ yaq-traits --help
Usage: yaq-traits [OPTIONS] COMMAND [ARGS]...

Options:
  --version  Show the version and exit.
  --help     Show this message and exit.

Commands:
  check
  compose
```
(4.3)

Try `yaq-traits --help` to learn more about a particular command.

list List: list available traits

```
$ yaq-traits list
has-limits
has-measure-trigger
has-position
has-turret
is-daemon
is-discrete
is-homeable
is-sensor
uses-i2c
uses-serial
uses-uart
```
(4.4)

compose Compose: Convert a simplified TOML file to a fully specified AVPR. This takes a path to a TOML file as an argument, and prints the AVPR to standard out.

```
$ yaq-traits compose my-daemon.toml > my-daemon.avpr
```
(4.5)

get Get: Retrieve a fully specified AVPR for a trait (similar to compose, but not a full daemon). This takes a name of a trait as an argument, and prints the AVPR to standard out.

```
$ yaq-traits get has-limits > has-limits.avpr
```
(4.6)

check Check: Verify that an AVPR file matches current trait behavior. This takes a path to an AVPR file as an argument, and prints a table of traits and whether the trait was found to be accurate. If it fails, the traits which are not specified are explicitly printed and a nonzero exit status is returned. The primary intent of `check` is to be used by Continuous Integration tests, which consistently get the latest version and will alert you if there is a change to the traits.

```
$ yaq-traits check fake-continuous-hardware.avpr
+---------------------+----------+----------+
| trait               | expected | measured |
+---------------------+----------+----------+
| has-limits          | true     | true     |
| has-position        | true     | true     |
| has-turret          | false    | false    |
| is-daemon           | true     | false    |
| is-discrete         | false    | false    |
| is-homeable         | false    | false    |
| uses-i2c            | false    | false    |
| uses-serial         | false    | false    |
| uses-uart           | false    | false    |
| has-measure-trigger | false    | false    |
| is-sensor           | false    | false    |
+---------------------+----------+----------+
Error: failed to verify expected trait(s):
  is-daemon
```
(4.7)

What to do when a check fails:

- check that you have the most up to date `yaq-traits`
- check the changelog[95] for `yaq-traits` to see what has been changed recently
- In many cases, the accompanying behavior change will be implemented in the `core` package for your language (e.g. Python[96]). If so, all you need to do is pin the version of core and rerun `yaq-traits compose`
- if the change requires your attention, make the necessary code changes and then rerun `yaq-traits compose`
- feel free to reach out to the yaq developers[97] or raise an issue[98] if you are unsure of what is needed

yaq TOML file specification

In general, the TOML file is used to specify the minimal description of a daemon. Inherited behavior from traits are not included. You may assign or reassign the default value of the default, but should not change the documentation, type, or arguments to a message. The same format is used to specify traits within ‘yaq-traits‘.

frontmatter These identifying information about the daemon are provided as top level keys.

protocol (string) The name of the daemon. The equivalent for a trait definition is **trait**.

doc (string) A description of the daemon, rendered at the top of its documentation page.

traits ({'items': 'string', 'type': 'array'}) List of traits implemented by the daemon. In trait definitions, the analogous entry is **requires**, which lists traits that are implied by using that trait. If a traits is implied by other traits (e.g. `has-limits` requires `has-position`) the required trait need not be specified. `is-daemon` must be specified by all daemons.

hardware ({'items': 'string', 'type': 'array'}) A list of supported hardware. The entries are formatted as 'make:model'. Supported hardware should also appear on the `yaq` website [99]. Used only for building documentation.

links The links are a map of arbitrary keys to URLs. In general links to documentation, source, and a bugtracker are good to have. Additionally links to the manufacturer/library used are common.

```toml
[links]  # all optional, arbitrary keys supported
documentation = "https://yaq.fyi/daemons/example-daemon"
source = "https://git.example.com/example-daemon"
bugtracker = "https://git.example.com/example-daemon/-/issues"
```
(4.8)

Installation Installation is just like links, except with the specific goal of pointing to installable package references. For python, this likely includes a link to PyPI and/or conda-forge.

```
[installation]  # all optional, arbitrary keys supported
PyPI = "https://pypi.org/project/yaqd-example"
conda-forge = "https://anaconda.org/conda-forge/yaqd-example"
```
(4.9)

types All valid Avro types[100] are valid for yaq.

In addition, `yaq-traits` provides a definition for an N-dimensional homogeneous array, which can be used by putting the string `"ndarray"` as the type.

Unions of types are specified using TOML arrays.

Since TOML does not include a null value (and null is a valid default value, distinct from not having a default) the string `"__null__"` is used in place of the null literal. Note that when referring to the *type*, `"null"` is used, just as `"int"` is used to specify the integer type.

Named types (e.g. records and enums) may be defined inline where they are used, but may also be provided as an array of tables:

```
[[types]]
name="Greeting"
type="record"
[[types.fields]]
name="message"
type="string"

[[types]]
name="Curse"
type="record"
# You may find inline tables/arrays more readable here
# Note that toml forbids multi-line inline tables (arrays can be multi-line)
fields = [{name="message", "type"="string"}]
```
(4.10)

config Each config parameter is a table with the name of the parameter as the key within the `[config]` table with the following keys:

type (required) Avro type definition, commonly a string such as `"int"` or `"ndarray"`, but may be a table representing collection types or a record or an array representing a union of types.

doc (optional) A string description of the config parameter

default (optional) The default value if the config parameter is omitted in the daemon configuration. If no default is given, it is considered required.

A daemon may override the default value defined by a trait (or provide one where none was given). Additionally, rather then overwriting the doc, a key **addendum** may be added to provide additional context to the variable (e.g. communicating that a parameter is required despite a null default value being defined in the trait).

```
[config]

[config.a_binary_config]
type = "boolean"
default = false
doc = "An example of a boolean config param"

[config.an_optional_array]
type = ["null", {type = "array", items = "int"}]
default = "__null__"

# Overriding a config from a trait
[config.baud_rate]
default = 57600
addendum = "I want to provide more info about why 57600 is the default"
```
(4.11)

state The state is configured exactly the same as config, except that *all* state variable *must* have a default value. The same rules about overriding defaults and addenda apply to state variables.

```
[state]

[state.a_binary_state]
type = "boolean"
default = true
doc = "An example of a boolean state variable"
```
(4.12)

messages Messages are the exposed remote procedure calls over the TCP interface. This follows closely the specification provided by Avro[101]. Note that all errors in the Python implementation are

treated as strings, though that may change in the future.

If the request field is omitted, the default is `[]`, i.e. no parameters.

If the response field is omitted, the default is `"null"`.

Messages defined by traits should be omitted, only messages unique to the daemon should be included.

Messages should not be overridden, as the behavior being the same to the client program is the intent of the traits system.

```
[messages.set_int]
request = [{"name"="int_value", "type"="int"}]
doc = "Set an example int value."

[messages.get_int] response = "int"
doc = "Get the example int value."
```
(4.13)

yaq AVPR file specification

The AVPR files generated are compliant with the Avro specification[92], however `yaq-traits` does add additional information such that the avpr alone can be used to render the documentation.

The AVPR files include the complete list of `config` and `state` (including those inherited from traits).

The links (including installation links) are also included. Additional keys specifying which trait provided a method or config/state variable are also added by `yaq-traits`

4.2.2 `yaqd-control`

installation

yaqd-control can be installed via PyPI[102] or conda-forge[103].

```
$ pip install yaqd-control
```
(4.14)

```
$ conda config --add channels conda-forge
$ conda install yaqd-control
```
(4.15)

Usage

yaqd-control is a command line application.

Help: learn more, right from your terminal.

```
$ yaqd --help
Usage: yaqd [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  clear-cache
  disable
  edit-config
  enable
  list
  reload
  restart
  scan
  start
  status
  stop
```
(4.16)

Try `yaqd --help` to learn more about a particular command.

the cache yaqd-control keeps track of known daemons, referred to as the cache

Status: yaqd-control can quickly show you the status of all daemons in yaqd-control's cache. This is usually the most used subcommand, as it gives a quick overview of the system, which daemons are offline, and which are currently busy.

```
$ yaqd status
+-----------+-------+--------------------------+------+---------+-------+
| host      | port  | kind                     | name | status  | busy  |
+-----------+-------+--------------------------+------+---------+-------+
| 127.0.0.1 | 38202 | system-monitor           | foo  | online  | False |
| 127.0.0.1 | 39054 | fake-continuous-hardware | bar  | online  | True  |
| 127.0.0.1 | 39055 | fake-continuous-hardware | baz  | online  | False |
| 127.0.0.1 | 39056 | fake-continuous-hardware | spam | offline | ?     |
| 127.0.0.1 | 37067 | fake-discrete-hardware   | ham  | online  | False |
| 127.0.0.1 | 37066 | fake-discrete-hardware   | eggs | online  | False |
+-----------+-------+--------------------------+------+---------+-------+
```
(4.17)

List: this is essentially the same as `status` except that it does not attempt to contact the daemons, so it does not give you additional context. List supports a flag --format which accepts "json" or "toml".

```
$ yaqd list
+-----------+-------+--------------------------+------+
| host      | port  | kind                     | name |
+-----------+-------+--------------------------+------+
| 127.0.0.1 | 38202 | system-monitor           | foo  |
| 127.0.0.1 | 39054 | fake-continuous-hardware | bar  |
| 127.0.0.1 | 39055 | fake-continuous-hardware | baz  |
| 127.0.0.1 | 39056 | fake-continuous-hardware | spam |
| 127.0.0.1 | 37067 | fake-discrete-hardware   | ham  |
| 127.0.0.1 | 37066 | fake-discrete-hardware   | eggs |
+-----------+-------+--------------------------+------+
```
(4.18)

Scan: Scanning allows you to add currently running daemons to the cache.

```
$ yaqd scan
scanning host 127.0.0.1 from 36000 to 39999...
...saw unchanged daemon fake-discrete-hardware:eggs on port 37066
...saw unchanged daemon fake-discrete-hardware:ham on port 37067
...found new daemon system-monitor:foo on port 38202
...found new daemon fake-continuous-hardware:bar on port 39054
...saw unchanged daemon fake-continuous-hardware:baz on port 39055
...known daemon fake-continuous-hardware:spam on port 39056 not responding
...done!
```
(4.19)

Scan has some additional options, passed as flags on the command line, which allow you to change the default scan range and host (for remotely accessed daemons):

```
$ yaqd scan --help
Usage: yaqd scan [OPTIONS]

Options:
  --host TEXT      Host to scan.
  --start INTEGER  Scan starting point.
  --stop INTEGER   Scan stopping point.
  --help           Show this message and exit.
```
(4.20)

Edit Config: yaqd-control provides an easy way to edit the default config file location for a daemon kind. This uses your default editor (EDITOR environment variable), and defaults to `notepad.exe` on Windows, and `vi` on other platforms. Using yaqd-control to edit config files means that you do not need to know the default location. Additionally, it does some basic validity checks (that the toml parses and that each daemon section has the `port` keyword). If an error is found, you are prompted to re-edit the file. Daemons from the config file are added to the cache. You may pass multiple daemon kinds, which will be opened in succession.

```
$ yaqd edit-config fake-continuous-hardware system-monitor
```
(4.21)

Clear Cache: Note that this is a destructive action. `clear-cache` deletes all daemons from the cache (thus `list` and `status` will give empty tables) There is no user feedback.

```
$ yaqd clear-cache
$ yaqd status
+------+------+------+------+--------+------+
| host | port | kind | name | status | busy |
+------+------+------+------+--------+------+
+------+------+------+------+--------+------+
```
(4.22)

Running in the background Each of the commands in this section can take multiple daemon kinds.

Enable: by enabling a daemon, you allow the operating system to manage that daemon in the background. An enabled daemon will always start again when you restart your computer. Enabling is required for the rest of the commands in this section to work as expected. After enabling, it's typical to start the daemon as well, this does not happen automatically. Enablement works in slightly different ways on different platforms, but the commands are the same (don't worry if the password prompts are differ-

ent). Currently supported platforms are Linux (systemd), MacOS (launchd) and Windows (via NSSM, bundled with the distribution).

```
$ yaqd enable system-monitor
[sudo] password for scipy2020:
==== AUTHENTICATING FOR org.freedesktop.systemd1.manage-unit-files ===
Authentication is required to manage system service or unit files.
Password:
==== AUTHENTICATION COMPLETE ===
```
(4.23)

Disable: this is the inverse operation to enable, which makes it so that the daemon does not start on reboot. This does not affect the running daemon.

```
$ yaqd disable system-monitor
==== AUTHENTICATING FOR org.freedesktop.systemd1.manage-unit-files ===
Authentication is required to manage system service or unit files.
Password:
==== AUTHENTICATION COMPLETE ===
Removed /etc/systemd/system/multi-user.target.wants/yaqd-system-monitor.service.
```
(4.24)

Start: This starts the daemon running in the background immediately. It must have been enabled to run in the background using this command.

```
$ yaqd start system-monitor
==== AUTHENTICATING FOR org.freedesktop.systemd1.manage-units ===
Authentication is required to start 'yaqd-system-monitor.service'.
Password:
==== AUTHENTICATION COMPLETE ===
```
(4.25)

Stop: This stops the daemon running in the background immediately. It must have been running in the background using yaqd-control (either on startup via enable or via the start command above).

```
$ yaqd stop system-monitor
==== AUTHENTICATING FOR org.freedesktop.systemd1.manage-units ===
Authentication is required to stop 'yaqd-system-monitor.service'.
Password:
==== AUTHENTICATION COMPLETE ===
```
(4.26)

Restart/Reload: This stops (if running) and restarts the daemon running in the background immediately.

Reload is slightly different in that it signals to the daemon to reload its configuration rather than completely restart, but effectively it is the same as restart (and is a pure alias where such a signal is not supported). It must have been enabled to run in the background using this command.

```
$ yaqd restart system-monitor
==== AUTHENTICATING FOR org.freedesktop.systemd1.manage-units ===
Authentication is required to restart 'yaqd-system-monitor.service'.
Password:
==== AUTHENTICATION COMPLETE ===
```
(4.27)

4.2.3 yaqc-qtpy

yaqc-qtpy is a graphical application which provides easy access to most features provided by yaq, especially those defined by traits and properties. It is designed to be generically useful to all yaq users, and does not attempt to specialize for particular use cases or perform data acquisition tasks. Rather, it is designed to be useful for "engineering" tasks such as alignment or initially acquiring signal. Its name comes because it is implemented using yaqc[69], the generic Python yaq client and QtPy[104], an abstraction layer providing a unified interface to several Python bindings for Qt[105], a graphical application toolkit.

Installation

yaqc-qtpy is included with the installation of the Python package of the same name. As such it is installed via:

```
conda install yaqc-qtpy
```
(4.28)

or:

```
pip install yaqc-qtpy
```
(4.29)

In the future, the application may be packaged so as to be installable as standard executable programs

with included Python and dependencies.

Usage

Starting `yaqc-qtpy` is done by running the following command:

```
yaqc-qtpy
```
(4.30)

In the future, there is likely to be the option to start the application from standard application menus.

When the application opens, there is a list of available daemons along the left sidebar and a picture of a yak in the main window, as shown in Figure 4.4. Daemons are shown in alphabetical order. Each daemon has a checkbox which indicates the current state of "busy" for that daemon. Offline daemons have a simple text box which shows that the daemon is offline. Each of the daemon sections can be expanded, providing additional options for controlling the hardware.

Figure 4.4: Image of `yaqc-qtpy` upon initial opening.

When expanded, each daemon provides a "card" which provides information and controls for that daemon. At the top of the card, the host and port that are used to communicate with the daemon are shown. Additionally, all properties (whether provided by a trait or unique to the daemon) that have "control_kind" set to "hinted" are shown in this sidebar. This provides immediate access to the most important information and controls for that daemon. Values with units can be set in compatible units, and values with enumerated options are provided as a drop down menu. Each daemon has a button to open the advanced menu for that daemon. The advanced menu appears in the main portion of the window (where the yak picture appears at initial start-up) and has trait-based user interfaces. Figure 4.5 shows an example of the left sidebar with some sections expanded.

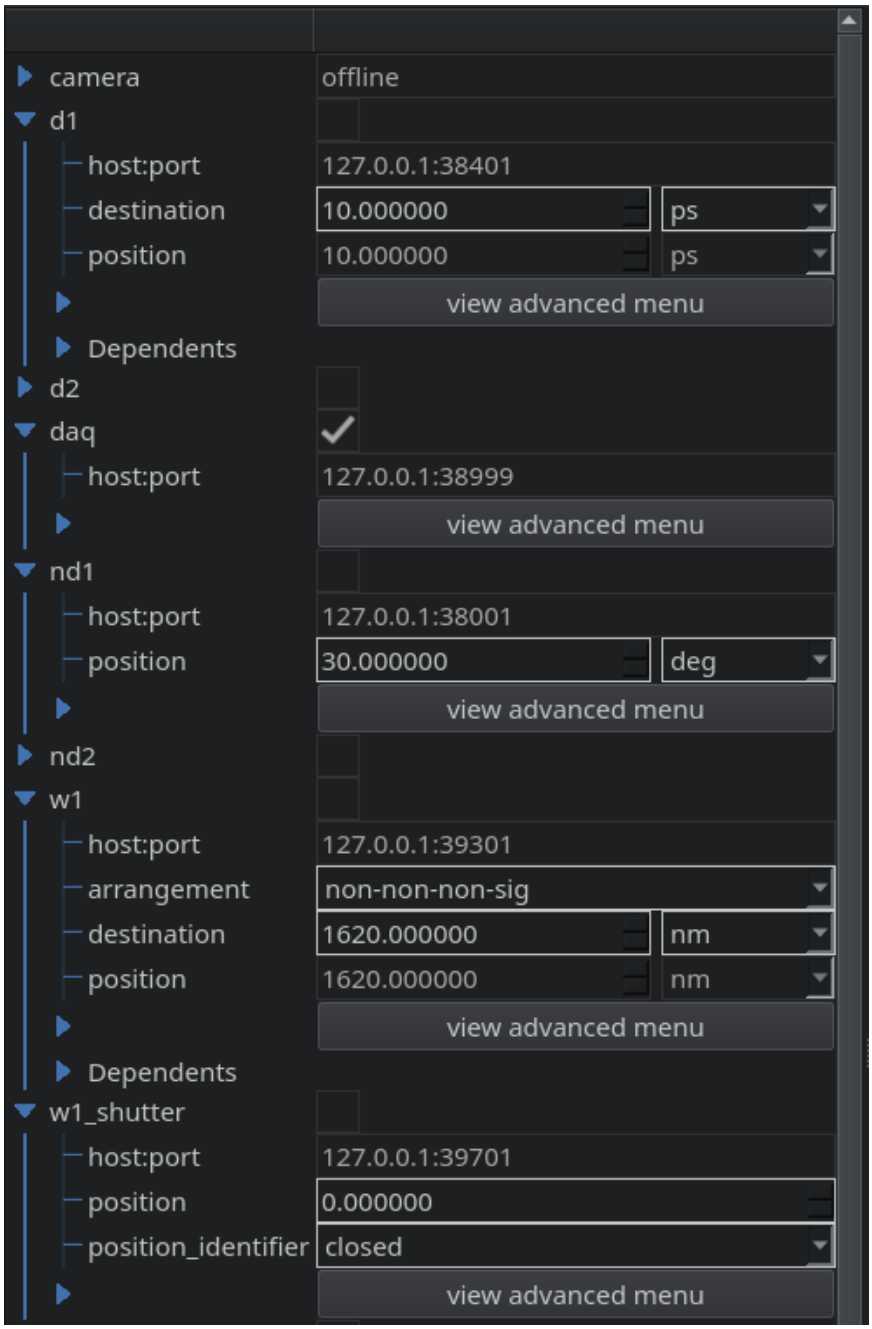

Figure 4.5: The left sidebar of `yaqc-qtpy` which provides collapsible sections for each known daemon. The busy state is represented by the checkbox on the line with the daemon name Properties with "control_kind" of "hinted" are shown directly in this sidebar. An advanced window for each daemon can be accessed by pressing the button below the properties.

Hiding

Because the number of daemons installed on a system is often larger the number of daemons that regularly need to be interacted with in engineering contexts, `yaqc-qtpy` implements a system of "hiding" those daemons that are not desired. Daemons which are hidden are still accessible in the application, but show up in a collapsed section below all of the non-hidden daemons. To hide (or unhide) a daemon, a button is provided as a collapsed subitem of the "view advanced menu" for that item. When clicked, the card for that daemon will move to the other section, in appropriate alphabetical order. Figure 4.6 shows the Hide and Unhide buttons for two daemons, along with the Hidden collapsable section which holds all of the hidden daemons. Daemons which are hidden are stored in a configuration file by their host:port string, and remain hidden through restarting the application.

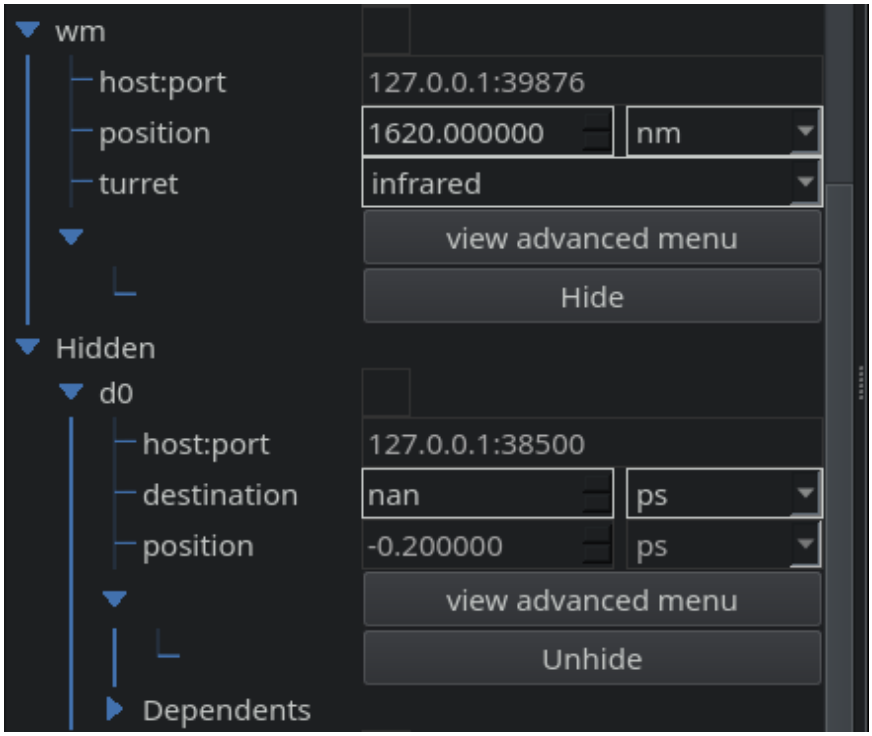

Figure 4.6: Under the collapsed menu for the advanced menu, a button to Hide (or Unhide) the daemon is provided. Hidden daemons appear at the bottom of the list of daemons, under a collapsed heading labeled "Hidden".

Dependent Daemons

Daemons which implement the "has-dependents" trait have additional daemons that are associated with that hardware. These are shown as a collapsable section labeled "Dependents". In this section, the cards for each of the sub daemons are shown. It is common (though by no means guaranteed) that the dependent daemons are also included in the top level list. Thus, it is also common to then hide those daemons as they are rarely interacted with directly, and even when they are it is usually in the context of their parent daemon. Figure 4.7 shows the dependent daemons for an Attune daemon.

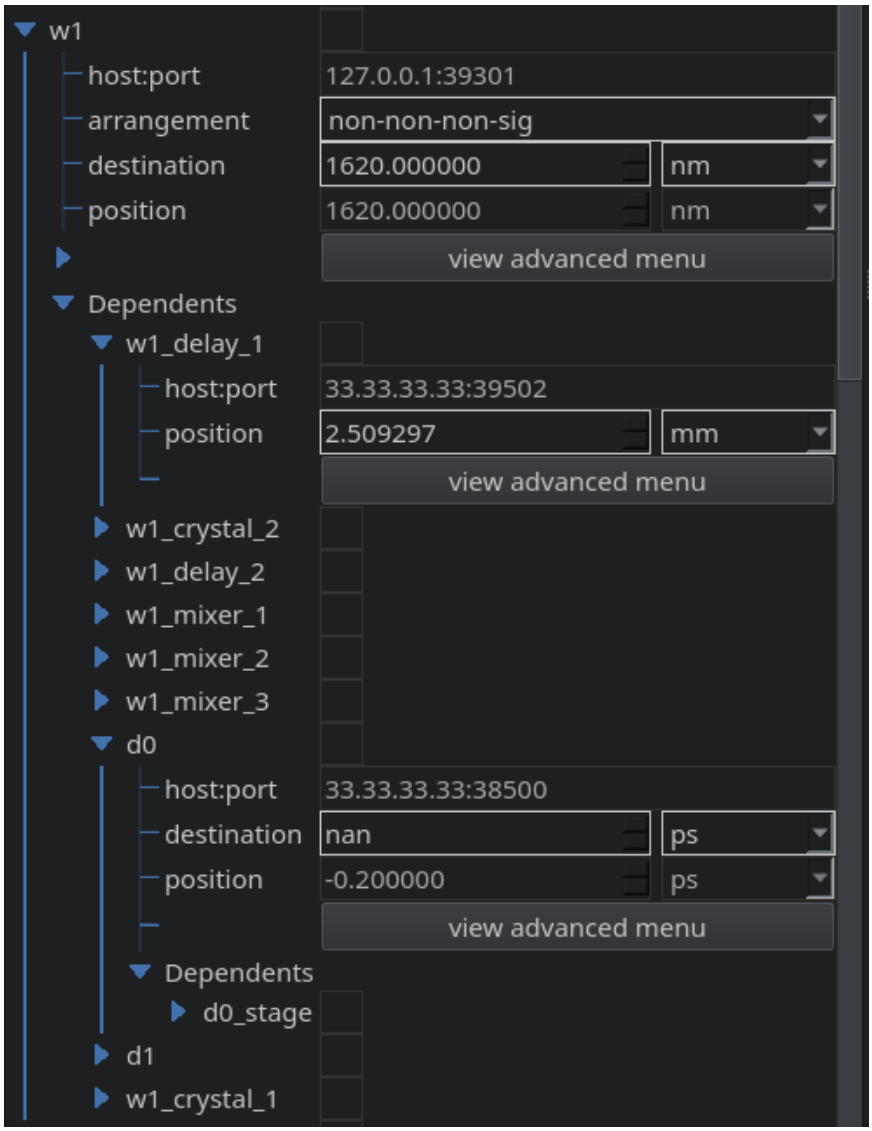

Figure 4.7: Daemons which implement the "has-dependents" trait have a collapsed section with the cards for each dependent.

Configuration Tab

Each daemon has an advanced menu which consists of some number of tabs. The first few tabs appear for all daemons and are fairly generic as a result. Next are tabs for particular traits, such as "has-position" and "is-sensor". Finally, individual daemons can provide plugins for `yaqc-qtpy` which provide additional specialized intefaces for that device. The default tab when opening is the last tab, which in general is the most specific interface available.

The first tab in the list, config, displays a read-only view of the full configuration of the daemon, displayed in TOML. This is directly the response from calling the `get_config` message for the daemon. Figure 4.8 shows this panel.

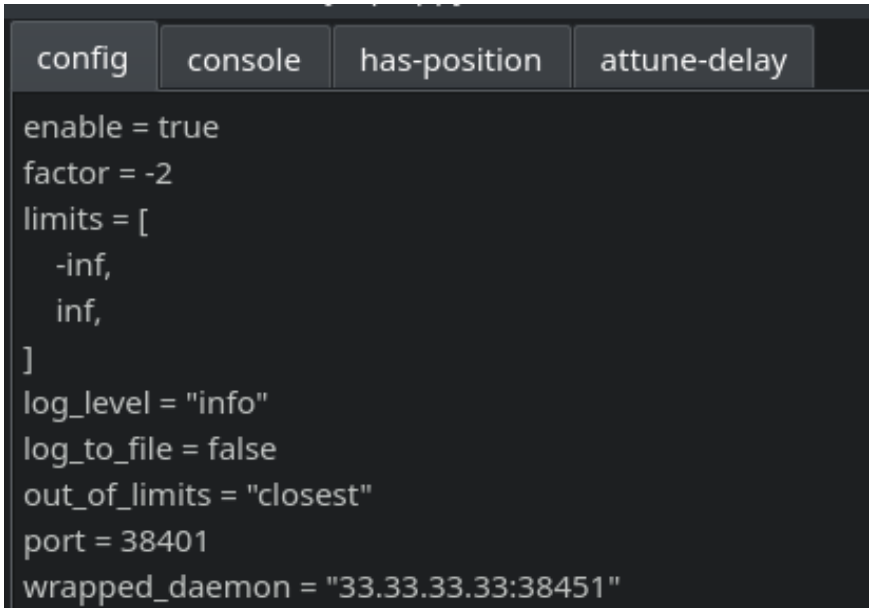

Figure 4.8: Each daemon's advanced menu includes the full configuration, displayed as TOML. This configuration is not editable, it is provided for information only.

Python Console Tab

The second tab, also available for each daemon, provides an IPython[106] console so that all capabilities of the daemon are available, including running arbitrary Python code to compute inputs. The console automatically runs a cell which creates a client using `yaqc`. Two names for the daemon are provided, the actual name of the daemon, and the one character variable name `c`, which stands for "client". This Python console, including the initial cell that is run with parameters appropriate for the individual daemon, is shown in Figure 4.9.

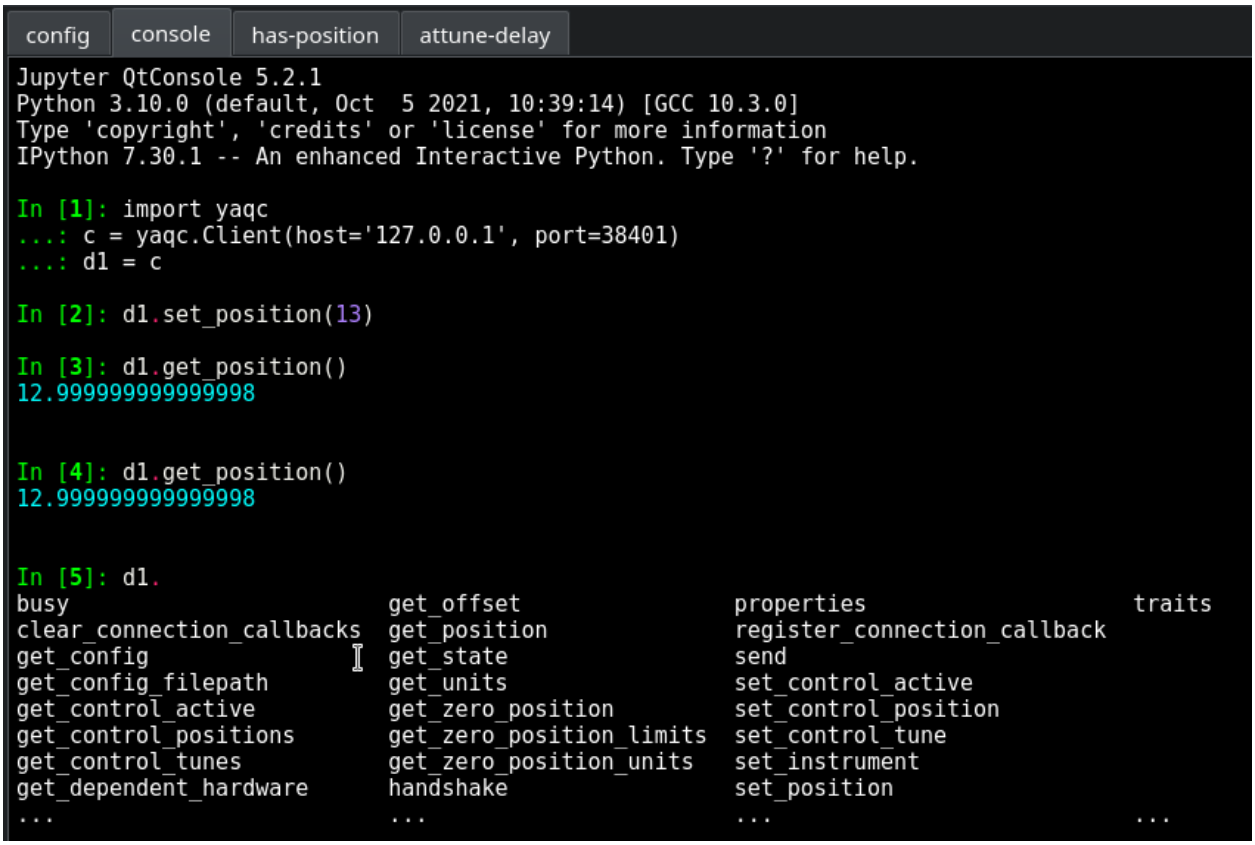

Figure 4.9: The advanced menu includes a Python console with the `yaqc` client for the daemon preloaded.

Has Position Tab

The "has-position" tab provides useful interactions for all hardware which implement the trait of the same name. Figure 4.10 shows an example of this tab for a daemon called "d1". At the top, the name and most recent reading are displayed in large font such that it can be viewed from accross the room. Underneath, a graph showing the last minute of recorded positions for that daemon is shown. Limits and the current setpoint are shown as horizontal lines, in red and green colors, respectively. While some daemons are unable to report intermediate positions, if available these will be shown for position of the daemon while in motion.

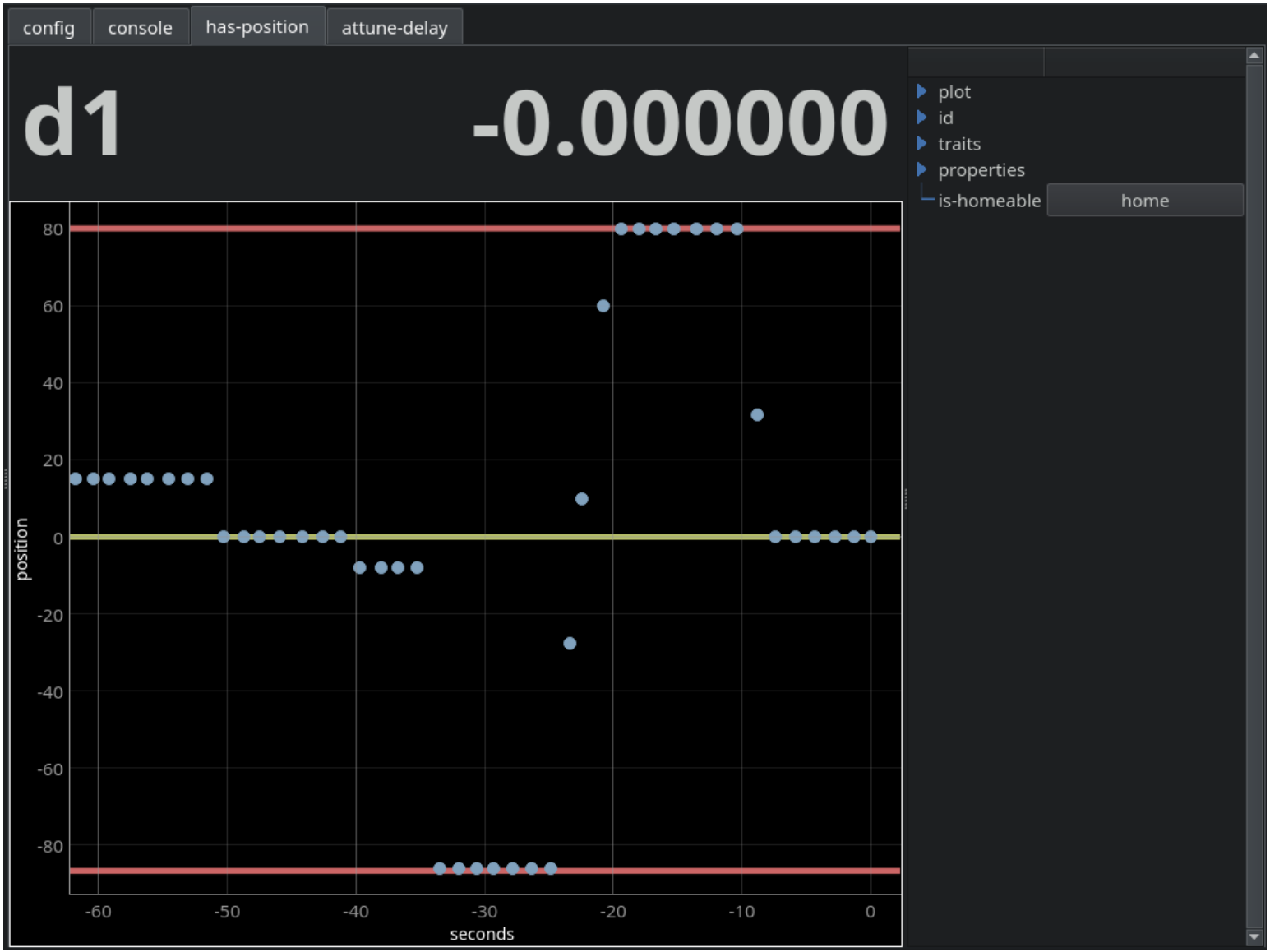

Figure 4.10: Graph produced for daemons which implement the "has-position" trait. Plots the position versus time, with the current set point shown as a horizontal green line. Limits are shown as horizontal red lines.

The right sidebar provides additional controls to interact with the daemon. At the top, a collapsable section allows control over the plot as shown. In particular, it allows locking of the y axis limits to focus on a particular range of the axis. This section also displays what the limits that are actively being used are. Figure 4.11 shows the plot controls tree.

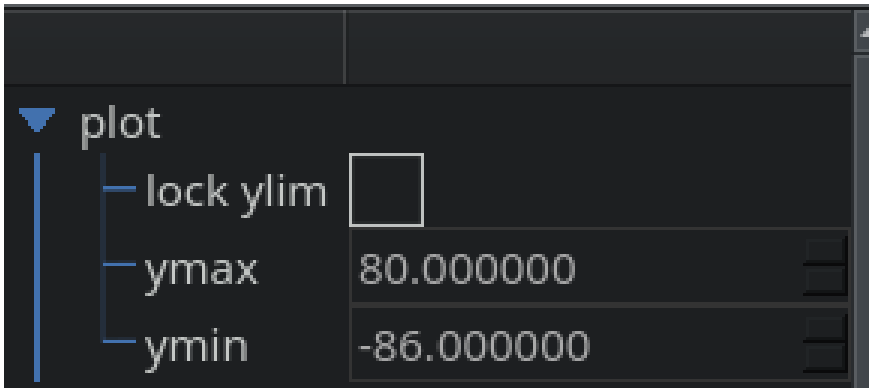

Figure 4.11: The controls and information about the plot for “has-position” daemons.

Below the plot controls are two sections which give information about the daemon, "id" and "traits", as shown in Figure 4.12. The "id" section provides information such as the name, make, model, and serial number as provided by `client.id()`. This information can be helpful for correlating the daemon to the physical hardware. Since the displayed daemon is a pure software daemon, the make, model and serial number fields are all blank. Below "id" is an array of trait names, with check boxes which are checked for the traits implemented by the daemon. This provides quick answers for whether or not you can use features from a given trait with that daemon.

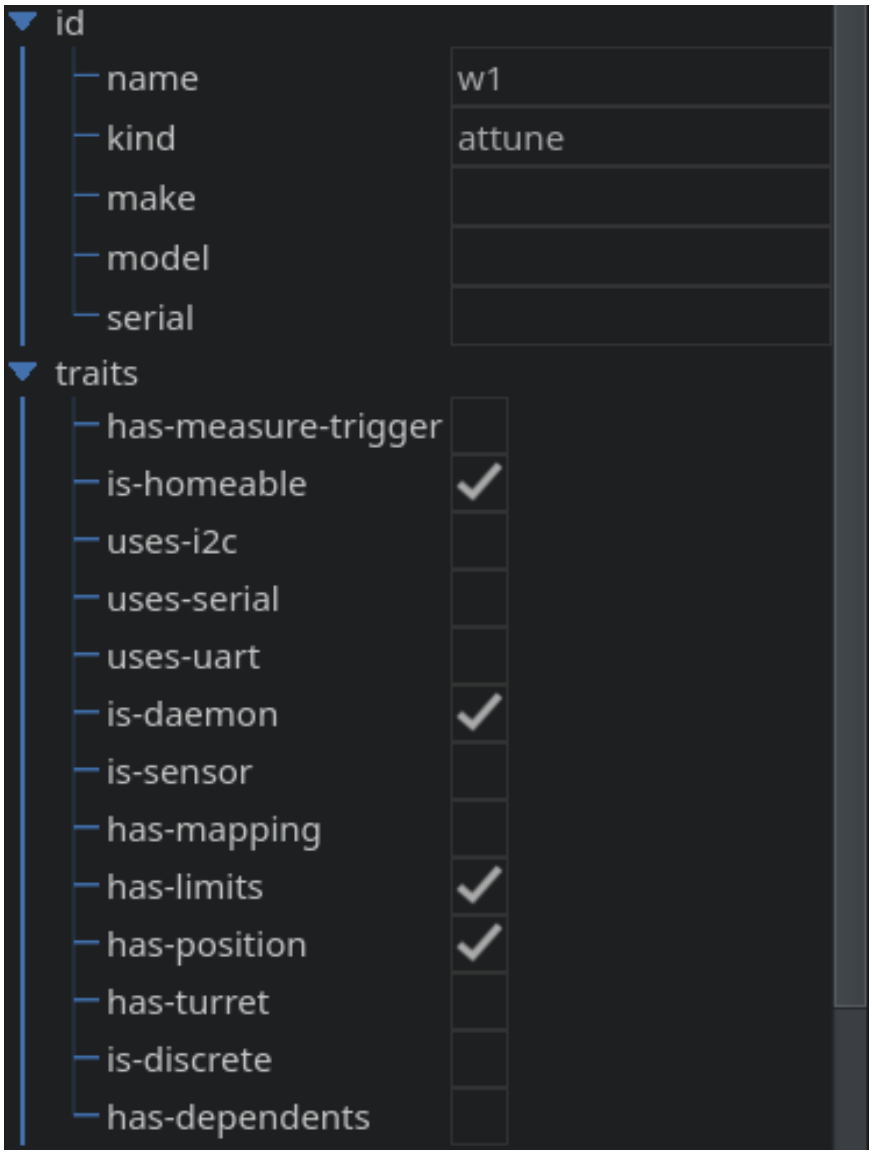

Figure 4.12: Information about the daemon, including the information from the "id" message and implemented traits.

Below the information sections is a section for "properties", as shown in Figure 4.13. This provides UI elements for all properties of "hinted" or "normal" control kind as specified by the properties entry in the AVPR. These properties may be added by traits or by individual daemons. Properties with a setter are editable fields which when edited will call the set message. All properties are polled for their current value. Properties with units can be set in alternative units from the user interface, the conversion is performed by the client and the native value is actually provided to the daemon. Any property that has type of boolean will be displayed as a check box. Strings, integers, and floating point numbers are shown as text boxes, with floating point numbers additionally having a units dropdown. Enumerated types, either using an Avro Enum type or by virtue of having an "options_getter" specified by the associated property, are displayed as a dropdown combo box. If the property has a type other than those specified, it is ignored, as there are no interface elements associated with other types. If a user wishes to interact with such properties, a plug-in module must be provided that includes interface to set and display the property with an alternate type.

Below the "properties", if the "is-homeable" trait is implemented, then a button is shown which calls `client.home()` when pressed.

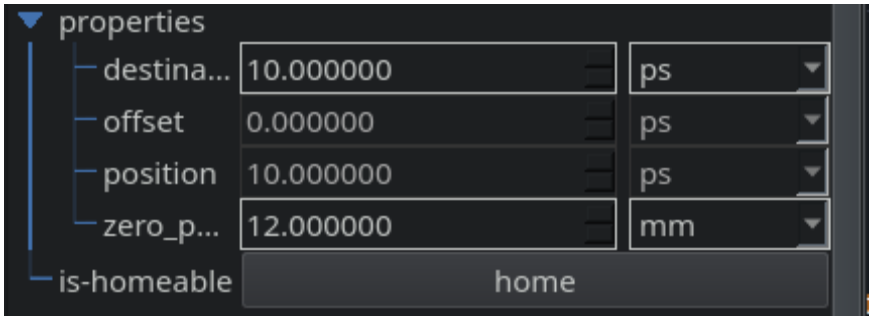

Figure 4.13: All of the properties that have "control_kind" of "hinted" or "normal" show up in the advanced sidebar. Additionally, daemons which implement "is-homeable" have a "home" button to initiate the homing procedure.

Is Sensor Tab

The "is-sensor" tab is designed to be informative for interacting with daemons that implement the trait of the same name. Its primary display is shown in Figure 4.14. At the top, much like the "has-position" tab, a name and most recent value are shown. In this case, however, it is the name of the channel, rather than of the daemon itself, as sensors can have multiple channels. Below the large number is a graph showing the most recent series of measurements. At this time, the tab is only useful for channels which are scalar values, such that the plot shows a plot of value over time. In the future, shaped sensors such as array detectors should show the appropriate measurement against their mapping or index values rather than only plotting versus time. Additionally, two dimensional detectors such as cameras should show their images rather than a lower dimensional slice.

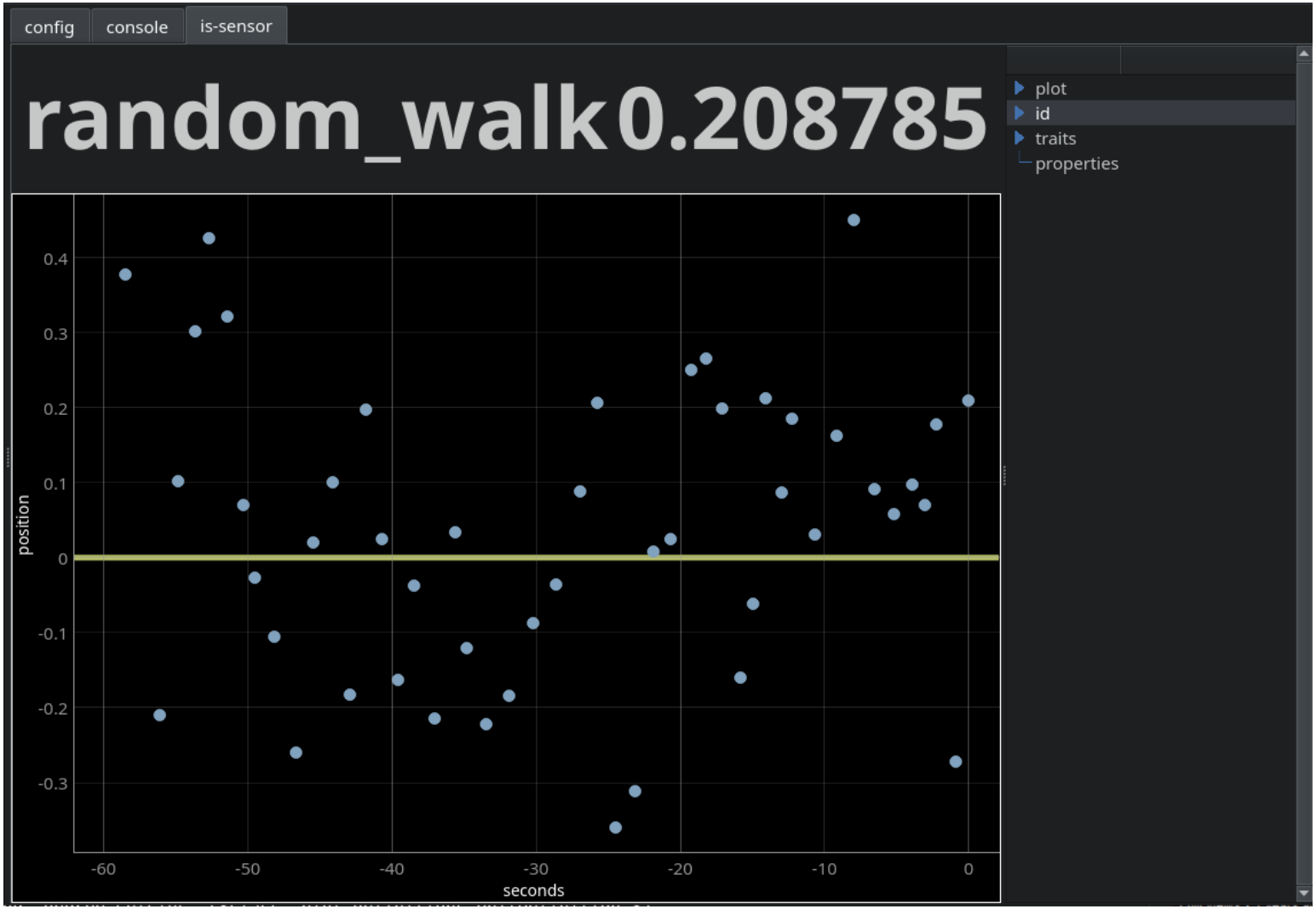

Figure 4.14: Graph produced for daemons which implement the "is-sensor" trait. Plots the channel versus time, with the zero shown as a horizontal green line. The most recent value and the channel name is shown at the top.

Similar to “has-position”, the right sidebar has controls for the plot, as shown in Figure 4.15. These controls are more extensive than the controls provided for “has-position”, providing separate controls for the x axis and y axis. The x axis controls allow control over the number of points that are kept in memory, along with the rate to poll for new readings, which defaults to half a second. Finally, it allows control over the range displayed on the x axis, defaulting to -60, or one minute in the past. The y axis controls allow for changing the displayed channel, and adjusting the limits. By default if the limits are not locked, then the ymax and ymin values will expand the range to show new data points, but never contract, even if the points that caused them to expand are no longer kept in memory. Pressing the button for resetting the y limits causes the limits to contract to the range currently held in memory, ignoring past values which may be larger or smaller.

In addition to the plot controls, the right panel contains much of the same options as provided by the “has-position” tab: “id”, “traits” and “properties”. Many sensors have no properties, but this is where options such as integration time would be provided.

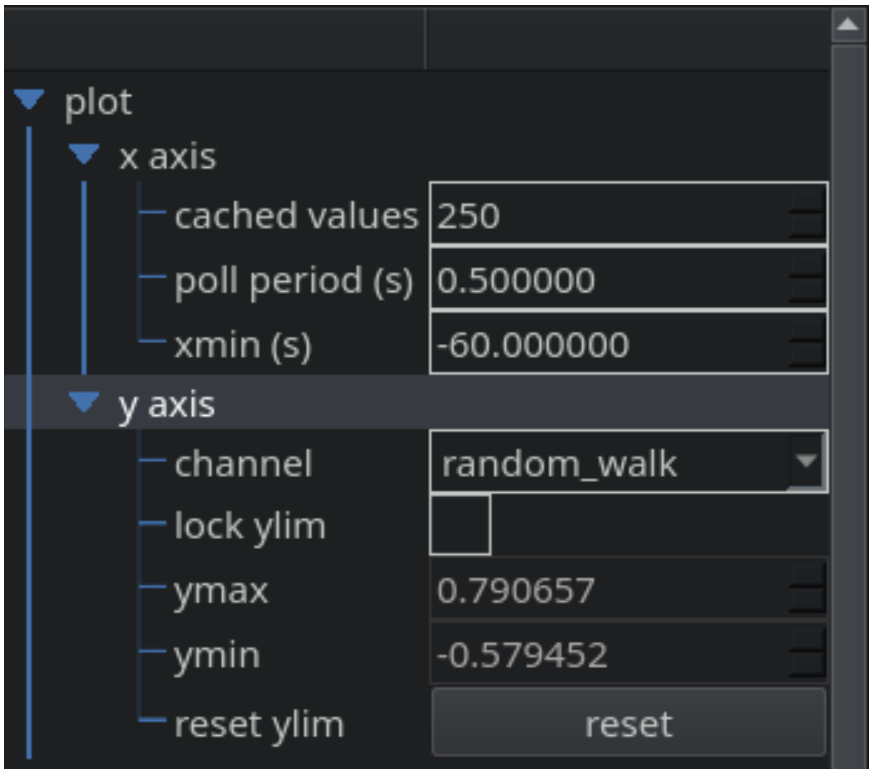

Figure 4.15: The plot controls for "is-sensor" graph, found in the right sidebar. The number of points, poll period, and axis limits are controlled for the x axis. The displayed channel can be changed using the y axis controls.

Specialized Plugins

While traits allow for useful and consistent interfaces, some hardware has additional functionality that benefits from more specialized user interface. These plugins use Python's system of "entrypoints", which provide named and grouped references to a Python class or function. For `yaqc-qtpy` plugins, this should be a reference to a Callable object which accepts an instance of `QClient` (the Qt wrapper around `yaqc`) and returns a QWidget that can be displayed in the main area of `yaqc-qtpy`. Most often this is actually a class which inherits from `QWidget`, but that is not a strict requirement. The entrypoint is given the group of `"yaqc_qtpy.main.<daemon kind>"` and the name of the individual entry point is the title that will appear as the tab.

attune The Attune daemon in particular has one such plugin for displaying the active Attune Instrument object for the daemon, shown in Figure 4.16. The right sidebar has many of the same fields as the has-position and is-sensor tabs: id, traits, properties, and a button since the daemon implements is-homeable. Additionally there is a collapsible information section in the sidebar which gives the names and ranges for each available Arrangement in the Instrument object. If you expand the arrangement range heading, a list of Tunes in that arrangement is shown.

The main portion of the widget is a graph which displays one tune, with setpoint on the x axis and motor position on the y axis. The current setpoint is shown as a vertical yellow line. Only tunes from the current arrangement can be graphed, and it is controlled using the top section from the right sidebar. Additionally, the x axis units can be selected.

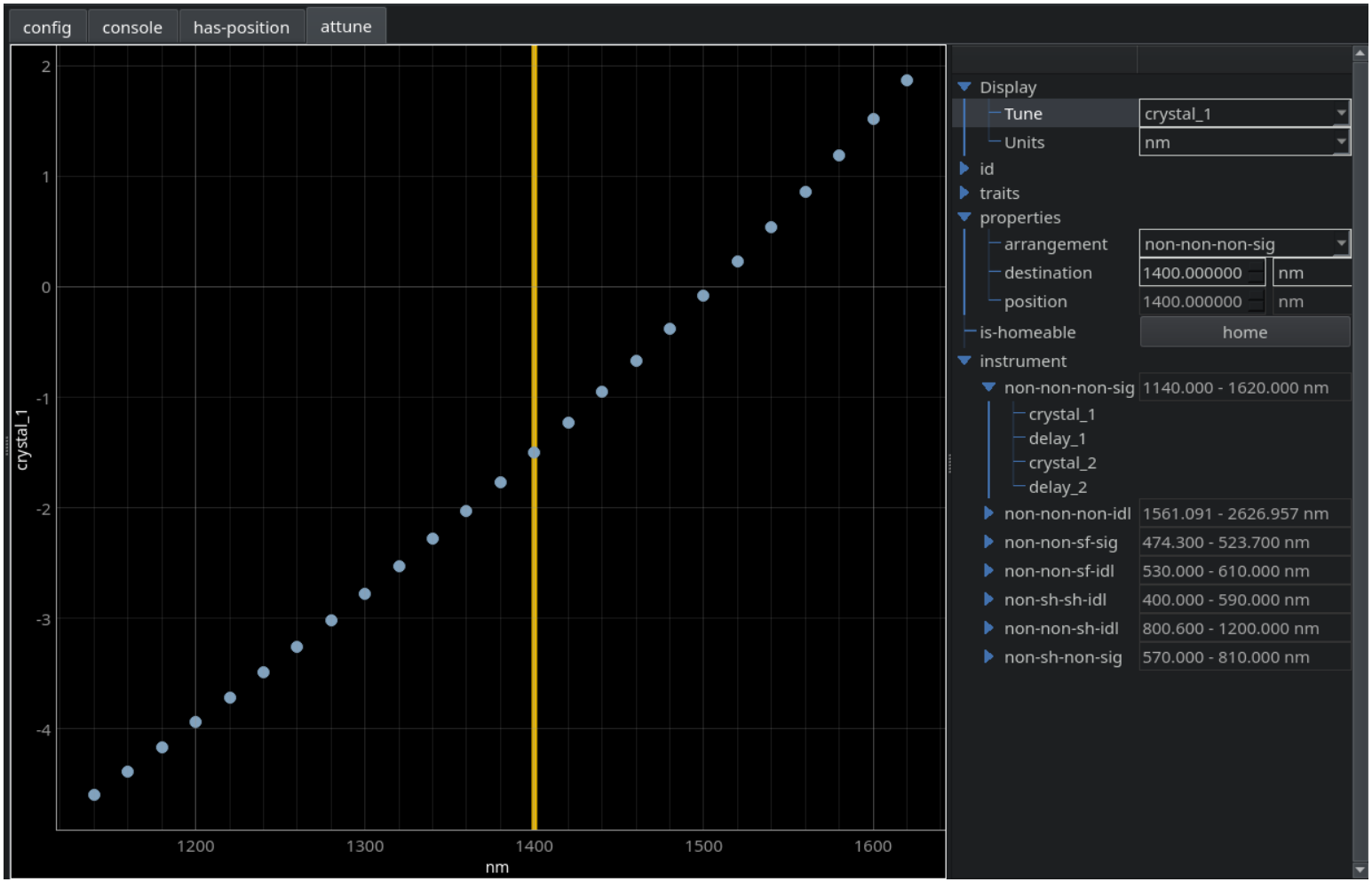

Figure 4.16: The attune daemon has a plugin which provides plots of the currently applied Attune Instrument. The right sidebar provides controls including which tune to show on the plot and information about the available tuning ranges and tunes in each arrangement.

attune-delay The attune-delay daemon provides a quite similar interface to the attune daemon, as shown in Figure 4.17. This daemon performse Spectral Delay Correction (SDC), which accounts for changes in arrival time of laser pulses as a function of wavelength by compensating using the motor that introduces delay. However, since the effective spectral delay is cumulative for multiple light sources, you can select the arrangement in the display controls. Additionally, the Arrangments for an `attune-delay` daemon do not have meaningful ranges and require controls to enable and disable each individual offset. Thus the "instrument" portion of the right sidebar includes boolean checkboxes which call the appropriate method to toggle each offset individually.

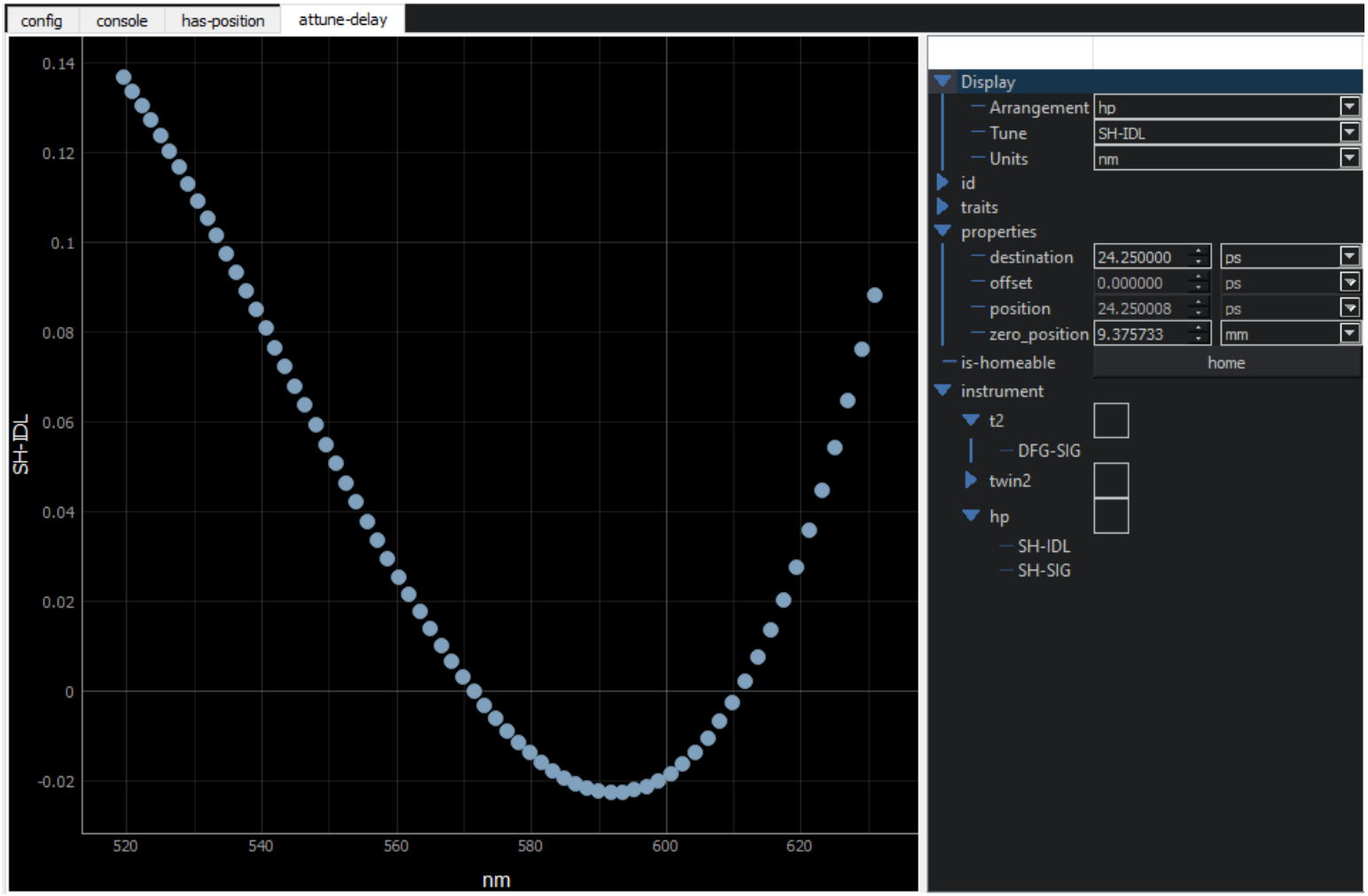

Figure 4.17: The `attune-delay` daemon has a plugin which provides plots of the currently applied spectral delay correction Attune Instrument. The right sidebar provides controls including which tune to show on the plot and information about the available spectral delay corrections.

ni-daqmx-tmux The `ni-daqmx-tmux` daemon has a specialized pluginwhich provides two tabs: one which displays samples from a single shot (Figure 4.18) and one which displays a sequence of measured shots (Figure 4.19). The former is useful for determining which samples are appropriate to use for computation of each channel value. The primary display of this tab is a graph which shows the meaqsured value for each sample in a single collected laser shot. The x axis is sample index, which is analogous to time, roughly equivalent to microseconds. The y axis is voltage measured for a single analog channel. Since this daemon is built for a DAQ that has a single digitizer, each sample represents an instant in time that can only be measured on one physical channel. Thus, to account for multiple sensors, different time windows must be chosen for each sensor. This often means taking advantage of the fact that different sensors have varied instrument response functions, and therefore peak positions in time. However, the peaks are rarely fully discriminated by time alone, so it is often true that the primary signal channel is priorized for capturing the peak, and other channels are captured on the tail. The values that change the behavior of the daemon are all found in the daemon's configuration, and therefore require external editing and restarting the daemon to take effect. The current values are shown in the right sidebar, for convenience. All user editible fields in the right sidebar affect the plot only, no function of the daemon itself.

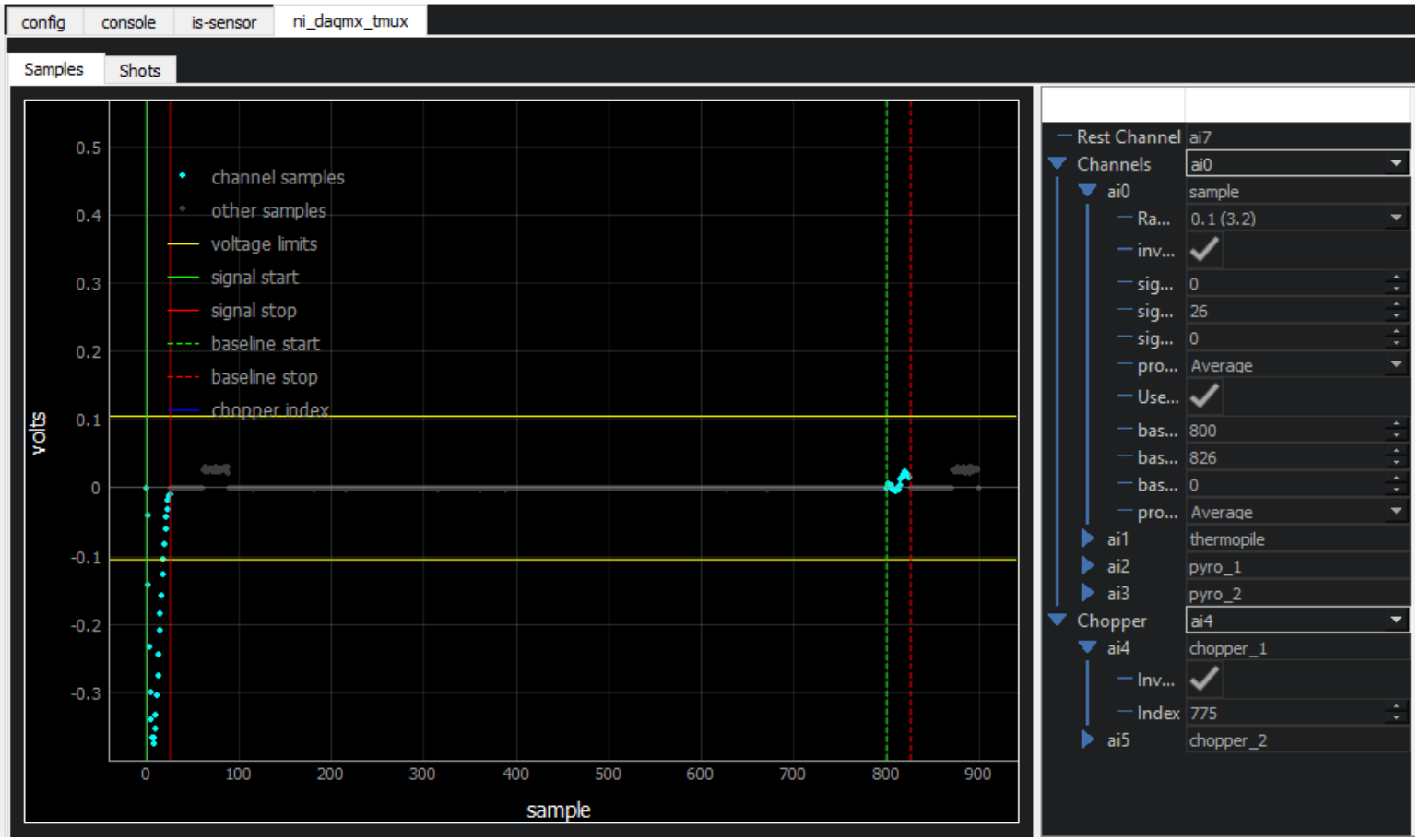

Figure 4.18: The measured samples for the `ni-daqmx-tmux` daemon. A selected channel (the primary signal channel in this case) is highlighted with relevant regions being delineated by marker lines.

The second tab, labeled “Shots”, displays the value of many sequential measurments for a single channel. The primary panel of this window is a graph of this channel, selectable by a drop-down menu on the right. The x axis of the graph is shot index, which is analogous to time and for a 1 kHz laser system is equivalent to one millisecond. The y axis of the graph is the voltage recorded. The particular graph shown as an example is a chopper which is blocking every other shot, therefore oscillating between -1 and 1 Volts on the y axis. This view is useful for evaluating signal to noise and ensuring that laser chopping schemes are functioning as intended.

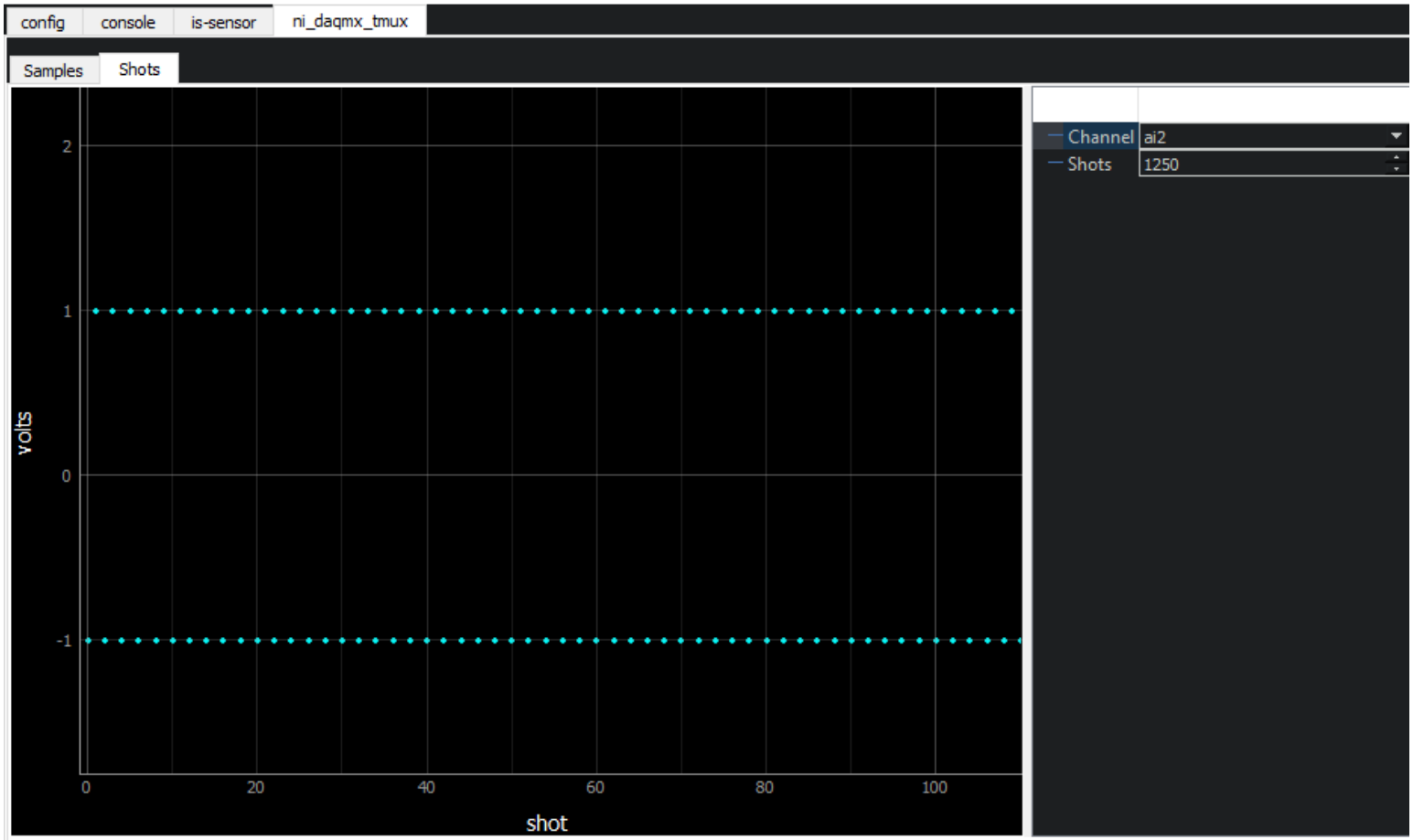

Figure 4.19: The "Shots" tab of the `ni-daqmx-tmux` plugin shows the contents of the measured shots for a selected channel. In this case, the shots are oscillating back and forth for each shot, because this is a chopper channel.

gage-samples The Gage samples tab is provided for the Gage Compuscope daemons. Its primary display is the voltage vs sample index for a single on-board averaged segment. The right hand sidebar provides access to poll for new samples and change the channel that is being plotted. An example screenshot of this panel is shown in Figure 4.20.

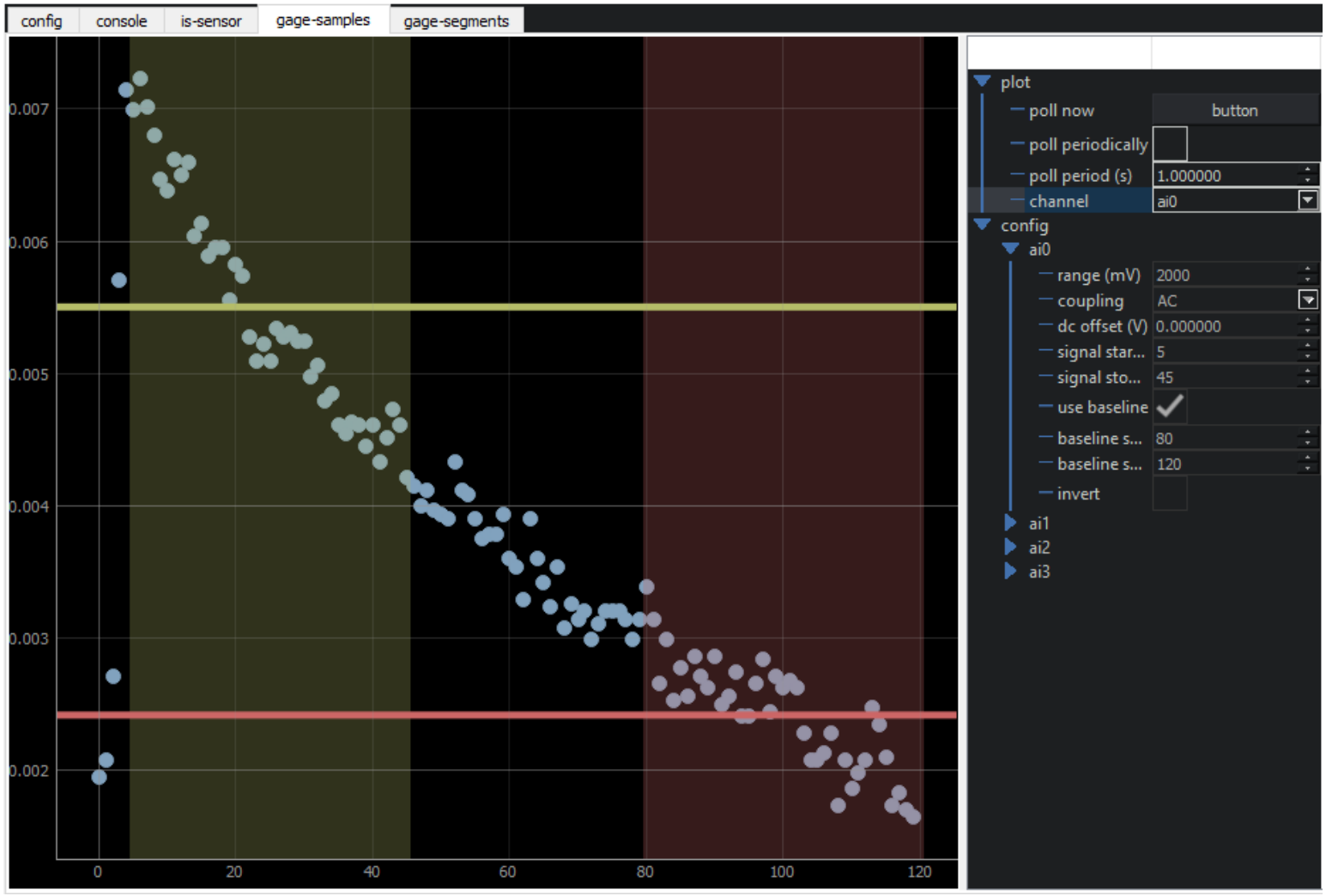

Figure 4.20: The `gage-samples` plugin shows the measured samples of a single segment. The x axis is sample index, which is analogous to time. The y axis is voltage. The two highlighted regions are the signal region (green) and the baseline region (red). The average of each region is shown as a horizontal line.

gage-segments The system which uses this DAQ is capable of a 100 kHz repetition rate. This increased repetition rate means that the chopper settings cannot reliably be synced to the repetition rate. Unlike the NI DAQ used by the 1 kHz laser systems, the Gage DAQ performs multiple levels of averaging, including on-board averaging. The on-board averaging for the Gage DAQ results in "segments" which consist of a series of laser shots averaged together at each measured sample. The "gage-segments" plugin for `yaqc-qtpy`, shown in Figure 4.21, is similar to the "Shots" tab of the NI plugin. However, it is important to acknowledge that each point shown in the "gage-segments" graph may actually represent multiple individual laser shots. The main panel of this plugin consists of a graph which shows the measured value of each averaged segment. The x axis is segment index, analogous to time, though the correlation to seconds is dependent on multiple configuration values, including those of the DAQ itself and of the upstream laser. The y axis is a voltage measurement. When using dual chopping, there are four chopping phases: both blocked, both open, and each chopper blocking alone. Each chopper phase has an associated color and the regions where each are valid are highlighted. The right hand sidebar has controls for updating the plot, including polling for new measurements and changing the number of segments shown and channel that is displayed. Below the plot controls is a portion of the configuration, which may be useful to reference. Finally, there is the properties of the daemon. In this case, these properties will affect the number of averaged shots in each segment, the proportion of segments that are omitted from futher calculation at chopper boundaries and the total number of segments collected to perform a single measurement.

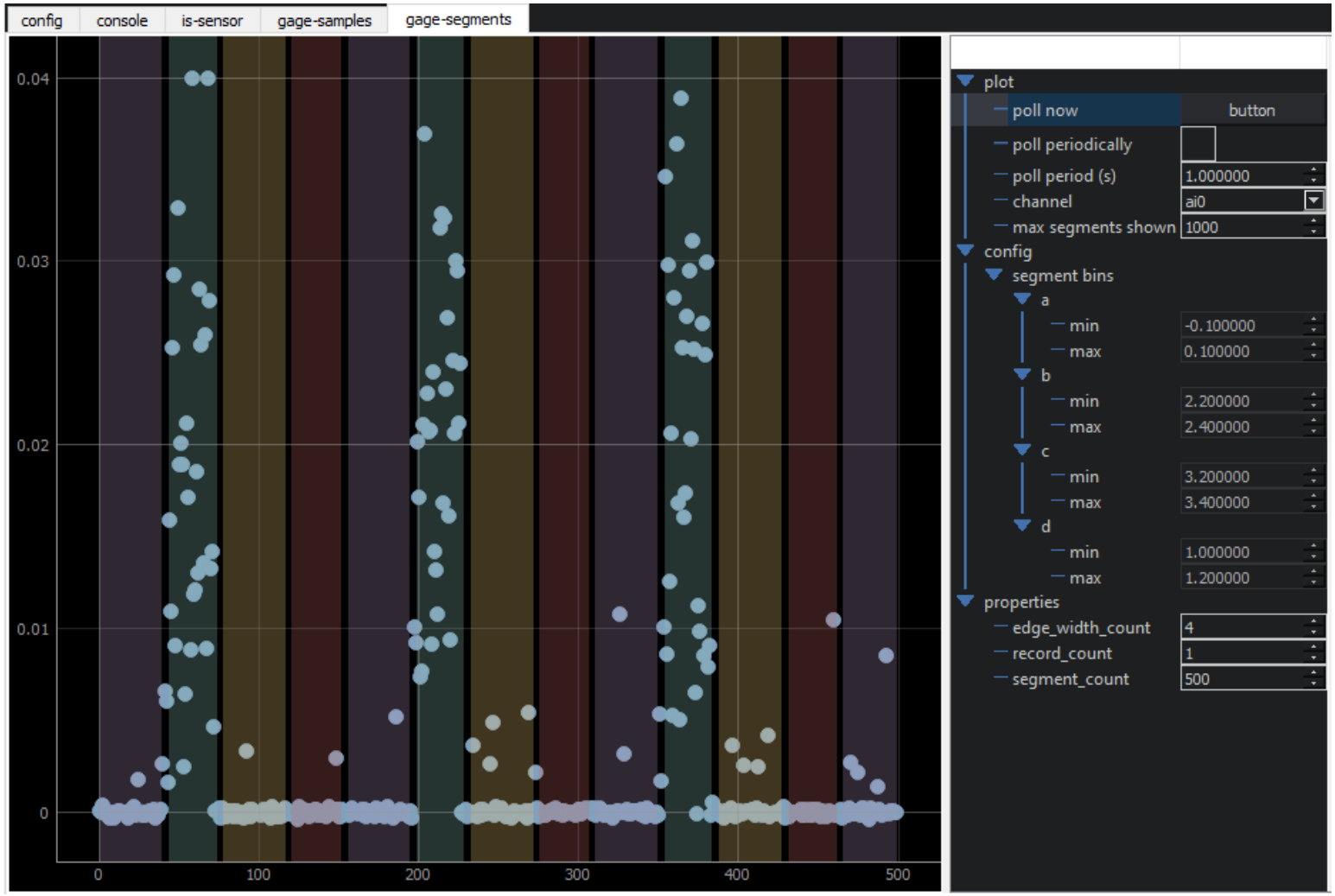

Figure 4.21: The `gage-segments` plugin shows the calculated signal time for a series of averaged segments. This is using dual chopping: the cyan segment is when both lasers are unblocked, red is when both are blocked, and the remaining two are one blocked but not the other.

When chopping, all shots in the same averaged segment must have the *same* chopper phases, otherwise it will produce an intermediate result. Segments with partial occlusion for some shots but not others are omitted from further calclulation. This allows using the chopper at a slower, more reasonable speed while still gaining the noise reduction of chopping. Unfortunately, there is not an easy way to apply a software phase offset of the measured chopper state, as there is for single shot chopping, where you can simply carefully select where to measure the chopper sensor in the shot so that it has a reliable relationship to the passing or blocking of the laser shot. Thus, it is required that the laser passes directly opposite of the sensor, so that the sensor reads the same phase as the laser and is partially occluded in the same segment. When viewing the channel that is used to compute chopper phase, the configured ranges of each phase are shown as horizontal regions. This can be seen in Figure 4.22. Segments which are in those regions are used to compute the different chopper phases and combined to give differential signal. Any segments either not in one of the bins or within a configurable width of the edges of the bins are ignored, as they may have a mix of chopper phases. The numeric ranges of the bins are configurable and shown on the right.

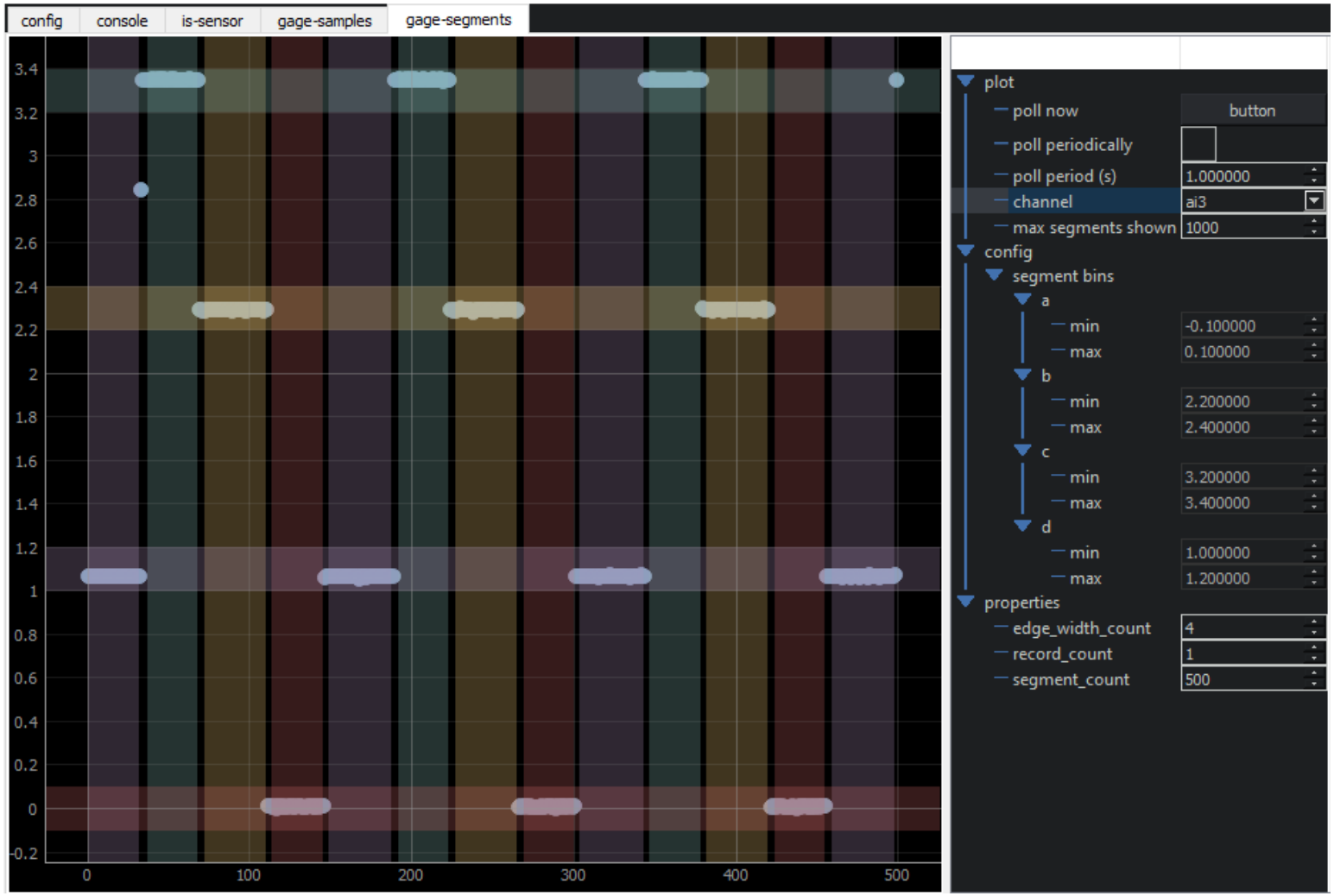

Figure 4.22: The `gage-segments` plugin when viewing the channel used for determining chopping phase. The bins are shown as horizontal regions.

4.3 Guides

This section provides a series of guides for implenting various features in `yaq`. The first guide is a general guide to writing daemons. This is followed by some general guidance on what is meant in `yaq` when we say "configuration" and "state". Then a discussion about the process for creating `yaq` standards is provided. Following this, a series of guides relating to the details of implementing each trait are provided, listed in alphabetical order. These guides are specific to implementing each trait in Python, using the mix-in classes provided by `yaqd-core`. Finally, there are a series of guides for implementing particular common patterns or features of the core Python implementation. This last section has a particular emphasis on properly accounting for the asynchronous nature of the reference implementation.

4.3.1 Writing a Daemon

Writing daemons is one of the key parts of the `yaq` ecosystem. It is designed to be as easy as possible, though precedence is given to ensuring client interactions are easy and consistent. An example of implementing a new daemon, specifically a New Era Instruments syringe pump, is provided as a video tutorial[107].

The first step in writing a daemon is to interface directly with the hardware in question, ignoring `yaq` entirely at first. This includes looking up manuals and other documentation, finding Python libraries built to communicate with the hardware, which may be provided by the manufacturer directly or by the scientific community of users of the hardware. It is reasonable to start in an interactive Python prompt, but it is strongly recommended to write a python script which does each distinct communication with the hardware: determine how to initiate communication, set the position, query the state of relevant variables or measurements, home the hardware, and gracefully shutdown communication. Writing a script is not strictly necessary, but it makes creating the daemon easier, a matter of coping and pasting each individual snippet into the appropriate method of the daemon and hooking in the parameters.

Once communication and proper control of the hardware is established, the first question is if it is the first daemon in a new package or if it is adding to an existing package. The standard practice is to group multiple daemons into packages for each manufacturer. While this is not a hard requirement, it

is a suggestion to balance installing only the packages that are required for a particular system while keeping the maintenance burden manageable. If this is the first daemon for a manufacturer, a new package must be created.

`yaq` provides a "cookiecutter" package template which helps to create new `yaq` packages[108]. Cookiecutter is a Python project that allows a template folder structure to be generated after asking a series of questions to obtain variables that are used for file names and contents[109].

To use the `yaq` cookiecutter package, first install `cookiecutter` itself (use the option appropriate for your system):

```
pip install cookiecutter
conda install cookiecutter
```
(4.31)

And run the command line program, giving the URL for the cookiecutter repository:

```
$ cookiecutter https://github.com/yaq-project/yaqd-cookiecutter-python
project_name [yaqd-boilerplate]: yaqd-new-era
project_slug [yaqd-new-era]:
project_src_dir [yaqd_new_era]:
project_short_description [Yaqd Boilerplate contains all the boilerplate you
↪  need to create a yaq daemon.]: Daemons for New Era Instruments hardware.
version [0.1.0]: 2022.11.0
Select open_source_license:
1 - GNU Lesser General Public License v3 (LGPL)
2 - MIT license
3 - BSD license
4 - ISC license
5 - Apache Software License 2.0
6 - GNU General Public License v3 (GPL)
7 - Not open source
Choose from 1, 2, 3, 4, 5, 6, 7 [1]:
use_pytest [n]:
first_daemon_kind [my-daemon]: new-era-ne1000
first_daemon_module [_new_era_ne1000]:
class_name [NewEraNe1000]:
```
(4.32)

This will result in a series of prompts, with the default choice given in brackets. Some of the later prompts default to values derived from earlier prompts. Some values are required to be valid Python identifier names, specifically those that have underscores instead of dashes and the `class_name`. If you

are okay with the default given, hit enter. Otherwise, provide a valid alternative.

The `product_name` typically starts with `"yaqd-"` and ends with the manufacturer name. The default `project_slug` and `project_src_dir` are derived from the `product_name` and are usually the correct value. While some of the other options will fill out default values in a handful of places, the only other options worth highlighting are the license and the `first_daemon_kind`. The license gives an option of several possible open source licenses, for which the appropriate text will be found in `LICENSE.txt`. `yaq`'s Python implementation itself is licensed under the GNU Lesser General Public License (LGPL), and that is a reasonable default if you are not opinionated about software licenses. `first_daemon_kind` is used to compute the default value for the last two options and represents the `kind` of the daemon for which the boilerplate code is provided. The cookiecutter will automatically create a Python file that has makes starting the new daemon easier. It will also provide the appropriate TOML file with the correct name and headings ready to be filled out.

In addition, the cookiecutter provides a number of files that provide configuration for tools used in the `yaq` ecosystem, including pre-commit[110], and GitHub Actions[111] workflows to run `mypy`[112], publish the package to PyPI[113], and to test that the command line applications provided by the python package run as expected. By default, there is no proper testing infrastructure for actual correctness of running daemons, as most daemons require hardware to even initialize. Instead, the tests ensure that the application runs, providing help and version information even without trying to start the actual daemons.

A full list of files created by the cookiecutter is provided below:

```
├── CHANGELOG.md
├── .github
│   └── workflows
│       ├── python-mypy.yml
│       ├── python-publish.yml
│       └── run-entry-points.yml
├── .gitignore
├── LICENSE
├── .pre-commit-config.yaml
├── pyproject.toml
├── README.md
└── yaqd_new_era
    ├── __init__.py
    ├── _new_era_ne1000.py
    ├── new-era-ne1000.toml
    ├── VERSION
    └── __version__.py
```

(4.33)

The provided `pyproject.toml` file will include the entrypoint that specifies the command line application for the first daemon.

When adding a new daemon to an existing repository, many of the files do not need to be created or edited. A new python file and associated TOML file should be added to the main python package source directory (`yaqd_new_era` in this example). `pyproject.toml` should be updated to include the new entrypoint, which should look similar to the existing entrypoint, but updated with the new file and daemon kind name. The `README.md` should be updated to include a reference to the new daemon. And lastly, the `run-entry-points.yml` GitHub Action should be updated to additionally test the new entrypoint.

The next step is to define the external interface of the daemon, which is done using the TOML file. First consider what traits are appropriate for this daemon. Is it a sensor? or a motor? Does it communicate using UART-style serial? Add the appropriate list of traits to the `traits` list in the TOML file.

Then add any additional messages, configuration, and state to the appropriate sections. If messages are logically connected such that one is a "getter" and one is a "setter" then it is preferable to group these messages into a "property" as well. Each message defines its input and output types.

Once complete, use `yaq-traits` to convert the TOML file into a fully specified Avro Protocol (avpr) file. This completes the target contract that is advertized to clients as the available interface. The

daemon is not complete until this contract is fulfilled.

The next step is to implement the Python module. Import all of the mix-in classes for the chosen traits from `yaqd_core`. Since these are mix-in classes are compositional in nature, but have some dependencies on each other, order does matter for the class declaration line. In general, more specific traits should appear further to the left, and more general traits should appear further to the right, e.g. `UsesUART` should appear left of `UsesSerial`. `IsDaemon` should generally be the last entry in the list.

From there, implement each specified trait, as detailed below. Copy the relevant lines from the script written initially for each method where hardware communication is required. If you are regularly polling information, this is typically done in a method called `update_state`.

Once all traits are implemented, add the implementation for any unique messages that are provided. Ensure that all configuration values that are required are used appropriately and that all state values that are declared are properly written to when they are updated.

Once implemented, the daemon can be installed using `flit`[114]:

```
$ flit install --pth-file
```
(4.34)

This will install the package in editable mode, such that changes to the local directory will change the daemon when it is run.

Before the daemon can actually be tested, a configuration file must be written:

```
$ yaqd edit-config new-era-ne1000
```
(4.35)

At a minimum a section providing the name as a TOML table and the `yaq` port must be provided. If there are additional required configuration values, those must be provided as well, otherwise the daemon will not initialize properly.

4.3.2 Configuration versus State

yaq provides two systems for dealing with values that persist across restarts of daemons: configuration and state. These systems are similar, each providing values as TOML files, and thus the allowed set of values is the same. Configuration values are set by the user and remain the same until the daemon is restarted. State values are updated by the daemon itself, and can change while the daemon is running. This may be in response to user input, such as calling a method over the yaq interface, or simply by virtue of the daemon needing to keep track of its own state.

On the protocol level, configuration and state are defined in the same way: a name mapping to its type, documentation, and default value. A configuration value may have no default, meaning that it is required that the user explicitly provides the value in the configuration file. A state value must have a default, as the daemon may not be able to start without it.

4.3.3 yaq Enhancement Proposals

yaq Enhancement Proposals, or YEPs, are the formal process for standardizing the yaq ecosystem. YEPs are modeled after similar systems used by various open source communities including Python itself[115] and Numpy[116]. YEPs include the formal definitions of various parts of the yaq ecosystem, including the RPC itself and traits.

The purpose of the formalism of the YEP process is to encourage carefully considered additions to yaq. This means that there are a few extra steps compared to what is required for simply implementing an idea. This friction is intentional. The proposol must be submitted, and reviewed by the community, open for comment, criticism, and alternative solutions to the same problem. YEPs start as draft proposals, submitted for initial review. Core team members can ask for clarification of scope and metadata about the YEP itself prior to accepting the draft YEP. Even if the YEP is included as a draft, it has not been accepted as a standard in the yaq ecosytem. Accepting a YEP requires at least two core maintainers to agree that the YEP should be considered a standard.

It is good practice to explain not only the suggested standard, but also briefly explain some alternatives

that were considered and why the chosen standard is preferred.

4.3.4 Implementing traits: has-dependents

`has-dependents` provides a mechanism to discover relationships between daemons. It provides a single method, `get_dependent_hardware`, which returns a map of names to host:port strings.

```
class MyDaemon(HasDependents, IsDaemon):

    ...

    def get_dependent_hardware(self):
        return {"child":
        ↪  f"{self._wrapped_daemon._host}:{self._wrapped_daemon._port}"}
```
(4.36)

4.3.5 Implementing traits: has-limits

`has-limits` augments `has-position` by adding boundaries which are queryable and checked when setting positions. Most of the implementation is handled by the mix-in class, so the only consideration on the implementation side is to set the state value `hw_limits`, which defines the limits imposed by the hardware itself. Sometimes this can be read from the device, other times it is known through documentation, and still others there is no actual way to tell and the `hw_limits` should be set to -infinity to +infinity. Even in the latter case, it can be useful to include the `has-limits` trait because software limits can be imposed by users via configuration.

4.3.6 Implementing traits: has-mapping

`has-mapping` builds upon the `is-sensor` trait by providing access to parallel arrays, such as the wavelength data for an array detector or spatial information for a camera. Since `has-mapping` requires `is-sensor`, all of the details of implementing a sensor apply. Implementing `has-mapping` requires only the setting of some instance attributes which are quite similar to how all `is-sensor` devices manage the channel names, units, and shapes. All of the messages provided by the trait are implemented by

the provided mix-in class, referencing the expected variables. Mappings have their own names, strings which identify them as separable arrays. A channel may have multiple mappings, including multiple mappings along the same axis, this simply means that all of the arrays are parallel. For instance, a camera may provide a mapping for both wavelength information and index information, the latter of which may identify where on the physical camera a selected area of interest is located, while the former provides information of interest to plotting. A channel may also have multiple mappings for different axis, such as the x axis and y axis of a camera.

The three variables are `self._channel_mappings`, `self._mapping_units`, and `self._mappings`. Each of these is a dictionary which provides particular information about the mappings. The first, `self._channel_mappings`, determines which mappings are associated with each channel. The keys are the channel names, and the values are a list of mapping names which apply to that channel. Each channel should appear in this list, but may have zero mappings associated. The second variable, `self._mapping_units` works much the same as `self._channel_units` for any sensor, except that the keys are the mapping names. The values are either `None` for unitless quantities or a string for the unit. Both `self._channel_mappings` and `self._mapping_units` should be static and only set once. Finally, `self._mappings` is a dictionary with the mapping names as keys and the actual arrays (or scalars, though such usage is uncommon) as values. The arrays must have shapes which broadcast to all associated channels identified by `self._channel_mappings`.

If the mapping is entirely static (e.g. it is read from the device and is not dependent on other inputs) then all of the variables can be set in the daemon's `__init__` method. If the mappings are dynamic, then they must be set as a response to updated inputs. The mix-in class manages the state of a fourth variable, `self._mapping_id`, which is an integer that simply increments every time new mapping arrays are provided.

The following snippet shows the relevant portions of a daemon which implements a camera with an area of interest specified by configuration variables and mappings which identify the physical indices of the axes:

```
class MyDaemon(HasMapping, HasMeasureTrigger, IsSensor, IsDaemon):
    def __init__(self, name, config, config_filepath):
        super().__init__(name, config, config_filepath):

  ...

  self.x_index = np.arange(
            config["aoi_left"], config["aoi_left"] + config["aoi_width"],
            ↪  dtype="i2"
        )[None, :]
        self.y_index = np.arange(
            config["aoi_top"], config["aoi_top"] + config["aoi_height"],
            ↪  dtype="i2"
        )[:, None]

        # populate channels
        self._channel_names = ["image"]
        self._channel_units = {"image": None}
        self._channel_mappings = {"image": ["x_index", "y_index"]}
        self._channel_shapes = {"image": [config["aoi_height"],
        ↪  config["aoi_width"]]}
        self._mappings = {"x_index": self.x_index, "y_index": self.y_index}
```
(4.37)

This example if for a daemon with a static mapping, one with a dynamic mapping would include a line such as:

```
self._mappings = self.gen_mapping()
```
(4.38)

Where `self.gen_mapping` is a function which returns a valid mapping dictionary. This line could appear multiple times and does not need to be in `__init__`, though it should be called at least once before a measurement is taken.

4.3.7 Implementing traits: has-position

`has-position` is one of the most commonly used traits, and it is required by several other traits. The central idea is that there is one value, the position, represented as a floating point number, which describes the core functionality of the daemon. This is a value that might be scanned for an acquisition such as the color of a light source or position of a translation stage. In `yaq`, even hardware which is fundementally discrete, such as shutters or valves, are mapped to a floating point position variable for consistency, though interacting using the `is-discrete` trait may be more natural in those cases.

When implementing a `has-position` daemon, there are actually only a small number of things to consider, as much of the functionality is provided by the mix-in class. You must consider the units of the position, if units are provided (they can be `None`, the default, if no units make logical sense). You must implement the procedure for actually communicating with the hardware and setting the position: `_set_position`. This is implemented as a private function (beginning with `_`) because the mix-in class (and, in fact, some of the mix-in classes for related traits) manage setting of some state variables such as `destination`. Next you must implement some mechanism of updating the state variable for `position`. Depending on the hardware interface, this could mean polling the hardware for its direct knowledge of its position, updating the position in response to communication initated by the hardware, or even simply setting the state value in the position setting function if there is no mechanism to query the hardware. Finally, you must carefully manage the `busy` state of the daemon. It is expected that the daemon will return busy at every instance between the request being sent and the motion being complete. `busy` is set to True by the mix-in class when the request is first processed, but it must be set back to False by the daemon implementation. It is common to update both busy and position in the `update_state` asynchronous method, though other options are valid.

```
class MyDaemon(HasPosition, IsDaemon):
    def __init__(self, name, config, config_filepath):
        super().__init__(self, name, config, config_filepath)
        self.units = "mm"
        self.dev = Device() # Some generic manufacturer interface

    def _set_position(self, position):
        self.dev.write(position)

    async def update_state(self):
        while True:
            self._state["position"] = self.dev.read()
            self.busy = not self.dev.is_still()
            await asyncio.sleep(0.1)
```
(4.39)

4.3.8 Implementing traits: has-transformed-position

At first, implementing `has-transformed-position` seems like a daunting task. There are many methods that are implemented which naturally are all interconnected. However, most of the heavy

lifting is implemented by the mix-in class, leaving only a couple variables for the simplest case and optionally a couple methods for more complex cases. If all you want is a simple shift of where the zero position is, then the default implementation will work and all you have to think about is setting the variable `self._native_units` in the same manner as `self._units` for a generic `has-position` device. Consideration for the default or setting the state value introduced by `has-transformed-position`, `native_reference_position`, at initialization time may be useful as well.

More complicated cases, such as those with non-unity transformation functions, must also implement two methods: `_relative_to_transformed` and `_transformed_to_relative`. These two functions work together to provide the scaling factors between the native position and the transformed position. They are referred to as "relative" rather than "native" because the mix-in class still manages adding and subtracting the `native_reference_position`. The two functions must provide the expected inversion such that calling one on the result of the other returns the original input (within rounding errors). Additional configuration and/or state values may be required to properly compute the transformation, but that will depend on the specific circumstances.

Additionally, some implementations may need to override one or more of the methods, for instance `set_native_reference`, to properly update their state, such as the `yaqd-attune-delay` daemon offseting the spectral delay correction curve.

Because the mix-in class relies on calling methods of `HasPosition`, care should be taken to ensure that the `HasTransformedPosition` class appears *before* `HasPosition` in the class declaration.

```
class MyDaemon(HasTransformedPosition, HasPosition, IsDaemon):
    def __init__(self, name, config, config_filepath):
        super().__init__(name, config, config_filepath)
        self._native_units = "cm"
        self._units = "cc"

    def _relative_to_transformed(self, relative_position):
        return relative_position ** 3

    def _transformed_to_relative(self, transformed_position):
        return transformed_position ** (1/3)
```
(4.40)

4.3.9 Implementing traits: has-turret

`has-turret` is designed for devices such as monochromators that have a secondary position called a "turret", such as the ones used for selecting gratings. The trait has three methods, comprising one single `yaq` property: `get_turret`, `set_turret`, and `get_turret_options`. There is also a state value, `turret`, which stores the current value of the property.

To implement `has-turret`, the setter and the options getter must be implemented. The options getter should return a list of string names which are valid options for `set_turret`. The getter is implemented by the `HasTurret` mix-in class, simply returning the value as stored in the state dictionary.

The state value can either be updated when it is set or asynchronously by reading from the device.

```
class MyDaemon(HasTurret, IsDaemon):

    ...

    def get_turret_options(self):
        return ["ir", "vis", "uv"]

    def set_turret(self, turret):
        self.device.set_turret(self.gratings[turret]["index"])
        self.device.set_position(self._state["destination"])
        self._calculate_limits()
```
(4.41)

4.3.10 Implementing traits: is-discrete

`is-discrete` is an addition to `has-position` which allows you to provide a set of named positions which can be set by providing the name instead of the value. The trait provides a standard configuration option, `identifiers`, which allows users to specify the named positions in config. However, some hardware natively support the functionality, so those are allowed to ignore the configuration value and read the list of identifiers from the device. If you are using the configuration values, then the only additional implementation consideration is to update the state value `position_identifier` with the appropriate value (a string indicating the descrete position or None, indicating that the device is not at one of the named positions. What it means to be "at" the named position will vary for the particular

details of the hardware. Some hardware will want a level of tolerance, some will always report an exact number, and some will not natively have any in-between values that are even possible.

```
class MyDaemon(IsDiscrete, HasPosition, IsDaemon):
    ...
    async def update_state(self):
        while True:
            self._state["position"] = self.dev.read()
            self.busy = not self.dev.is_still()
            self._state["position_identifier"] = self.dev.get_position_name()
            await asyncio.sleep(0.1)
```
(4.42)

4.3.11 Implementing traits: is-homeable

`is-homeable` is a trait which provides a single method, `home`, which must be implemented.

Notably, what "home" means in `yaq` is "Go to your limit to determine your absolute position, then return to your current destination". This is often *not* the same as what an individual device will do when homing, which is often only the first half. Additionally, homing is a task which takes time to complete, which is contrary to the `yaq` philosophy which expects methods to return quickly. As such, there are some common patterns to homing hardware where the `home` message itself initiates an asynchronous task that actually accomplishes homing. The details of how each device knows when to move on will vary, but in general it looks something like:

```
class MyDaemon(IsHomeable, HasPosition, IsDaemon):

    ...

    def home(self):
        self._busy = True
        self._loop.create_task(self._home())

    async def _home(self):
        self._homing = True
        self._done_homing = False
        self._busy = True
        self.device.home()
        while not self.device.homed():
            await asyncio.sleep(0.01)
        self.set_position(self._state["destination"])
        self._homing = False
```
(4.43)

Care must be taken to avoid reporting that `busy` is False at any point during the homing procedure.

4.3.12 Implementing traits: is-sensor and has-measure-trigger

Most sensor devices in the `yaq` ecosystem also implement `has-measure-trigger` for software triggering of process of measurement. That trait actually simplifies implementation by providing a standard way of updating the measurements that the generic `is-sensor` daemon cannot guarantee in order to be flexible to more kinds of sensors.

All `is-sensor` daemons must ensure that a few variables are consistent, usually in their `__init__` method:

- `self._channel_names`: a list of string names
- `self._channel_units`: A mapping of the same names from `_channel_names` to string units or None
- `self._channel_shapes`: A mapping of theh same names from `_channel_names` to tuples representing the shape of the channel. If all channels are scalar values, this can be ignored and default behavior will properly report as much.

If you are implementing a `has-measure-trigger` daemon, then much of this bookkeeping is provided by the associated mix-in class. The only thing to do is implement `self._measure()`, a method which

returns the dictionary of channel names to measured values. This is an asynchronous method, so long running data acquisitions, such as waiting for integration times, should not block the daemon such that additional queries are responded to quickly. If the native interface itself must block in a way that asyncio cannot bypass, then the measurment must be completed in a separate thread so that the daemon itself is not completely locked.

```
class MyDaemon(HasMeasureTrigger, IsSensor, IsDaemon):
    def __init__(self, name, config, config_filepath):
        super().__init__(name, config, config_filepath):
        self.device = AsyncDevice()
        self.channel_names = ["ch0", "ch1", "ch2", "ch3"]
        self.channel_units = {x: "V" for x in self.channel_names}

    async def _measure(self):
        out = {}
        for ch in self.channel_names:
            out[ch] = await self.device.read(ch)
        return out
```
(4.44)

If you are implementing a sensor that does not have a software trigger, such as one that subscribes to updates produced elsewhere, then you are responsible for updating the `self._measured` dictionary, including adding the `measurement_id`.

4.3.13 Implementing traits: uses-i2c

`uses-i2c` is a trait which exists solely to standardize the name of the config value for the I2C address, `i2c_addr`.

I2C (inter-integrated circuit) is a low-level communication protocol for serial communication designed for communication between two devices such as microcontollers[117]. SMBus[118] is a similar protocol that is slightly more strict, but largely is interoperable with I2C, and is thus included for the purposes of the `yaq` trait. I2C typically exists on a network with one primary device and a series of secondary devices. The primary device controls the timing and generates requests for data from the secondary devices. The address is an integer which is used to designate which secondary device should accept the data and write responses.

Most of the devices that currently exist in the yaq ecosystem which implement this trait are intended to work with a Raspberry Pi, which provides direct access to an I2C bus via the General Purpose Input Output (GPIO) pins.

As I2C is a serial protocol, use of this trait requires the use and implementation of the `uses-serial` trait.

You can add a default value to your particular configuration if one makes sense for your hardware, though often it is reasonable to say that the address is required in all cases.

4.3.14 Implementing traits: uses-serial

`uses-serial` is a trait which defines only one message: `direct_serial_write(bytes message)`. This trait is required by the `uses-i2c` and `uses-uart` traits. The method provided by this trait is intended for use as a debugging tool, and not for normal operations.

To implement a `uses-serial` daemon, the programmer needs to simply implent the method directly. You may wish to include logging data read from the device as a response to the command, though what makes sense will depend heavily on the hardware.

```python
class MyDaemon(UsesSerial, IsDaemon):

    ...

    def direct_serial_write(self, message: bytes) -> None:
        self.ser.write(message)
        out = self.ser.readline()
        self.logger.debug(f"direct serial write: {out.decode()}")
```
(4.45)

4.3.15 Implementing traits: uses-uart

`uses-uart` is a trait which exists solely to standardize the names provided to common configuration values among Universal Asynchronous Receiver-Transmitter (UART) style communication. Common examples of UART style communication are RS-232 and RS-485.

As UART is a serial protocol, use of this trait requires the use and implementation of the `uses-serial` trait.

These devices will typically appear as a `COM<n>` port in Windows or something similar to `/dev/ttyACM0` in Linux. The operating system specific identifier is the first config field specified by the `uses-uart` trait, `serial_port`. The second configuration value, `baud_rate`, indicates the speed of data transmission, which must be the same for both devices on either end of the transmission. Some devices have variable baud rates, while others have fixed baud rates. In the latter case (or even if it is variable, but the device itself has a default) it often makes sense to add a default value for `baud_rate` in the TOML file for the protocol.

```
traits = ["uses-uart", "uses-serial", "is-daemon"]
[config]
baud_rate.default = 19200
```
(4.46)

As a python implementation, the `UsesUart` class does nothing other than ensure that the required `UsesSerial` class is included in the class inheritance. The config values defined by `uses-uart` are accessed in the normal fashion.

While devices using this trait may wish to consider using some of the more advanced patterns below, it is entirely valid to start with a simple PySerial[81] implementation as shown here:

```
import serial
from yaqd_core import UsesUart, UsesSerial, IsDaemon

class MyDaemon(UsesUart, UsesSerial, IsDaemon):
    def __init__(self, name, config, config_filepath):
        super().__init__(name, config, config_filepath)
        self._ser = serial.Serial(config["serial_port"], config["baud_rate"])

    def close(self):
        self._ser.close()
```
(4.47)

4.3.16 Daemon patterns: loading and saving state

Daemon state management is usually fairly automated such that individual daemon authors rarely need to think deeply on the subject. State files are stored in a standard location as a TOML file as specified by YEP 103[119] At initialization time, the file is read if it exists, and missing values are filled with the default value from the protocol definition.

These values are held in memory in a dictionary called `self._state`, which is updated by the daemon when the daemons state changes, either due to user interaction or due to changes read from the hardware itself.

On a regular schedule (approximately 10 Hz when a daemon is busy, and approximately 1 Hz when a daemon is not busy) the daemon will write out the contents of its in-memory state, if it has been updated. Note that the daemon will acknowledge that the state is updated automatically only if the top level keys or values have been updated. If you have a complicated state such as a nested dictionary, then you may wish to explicitly mark the state as updated by including `self._state.updated = True` when the dictionary is updated without updating the top level of the state.

4.3.17 Daemon patterns: logging

Logging can be a powerful tool for decyphering what a daemon is doing. In the Python implementation, the logging system of `yaq` is integrated with the standard library logging module[120]. `yaq` defines a few extra logging levels to conform to a non-python specific standard initially provided by sd-daemon[121] and syslog[122]. In practice, however, the extra levels are rarely used.

Each daemon instance has its own logger, `self.logger`, which is set up to properly format the log messages and print to STDERR and, if configured, output to a file as well. Daemon implementors are encouraged to sprinkle helpful log messages where it makes sense. Messages which are only useful in debugging specific contexts, particularly those that are verbose to the point of obscuring other log messages, should be included using `self.logger.debug(message)`. By default, these messages will not be displayed, but by passing `--verbose` to the daemon command or editing the configuration

file the debug logs will be included. Messages about ordinary operations, particularly information provided at start up or shut down should be logged with `self.logger.info(message)`. Logs related to errors, such as errors being reported by the hardware, or errors in communication with the hardware itself should be logged using `self.logger.error(message)` Additionally, while less comonly used, `self.logger.warning(message)` exists for warning users of potential unexpected behavior, such as that caused by falling back to default values.

```
class MyDaemon(IsDaemon):

    ...

    def test_logging(self):
        self.logger.debug("This is a debug message")
        self.logger.info("This is an informational message")
        self.logger.warning("This is a warning message")
        self.logger.error("This is an error message")
```
(4.48)

4.3.18 Daemon patterns: async interfaces

The Python implementation of `yaq` makes use of Asyncio[123] to do what is called "cooperative multitasking". Asyncio is built into the Python language and provides additional syntax to declare functions which are called asynchronously. Asyncio has an "event loop" which schedules a series of tasks, tracking dependencies between tasks and allowing those tasks that are able to run execution time. The key idea, and the reason it is called "cooperative", is that each task should not block so as to allow all of the other tasks to complete. Only one task is ever actually running at any given instant, though many may be ready to run and awaiting selection, giving an effect very similar to having multiple threads, but without as much overhead or risk of invalid shared data. Asyncio is good for tasks which are mostly waiting for things to happen such as user input or network traffic. This behavior makes Asyncio an ideal choice for tasks such as networking or other input-output (IO) bound tasks, which is why it is used by `yaq`, a framework that does both networking and IO with hardware. Conversely, Asyncio is a poor choice for computation heavy tasks, as it does add overhead and a long running computation will prevent other tasks from receiving computation time.

While the internals of the core `yaq` library make extensive use of Asyncio to manage incoming RPC messages and running multiple daemon instances in the same process, where possible the usage of Asyncio is not something that daemon implementors need to spend a lot of time worrying about for most messages. There are some exceptions, though, as any long running tasks require at least a little bit of knowledge of Asyncio and best practices. The actual python function called as a response to an incoming RPC message is always synchronous. It is expected to itself return quickly, which is better for both the daemon process itself and for clients. The synchronous method that is directly called may, however, initiate an asynchronous task.

Asyncio introduces two additional keywords for the Python language: `async` and `await`. `async` is used to declare functions as asynchronous and `await` is used to call such functions, preventing the calling function from continuing until a result is provided:

```
async def async_function(self):
        await asyncio.sleep(1)
```
(4.49)

Functions which have the `async` keyword are often referred to as "coroutines" to indicate that the mechanism by which they are called differs from standard function calls and that they are part of the cooperative multitasking. `await` can only be used within an asynchronous function, so to start a parallel task from a synchronous method you would do:

```
def sync_funtion(self):
        self._loop.create_task(async_function)
```
(4.50)

This works in `yaq` because there is already a running event loop and each daemon instance holds a reference to the event loop.

When writing an asynchronous method, you must be considerate of other tasks that may be running. This means that there should not be places where the code is blocking, waiting for something to happen. Instead, use an asynchronous equivalent where possible. For instance, instead of using `time.sleep(1.0)`, use `await asyncio.sleep(1.0)`. An increasing number of libraries are either implementing an Asyncio interface or have an alternative dependency that is only Asyncio, consider using

these interfaces if ones built specifically for your hardware exist. The `await` keyword returns execution control to the scheduler and allows other tasks to complete rather than holding on to the processing. If there is a way to wait specifically for something to happen or a result to be finished, rather than polling, that is preferable. If there is a loop, especially one that may be an infinite loop, ensure that there is an `await` statement for every possible code path. Even if you have no explicit thing to wait for, there should be a line for `await asyncio.sleep(0)`. This tells the scheduler that other tasks can run, but if no other tasks are ready, this task can continue immediately. This ensures that other tasks get an opportunity to run, rather than having one task which monopolizes the computation time. In particular, if you are handling exceptions, make sure that an await happens either before the exception can occur or in the `except` block.

There are three main areas that a daemon implementor needs to think about Asyncio: monitoring state updates from the daemon that are not in response to client messages, homing motors, and reading sensors.

Monitoring state updates

For a lot of hardware, there are messages to determine the state of the hardware such as position or whether or not it is actively moving, as read directly from the hardware interface. It is generally good practice to poll this information to maintain a complete and correct understanding of the state of the hardware. However, these updates are not raised as a reaction to direct client RPC calls, and instead have to happen in parallel, as a separate Asyncio task. By convention, a task called `update_state` is used for this purpose. There is nothing actually special about this method other than that it is explicitly cancelled upon shutdown, automatically. Users are free to start their own tasks if it makes logical sense to have more than one task running in parallel. In fact, by adding any created tasks, as returned by `self._loop.create_task`, to the `self._tasks` list these additional tasks will be similarly cancelled at when a shutdown is requested. By default, the method does precisely nothing, simply returning and ending the task.

When it is used, it almost always has a loop, often one that is intentionally infinite. Take, for example, the version of the `update_state` method from the Attune Delay daemon:

```
async def update_state(self):
        """Continually monitor and update the current daemon state."""
        while True:
            self._busy = self._wrapped_daemon.busy()
            self._state["position"] =
            ↪  self._to_ps(self._wrapped_daemon.get_position())
            if self._busy:
                await asyncio.sleep(0.01)
            else:
                await asyncio.sleep(0.1)
```
(4.51)

This method starts with `while True:`, which is an infinite loop. There are no `break` statements, nor do we expect in normal operation for there to be any exceptions raised. The task will not terminate without being cancelled from another task. Since this particular daemon wraps another daemon, the busy state is read and updated, along with a small amount of calculation for the position. Importantly, this is followed by an `await` statement, though the poll freqency is faster when busy compared to when it is not busy.

Homing

An example of this pattern is already shown above in the section on implementing the `is-homing` trait. Essentially this is a case of the synchronous method needing to initiate a longer task. For some hardware, it may make sense to treat simple setting of position in much the same way, though most manage it by polling the state for updates. There are often additional flags or Asyncio events that must be managed in relation to homing, the specifics will vary for particular hardware.

Reading sensors

The abstract method for all sensors that implement `has-measure-trigger` is an asynchronous method, `self._measure`. In many cases, the fact of this being asynchronous can be largely ignored. If the hardware has an interface which returns quickly and there is no necessary waiting period such as an integration time, then this method can simply communicate with the hardware and return. It is not necessary to have an `await` statement just because it is an `async` function.

The reason the abstract method is asynchronous is so that it generalizes to hardware which may have either waiting periods or natively asynchronous communication. If it had been implemented as a synchronous method, then implementing such interfaces would have been much more difficult. Conversely, since it is an asynchronous method, those implementations that would have been served by a synchronous implementation only have an additional keyword.

Using Asyncio works well for sensors which have a "kickoff" method to initiate the measurement which is not itself a blocking method call. These sensors typically also have a method that informs the caller of whether or not the measurment is done. Such methods can be polled with an `await asyncio.sleep` call to limit the poll rate and allow other tasks to complete. Finally, when it is ready the actual data can be read and returned.

There are some sensors that do not fit well into that pattern, though. Some sensors only provide fully blocking calls that wait for the entire acquisition time. In this instance, the blocking method can still be used, but should be wrapped and called in a way that does not prevent the Asyncio event loop from running additional tasks. This is done by using `self._loop.run_in_executor`, which is a method provided by Asyncio that runs a blocking call in a separate thread, and allows the calling funtion to `await` the result as if it is an asynchronous function. This is not preferable behavior, as the standard hazards of using threads once again apply, even the ones that using Asyncio usually sidesteps, but it allows for daemons with such hardware interfaces to behave correctly.

An example of a blocking interface using a thread pool executor:

```
async def _measure(self):
        arr = await self._loop.run_in_executor(
            None,
            self.device.trigger_and_read
        )
        return {"chan": arr}
```
(4.52)

4.3.19 Daemon patterns: busy

The concept of `busy` is something all `yaq` daemons have in common. The exact specifics do vary, but generally for motors it means that the motor is actively moving and for sensors it means that the daemon is acquiring a measurement. A sensor that is looping or a sensor that does not implement `has-measure-trigger` will always report that `busy` is True. This perhaps unintuitive choice was made so that a sensor that is in looping mode is no different to a client that is expecting a non-triggered sensor.

In general, `busy` gets set to True when an RPC message which changes the state of the daemon is called, and is set to False when that change is complete. This is then used to guard against unexpectedly using the result of RPC calls when the daemon is in a state of flux or otherwise is not ready. When possibly, busy can be read directly from the hardware, though care must be taken to ensure that if a change has been requested busy never reports False until that change is complete. A specific (but recurring) instance of this idea is explored in detail below.

`busy` is a boolean flag that can be read internally by daemon methods using `self._busy` and by clients over the RPC by using `client.busy()`. If one wants to wait for a specific state, it is possible to simply poll the busy state until it is either True or False, whichever is required. While that is the only mechanism available to clients, internally there are additional features that allow you to explicitly wait for the rising or falling edge of `busy` without polling. There is a pair of `asyncio.Event` objects[124] which allow explicit awaiting on either busy or not busy, `self._busy_sig` and `self._not_busy_sig`. These provide signals that the Asyncio event loop explicitly knows that tasks waiting on them are not ready until the associated `event.set()` method is called. These `set` methods are called (or reset) whenever `self._busy` is updated, so daemon implementors need only to properly update that boolean for everything else to work as expected.

Example usage of the `busy` Asyncio features:

```
async def update_state(self):
    while True:
        await self._not_busy_sig.wait()
        self._logger.info("No longer busy")
        await self._busy_sig.wait()
        self._logger.info("Busy again")
```
(4.53)

4.3.20 Daemon patterns: async serial devices

One of the most common hardware interfaces is RS-232 style serial communication. While the exact protocol for what bytes get sent over the wire differs greatly, many of these interfaces share common behaviors. Often there are multiple devices that are addressed and communicated to over the same serial bus. In some cases, the information provided by replies from hardware is insufficient to in and of itself identify which method is being replied to. For this hardware, careful timing control must be used to correlate the request to the reply. In other cases, the reply itself contains both information about which device and what information is contained in the message. Sometimes, these self describing messages can even originate directly from the hardware, with no explicit request from the daemon side of the communication.

When the messages are self describing, a good pattern is to allow the write actions, that is requests for information or commands to move, etc., to happen in multiple places, but have a centralized task to interpret the replies. Using this pattern, the code can be more logically separated, and there is less strict timing considerations regarding ensuring that you get the reply to the message you requested specifically. This is because all replies are handled, but with grace for time between replies being variable, and multiple requests can be pending.

However, there are some common pitfalls of this approach, for which care must be taken. Because the Asyncio scheduler does not guarantee order of execution (and simply because communication with hardware takes time), there may be replies that are not processed by reading task that indicate that the device is not busy, even when a move has been requested. If care is not taken to account for this, the reply handler will accept that the daemon is not busy and clients may read that it is not busy and assume it is safe to continue. The same (though potentially even more avenues for "not busy" to be

unintentionally reported) can be said for homing of motors that communicate in this way. That said, once you understand what is happening, it is not so difficult to enforce correct external behavior using a few additional boolean flags.

To put this in concrete terms, I will walk through the Thorlabs Ell series daemon. This daemon uses a class called a `SerialDispatcher` which buffers write actions to ensure that the devices have time to process the commands and does some parsing of all incoming data from hardware. The dispatcher is responsible for determining that a message is correctly formed and which specific daemon instance should receive the message. The dispatcher is not required for all hardware, particularly if there is never more than one device on the serial bus.

First, the `__init__` method:

```
def __init__(self, name, config, config_filepath):
    self._homing = True
    self._homing_sig = asyncio.Event()
    self._move_started = False
    self._address = config["address"]
    if config["serial_port"] in ThorlabsEllx.serial_dispatchers:
        self._serial = ThorlabsEllx.serial_dispatchers[config["serial_port"]]
    else:
        self._serial = SerialDispatcherEll(
            aserial.ASerial(config["serial_port"], config["baud_rate"])
        )
        ThorlabsEllx.serial_dispatchers[config["serial_port"]] = self._serial
    self._read_queue = asyncio.Queue()
    self._serial.workers[self._address] = self._read_queue
    super().__init__(name, config, config_filepath)
    self._units = config["units"]
    self._conversion = config["scalar"]
    self._serial.write(f"{self._address:X}gs\r\n".encode())
    self._state["status"] = ""
    self._tasks.append(self._loop.create_task(self._home()))
    self._tasks.append(self._loop.create_task(self._consume_from_serial()))
```
(4.54)

`_homing` and `_move_started` are the two boolean flag variables that will help ensure that `busy` is properly reported at all times. `_homing_sig` is an Asyncio Event, much like those associated with `busy`. It is used to inform the homing task that homing is complete so that it can reset the position to the current destination. The `SerialDispatcher` objects are shared among all daemons that communicate over the same serial bus, so if it already exists, the instance retains a reference to the previously made

dispatcher, and creates a new one if it does not already exist.

`read_queue` is an Asyncio Queue[125] object which is a first-in first-out (FIFO) queue that is appended to by the dispatcher, and consumed by the daemon. It is added to the serial dispatcher with the address as a key.

At startup, the daemon requests the initial status using `gs`. It also requests a Home operation and initiates the `_consume_from_serial` task, each of which are added to the list of `self._tasks` which are properly cancelled upon shutdown.

`_consume_from_serial` is the task which waits for messages from the hardware and properly updates the daemon's internal state accordingly:

```
async def _consume_from_serial(self):
    while True:
        comm, val = await self._read_queue.get()
        self.logger.debug(f"incoming serial: {comm} {val}")

        if "PO" == comm:
            position = struct.unpack(">l", bytes.fromhex(val))[0]
            position /= self._conversion
            self._state["position"] = position
        elif "GS" == comm:
            hw_busy = int(val, 16) != 0
            if not hw_busy and self._homing:
                self.homing_sig.set()
            self._busy = hw_busy or self._homing or self._move_started
            self._state["status"] = self.error_dict.get(int(val, 16), "")
            if int(val, 16) not in (0, 9):
                # ignore normal busy/ready mode, log any other error
                self.logger.error(f"ERROR CODE: {self._state['status']}")
        else:
            self.logger.warning(f"Unhandled serial response: {comm}{val}")

        self._read_queue.task_done()
        if self._read_queue.empty():
            self._move_started = False
```
(4.55)

The `await` for this `async` method is not a simple poll frequency, but rather waiting for a preprocessed message from the dispatcher. The message comes as a tuple of the command ID and the remaining unparsed contents, which contain the actual information. Unrecognized messages are logged, as well as reported error states from the hardware. Importantly, the line which sets `self._busy` does not only

account for the hardware busy state, but also for the daemon's own knowledge of whether a home or set position has been initiated, using the boolean flags. At the bottom, the queue marks the task as processed and resets the `_move_started` flag if there are no messages left to process. This works specifically because the problem of having unprocessed messages which contradict the correct `busy` state is no longer possible once there are no active messages. The hardware has been informed to move, and will correctly report `busy` in any messages received *after* the write is complete.

However, processing incoming information only happens for this device if it has been requested, so this daemon uses `self.update_state` to request updates for the status and position:

```
async def update_state(self):
    while True:
        if not self._move_started:
            self._serial.write(f"{self._address:X}gs\r\n".encode())
        if not self._homing and not self._move_started:
            self._serial.write(f"{self._address:X}gp\r\n".encode())
        await asyncio.sleep(0.2)
```
(4.56)

The poll frequency matters, as if you ask faster than you can parse the replies, the queue will never empty. A faster polling frequency will give finer grained updates in position. Ideally, the polling is at a rate where it does not matter if you are one polling period later for the purposes of timing control, but that the replies are always processed before new requests are made. To fully ensure that the queue empties when the timing control requires it, fewer additional replies are requested when the boolean flags are set, though homing does rely on the status being updated. Note that in this case, `update_state` does *cause* the state to be updated, but the actual updates are in `_consume_from_serial`.

Finally, `home` and `set_position` take into account the added variables:

```
def _set_position(self, position):
    self._move_started = True
    if not self._homing:
        pos = round(position * (self._conversion))
        pos1 = struct.pack(">l", pos).hex().upper()
        self._serial.write(f"{self._address:X}ma{pos1}\r\n".encode())

def home(self):
    self._busy = True
    self._loop.create_task(self._home())

async def _home(self):
    self._homing = True
    self._serial.write(f"{self._address:X}ho0\r\n".encode())
    await asyncio.sleep(0.2)
    await self._read_queue.join()
    self._homing_sig.clear()
    await self._homing_sig.wait()
    self._homing = False
    self.set_position(self._state["destination"])
```

(4.57)

_set_position flips _move_started to True and does not preempt an active home. If a home is occurring, the position will be properly set at the end of the home operation anyway.

Homing may seem complicated at first, but is actually only a few simple ideas. First, there is the homing flag, which surrounds all of the homing operation, set to true at the start and False at the end of the method. This ensures that busy never goes False during this method (and the behavior of set_position itself ensures it never goes False until that move is complete as well). Next, the method actually writes the home command, and waits a short bit to ensure that the message gets sent since the Serial Dispatcher buffers those commands and writes them asynchronously. Third, it clears the incoming message buffer, ensuring that any messages that had outdated busy state information are processed while the homing flag prevents busy from being False. Next it resets the signal used to determine if homing is complete and waits for it to be set by _consume_from_serial. Finally, it sets the position to the expected destination.

Chapter 5

Acquisition

5.1 Introduction

This chapter details several iterations of data acquisition software used by the Wright Group. The first, PyCMDS, is a program originally written by Dr. Blaise Thompson. Upon joining the group, I took over some of the maintainership of this program, and moved to primary maintainer upon Dr. Thompson's graduation. Next, PyCMDS was modified to use `yaq` for hardware control, and was given a new name, `yaqc-cmds`. Dr. Thompson did help with the transition, though continued maintenance remained under my purview. Finally, we entered into a collaboration with people running many large scale facilities to use Bluesky. Bluesky is a project that was originally developed at Brookhaven National Lab for controlling the National Synchrotron Light Source II. While many parts of the Bluesky system, including the core "Run Engine" and the "Queue Server" are written and maintained by external organizations, many projects required particular attention to adapt these powerful tools to the Wright Group. In particular, I (with consultation with Dr. Blaise Thompson) introduced `wright-plans` , `bluesky-in-a-box` , and `bluesky-cmds` . `wright-plans` provides specialized experimental procedures used by the Wright Group. `bluesky-in-a-box` bundles services written elsewhere into a single application that can be run more easily. Additionally, it adds writing of WrightTools data files from acquisitions taken using Bluesky. `bluesky-cmds` provides a graphical user interface for interacting with the upstream-provided Queue Server.

5.1.1 PyCMDS: Monolithic Data Acquisition Program

PyCMDS is a program originally created by Dr. Blaise Thompson to perform CMDS experiments[126]. Prior to PyCMDS, the Wright Group had been through several iterations of programs perform experiments, primarily written using LabVIEW[76][75]. As the name suggests, PyCMDS was written in Python. This jump to a standard text based language meant that changes to the software could be shared more easily. PyCMDS was the first time the Wright Group wrote acquisition software to run on multiple laser systems. To achieve this, all of the possible known hardware interfaces were included in application code and a series of configuration files were used to specify which hardware was active.

PyCMDS was a monolithic program, incorporating everything from communicating with hardware to

orchestrating motor motion, to recording data files, and providing graphical user interfaces. While "modularity" was an idea that occurred for PyCMDS, here it meant that internal structures were reusable, as opposed to allowing integrations with external programs. Parts of PyCMDS were tightly coupled, meaning that deep knowledge of the entire system was required for even seemingly small changes.

PyCMDS introduced the idea of "Scan Modules", separately parameterizable instructions to do different kinds of data acquisitions. Initially, the goal was for scientists to be able to create their own scan modules for each new kind of experiment. However, the complexity of actually implementing them, combined with lack of documentation and the raw flexibility of the "default" module, "SCAN", led to no one actually implementing many additional modules. The concept of "Constants", expressions which are used to set hardware that is not an axis of the scan but nonetheless needs to be moved for the desired slice, was introduced by the "SCAN" module.

While the design of PyCMDS was good enough to get it running two laser systems that are quite similar in nature, if consisting of different parts from different manufacturers, its faults were evident. Virtual hardware was implemented to allow for development of PyCMDS away from the active instruments, but even then it made some development tasks arduous. Ultimately, the rigidity of some hardware interfaces led to the creation of `yaq`, and the first inclusions of `yaq` in PyCMDS. Specifically, some of the hardware interface code was not able to run on Windows 10, making upgrading the system impossible without a large effort. That upgrade path, along with some issues relating to losing motor positions, was the driving factor in the creation of `yaq`.

5.1.2 `yaqc-cmds`: Steps towards modularity

While the transition was not entirely made overnight, upon removing the last of the hardware interface code from the acquisition program, we decided that it deserved a new name, `yaqc-cmds`, pronounced "Yak C Commands". This name comes from the fact that it is a `yaq` client, the "yaqc", and that it instructs, or "commands" the instrument to acquire CMDS data, the "cmds". This name represents one of the biggest steps in the modular approach to instrumentation. `yaq` plays a central role in the data collection for the program.

To users, the change was fairly minimal: the user interface was largely unchanged and the available acquisitions were the same. At the same time, the other big change to the program was that the output data format was updated to be the HDF5-based "WT5" format from WrightTools, rather than the plain text PyCMDS data format. This data format change allows for better representation of multidimensional data sets, especially those collected by using array detectors or cameras, while keeping data file sizes manageable.

`yaqc-cmds` is a program attempting to be a one stop shop for both engineering tasks like working up initial signal and for actual data collection. Figure 5.1 shows the main queue panel of the application. Along the left side of the program is a control panel for the hardware known to the program. This allows setting of position and associated values whenever no scans are currently acquiring. In the middle is the queue itself, running top to bottom in order of execution. On the right, there is a panel that gives controls for the queue, including creating a new queue, running the queue, and enqueueing new items.

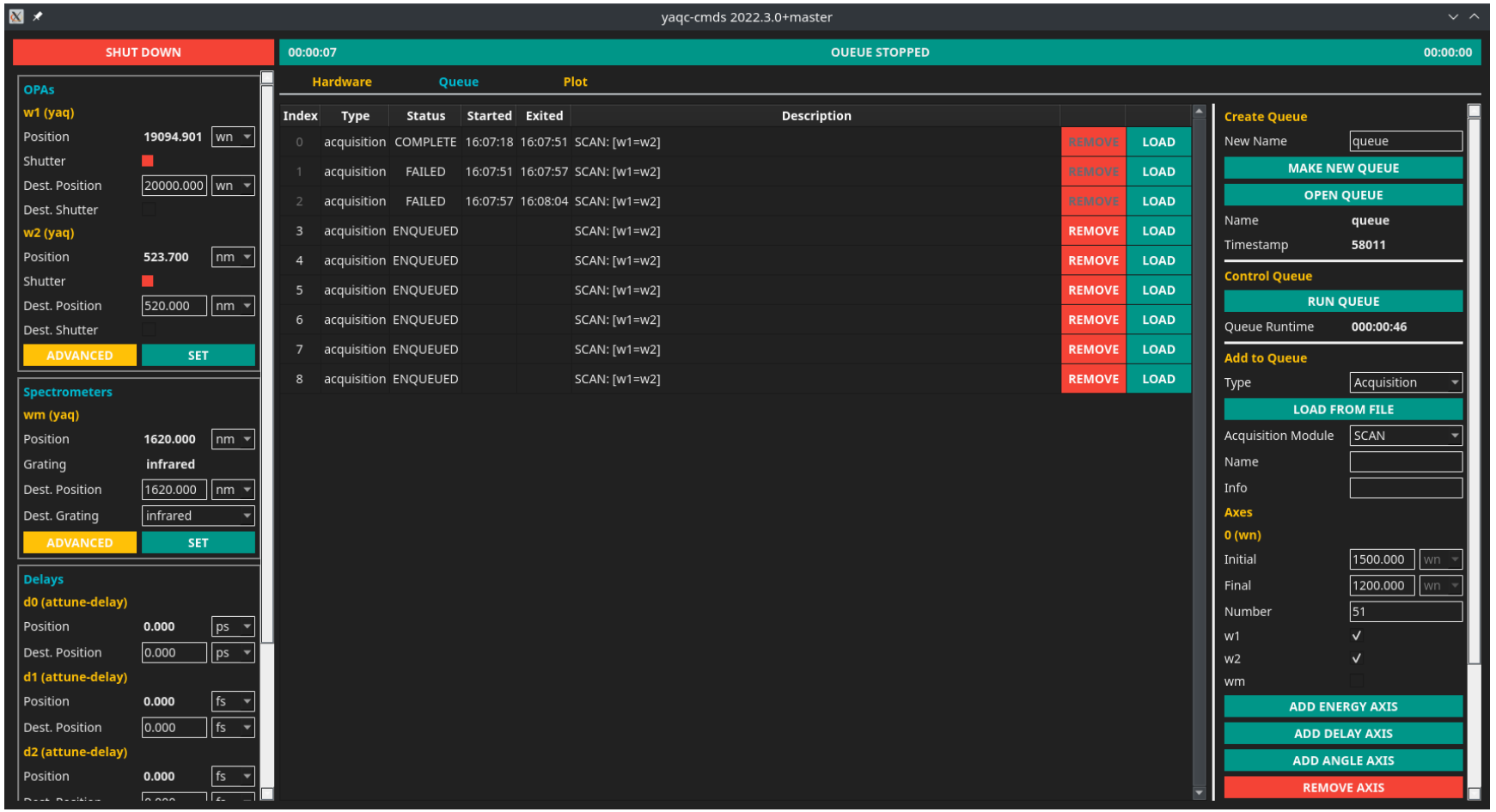

Figure 5.1: The main queue window of `yaqc-cmds`.

In addition to the queue, `yaqc-cmds` has advanced menus for each hardware, including the menus that eventually became part of `yaqc-qtpy` for the Attune daemons. These advanced menus were much more critical prior to using `yaq`, but were kept for consistency.

Finally, there is a live plot tab which plots individual slices of data as they are collected. When the queue is idle, a large number at the top displays the most recent reading from a sensor, which is being polled.

While less monolithic than its progenitor, `yaqc-cmds` retained some of its faults. Writing new acquisition modules, while possible, was not well documented and was challenging to do correctly without deep knowledge of the internals of the program. This meant that most users, even advanced users, had little to no path to actually realize new ideas they had. There is no mechanism for running scans outside of the application, so developing new acquisition modules necessitated also implementing their user interface, and the systems were deeply integrated together.

The program also required carefully constructed configuration files. While each section was relatively easy, `yaqc-cmds` expected all enabled hardware to be available when the program starts, and would freeze or fail to open if even one instance of the expected hardware was missing or unresponsive. Users would frequently need to reconfigure and restart the program when hardware configuration changed, such as enabling or disabling a particular sensor.

Ultimately, the flaws of `yaqc-cmds` led to the separation of the program into `yaqc-qtpy` for the engineering focused tasks and a new program for the task of data acquisition. This new program builds on top of an ecosystem provided by the broader scientific community, Bluesky[71].

5.2 `bluesky`: Fully modular data acquisition

5.2.1 Bluesky

Bluesky is a project initially created for the National Synchrotron Light Source II at Brookhaven National Lab. The project has grown and is now a collaboration from multiple facilities and experimental domains. Bluesky as a project provides a collection of Python libraries with an aim towards standardizing experimental data collection to be useful at a variety of scales, from a single machine instrument all the way to the large synchrotron facilities.

Bluesky provides a set of standards for how to interact with hardware, and how to format data collection. It provides libraries for interacting with hardware, and for handling the data that is collected. At the center is the Bluesky library itself, which provides the orchestration of experiments.

The libraries provided by the Bluesky project are often developed together, providing integration where needed. However, the libraries are still separable, allowing for alternatives to be incorporated with ease to suit particular needs of an instrument. For instance, while the Bluesky Project provides Ophyd[127], a useful hardware abstraction library with a particular focus on EPICS process values, the Wright Group has provided an alternative hardware interface based on `yaq`, `yaqc-bluesky`[128].

The Event Model

The Event Model[129] is Bluesky's mechanism for representing collected data. Each experiment is broken down into a sequence of JSON-compatible objects with particular structures. Figure 5.2 demonstrates pictorially the four most useful Event Model document types. An experiment, or "run", begins with a "start" document. The start document contains metadata which pertains to the entire experiment. This includes things such as the parameters of the requested scan, the expected number of points, and user provided information. Each run is give a UUID which provides a unique key which serves as a name for that particular instance of data collection. A data stream then begins with a "descriptor" document. The descriptor document provides a set of variables collected together as a "row" of data. The descriptor provides metadata pertaining to each individual value collected such as the source of the

variable, the shape of the data, the data type, and its units. The descriptor contains a reference to the run start identifier, and also provides its own UUID and human readable name. The actual recorded data points then are provided by "event" documents, which provide timestamps and values for each key listed by its associated descriptor document. Finally, a "stop" document closes a run, referencing the UUID of the start document, providing an exit status, error message for failed scans, and information about the actual number of collected events (which may differ from the expected number from the start document).

These four basic events are sufficient to represent the data from most experiments. The Event Model does provide some additional kinds of documents for specialized tasks such as referencing data that is stored elsewhere (which is useful for large array data) and for bundling multiple events into one document. However, the details of those document classes are not important for understanding how Bluesky works and records data.

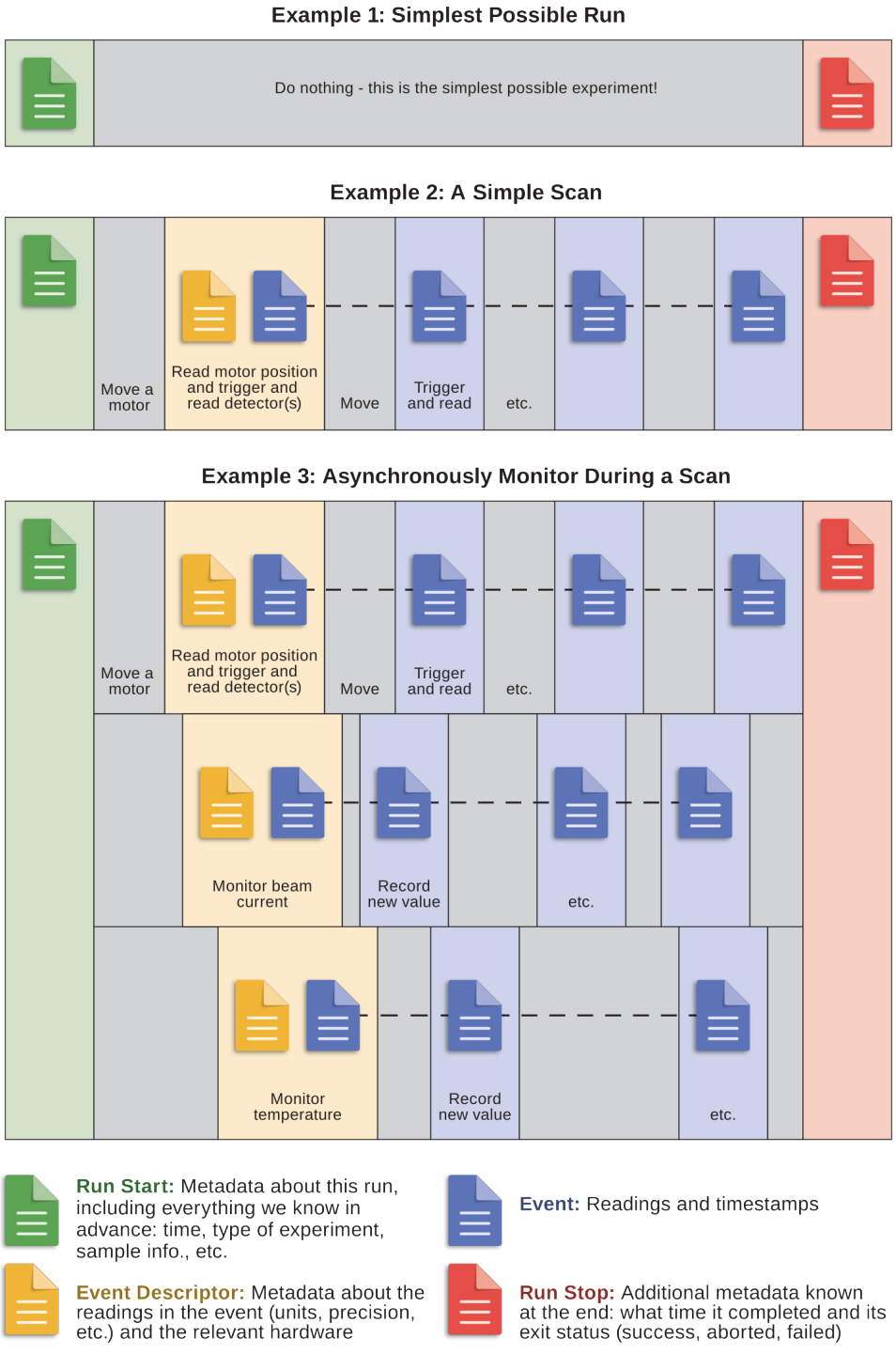

Figure 5.2: This figure demonstrates some common usage patterns of the event model. A "run" is opened with a start document and closed with a "stop" document. Data streams are initiated with a "descriptor" document and populated with "event" documents. Multiple streams can be included in a single run. This figure was created by the Bluesky project and is reproduced with permission.

The Run Engine

The Run Engine is the beating drum of a Bluesky experiment. At its core, the Run Engine is a loop that consumes "Messages". Messages are instructions which the Run Engine knows how to process. Some messages cause interactions with hardware, such as setting motor positions or initiating data collection, while other messages output Event Model documents to a series of callback functions. The callback functions which subscribe to the events can be used to do a variety of tasks from storing data into a file on disk, to printing a table to the console, to plotting data as it is collected.

Plans

Plans are the source of messages to the run engine. A plan is a Python generator which yields a series of Message objects. Plans are parameterized to make common acquisitions easy to specify. The Bluesky Python library comes with a set of pre-assembled plans which allow specification of a variety of trajectories through motor space and collecting data from sensors at each point. While the built in plans are sufficient for many experiments, there are often either additional functionalities or better parameterizations for domain-specific tasks which require specialized plans.

5.2.2 `wright-plans`

Broadly, `wright-plans` provides wrappers of the standard Bluesky built-in plans which add unit awareness and the idea of "constants" allowing setting of motors to algebraic expressions of other motors. These plans each bear the name of the underlying Bluesky plan with the suffix "`_wp`" which simply stands for `wright-plans` and is included to differentiate from the built-in plans. Additionally, `wright-plans` provides specialized plans for performing tuning operations of OPAs.

Common parameters to multiple plans

Many of the plans specified in `wright-plans` share common behavior and meanings assigned to parameters. Rather than repeating the details as it pertains to each plan, this section provides how each

of these are specified and used.

Detectors In all of the plans provided by `wright-plans` , the first argument is a list of detectors, i.e. a list of Bluesky protocol compliant objects that implement the Readable protocol. These devices are read at every data point and recorded into the resultant data event stream. This may include cameras or array detectors which produce channels that have additional axes in the resultant data.

Units support Many of the standard built-in Bluesky plans create linearly spaced arrays in the native units for each device. This proves limiting especially in the case of spectroscopy because different subfields often primarily work with different unit systems; A materials scientist often thinks in eV, while a molecular spectroscopist focusing on vibrational modes mostly thinks in wavenumbers (cm^{-1}). Making it easy for scientists to think in the more natural unit system for their experiment was a priority. This is even more critical because the units for energy and the units for wavelength are inversely proportional, so a linear spaced array of points in one unit is not the same as a linearly spaced array in an alternative unit.

A common pattern in the built-in Bluesky plans is to use a "variadic cycle" a repeating pattern of three to six arguments passed positionally where each set defines one axis of the scan. Where this pattern was used in the underlying plan, `wright-plans` adds the units as a string to the end of that cycle of arguments. The exception to this is `scan_nd_wp`, which is specified as a single `cycler`[130] object to define the positions. In this case, the units are specified as a dictionary mapping the device object to the unit string. If the unit is not provided in the dictionary or the unit is given as `None`, then the native units of the device in question are used.

Constants Constants allow setting of additional hardware during the course of a scan according to algebraic relationships. Specifically, Constants allow for positions to be set to linear combinations of other hardware positions. The simplest case of a Constant is one which is actually Constant throughout the scan. For example, setting a monochromator to zero nanometers or a delay to one picosecond delay. Another common example would be scanning two motors simultaneously, as in setting Delay 2 equal to the value of Delay 1. The most complicated common usage of Constants is to track the phase matching

condition of the experiment. For example, setting a monochromator to the sum of three light sources for a nonlinear spectroscopy experiment.

Constants are also unit aware, with the constraint that each term of the expression have compatible units, meaning that the units can be converted to one another. The algebra is performed on values in the Constant's specified units, so different resulting positions will occur for a Constant calculated in nanometers versus a Constant with an identical expression but units of wavenumbers.

Constants can be constructed using Python objects provided by `wright-plans` , as a list of `(motor, units, terms)` tuples, or as a dictionary mapping motors to `(units, terms)` tuples. The terms themselves can be created as a list of `ConstantTerm` objects or as a list of `(coefficient, variable)` tuples, with the variable being `None` for the constant term and a motor for a variable term. The motors specified by constant terms are not required to be motors otherwise specified by the plan, though if they are not, the motor positions specified by the plan are sensitive to motor positions at the start of the scan. It is required that there are no cycles for the dependency relationships in constants (i.e. m0 cannot be set to be 1*m1 while m1 is set to 2*m0, as that is an undefined position).

Per Step Per Step is a tool that can be used by advanced users to modify the behavior of a plan. Most standard users can safely ignore Per Step. It is provided in part because modifying Per Step is the mechanism by which `wright-plans` inserts the unit awareness and constant behavior, thus providing the same tool to downstream plan authors who may wish to build plans with additional behavior. Per Step is actually itself a Bluesky plan which takes the list of detectors, the motor positions specified for that step, and a cache of prior motor positions. Per Step then actually requests the motor motion and the measurement from the detectors.

`md` `md` is a dictionary of additional metadata provided for the plan. The plans themselves add metadata relating to the plan, but this can be overridden using the `md` keyword argument. The expectation of `md` is that the contents will be encoded to JSON.

`scan_wp`

Scan performs a one dimensional acquisition of linearly spaced points. If multiple motors are specified, each provides a start and stop position as well as their units, but there is only one keyword argument, `num`, which sets the number of points in the scan. This is referred to as the "inner product" of the two lists. Constants are allowed for scan, though the built-in functionality makes them largely superfluous. It can, however, be easier to think in terms of either end points or expressions for different experiments. Constants also have requirements regarding units that are not required for motors specified as endpoints.

The signature for `scan_wp` is:

```
def scan_wp(detectors, *args, num=None, constants=None, per_step=None, md=None):
```
(5.1)

Here, `args` is a variadic cycle of four values: A motor (Bluesky Setable), a start position, a stop position (both typed as appropriate for the motor, commonly floating point numbers), and a string specifying the units. `num` specifies the number of points. It is technically allowed, though discouraged, to pass the number of points positionally (i.e. as the last element of `args`), though explicitly passing using the keyword is preferred. `detectors`, `constants`, `per_step` and `md` are all as described above in 5.2.2.

```
## a simple scan
scan0 = scan_wp([det], m0, 0, 1, "mm", num=11)
# a scan with multiple motors specified
scan1 = scan_wp([det], m0, 0, 1, "mm", m1, 2, 3, "mm", num=11)
# equivalent scan using constant of "1*m0 + 1.0" instead of additional axis
scan2 = scan_wp([det], m0, 0, 1, "mm", num=11,
                constants=[[m1, "mm", [(1, m0), (1.0, None)]]])
```
(5.2)

`list_scan_wp`

List Scan is nearly equivalent to Scan, except that instead of specifying end points, an iterable list of positions is provided instead of endpoints and number of points. Since List Scan provides each of the points that are scanned, there is no requirement that points are linearly spaced. However, there is a requirement that each of the provided lists are the same length, as it is still doing an inner product.

This makes it a useful plan for logarithmically spaced points spanning multiple orders of magnitude[1]. Similarly, Constants are provided as an option, though they are technically subsumed by the built-in functionality of a List Scan.

The signature for `list_scan_wp` is:

```
def list_scan_wp(detectors, *args, constants=None, per_step=None, md=None):
```
(5.3)

Here, `args` is a three member variadic cycle consisting of a motor, a list of points (as opposed to the start and stop above), and a string specifying units for each motor. `detectors`, `constants`, `per_step` and `md` are all as described above in 5.2.2.

```
## a simple list scan, equivalent to a "normal" scan
scan0 = list_scan_wp([det], m0, np.linspace(0, 1, 11), "mm")
# a list scan with non-linearly spaced points
scan1 = list_scan_wp([det], m0, [0, 1,  2, 5, 10], "mm")
# a list scan with multiple motors specified, one pseudolog spaced
# and one linearly spaced
scan2 = list_scan_wp([det], m0, [0, 1, 2, 5, 10], "mm",
                     m1, [0, 1, 2, 3, 4], "mm")
```
(5.4)

`grid_scan_wp`

Grid Scan is the "outer product" counterpart to Scan. This plan provides a way of doing N-Dimensional scans, where each specified axis rather than scanning together scan against one another. Grid Scan is the workhorse plan for many data acquisition tasks. Similar to Scan, the axes are specified as endpoints plus number of points, however each axis now has its own number of points instead of having a single number for all motors. Constants are needed for Grid Scan because the built-in version does not have any mechanism to specify that two motors should move together rather than against each other. Additionally, Constants provide a way to specify that some condition, such as that the monochromator tracks the sum of two or three light sources.

[1]The built-in plans actually do provide a `log_scan` plan which acts as `scan`, where the endpoints and number of points are specified, however that is de-facto deprecated in favor of `list_scan_wp`, so it was not wrapped. The latter plan additionally allows similar ease to do symmetric log scans about zero.

The signature for `grid_scan_wp` is:

```
def grid_scan_wp(
    detectors, *args, constants=None, snake_axes=False, per_step=None, md=None
):
```
(5.5)

`detectors`, `constants`, `per_step` and `md` are all as described above in 5.2.2.

Here, `args` is a five member variadic cycle consisting of: a motor, start position, stop position, number of points, and units. `snake_axes` allows the user to specify if the trajectory of the scan proceeds to the start position after reaching the end of the slice (the default, False) or winds back and forth, reversing the order that points are collected for each slice (True). Even finer control of snaking behavior is available by passing a list of motors (which must be motors specified in the scan), which will cause only those axes to snake rather than all axes. The outermost (first) motor must not be included in that list, as it only ever is set to each position once, and thus has no opportunity to "snake" A Grid Scan with only one axis is equivalent to Scan, and the addition of constants means that most desired acquisitions can be accomplished with Grid Scan.

```
## a simple grid scan, equivalent to the simple scan above
scan0 = grid_scan_wp([det], m0, 0, 1, 11, "mm")
# a 2D scan
scan1 = grid_scan_wp([det], m0, 0, 1, 11, "mm",
                            m1, 2, 3, 21, "mm")
# scan representing a triple sum frequency experiment
# the constant is m3=m0+m1+m2, evaluated in wavenumbers
scan2 = grid_scan_wp([det], m0, 15000, 20000, "wn", 51,
                            m1, 1500, 2000, 21, "wn",
                            m2, 1500, 2000, 21, "wn",
                            constants=[[m3, "wn", [(1, m0), (1, m1), (1, m2)]]])
```
(5.6)

`list_grid_scan_wp`

List Grid Scan is to Grid Scan what List Scan is to Scan: rather than specifying endpoints and number of points, lists are provided for each axis. Similarly, this allows for scanning motor positions which are not linearly spaced, such as logarithmically spaced points. Since this is an outer product, there is no

requirement that the axes have the same number of points, and since the lists have lengths, there is no explicit requirement to specify the length at all. Constants provide the same benefits to List Grid Scan as they do to Grid Scan.

The signature for `list_grid_scan_wp` is:

```
def list_grid_scan_wp(
    detectors, *args, constants=None, snake_axes=False, per_step=None, md=None
):
```
(5.7)

Here, `args` is a three member variadic cycle: a motor, a list of positions, and the units. `detectors`, `constants`, `per_step` and `md` are all as described above in 5.2.2.

```
## a list grid scan with multiple motors specified, one pseudolog spaced and one
↪  linearly spaced
scan0 = list_grid_scan_wp([det], m0, [0, 1, 2, 5, 10], "mm",
                                 m1, [0, 1, 2], "mm")
```
(5.8)

`scan_nd_wp`

Scan ND is the mechanism by which all of the previous plans actually work. It is more complicated to use, and thus is rarely directly called, but it is also able to perform more unique scans than are provided by the parameterizations for more standard scans. It is highly recommended to consider if an equivalent acquisition could be done using one of the above plans prior to reaching for Scan ND; The reparameterized plans provide guardrails which make it harder to be incorrect or to get unexpected results. In fact, the behavior of Constants is also technically not required for Scan ND, though complicated examples involving multiple terms and unit conversion would be difficult to compute for a native Scan ND.

The signature for `scan_nd_wp` is:

```
def scan_nd_wp(
    detectors, cycler, *, axis_units=None, constants=None, per_step=None,
    ↪  md=None
):
```
(5.9)

Instead of specifying motor motion independently to the plan, this is done prior and a single `cycler`[130] object is handed to the Scan ND plan. This cycler can perform arbitrary inner and outer product computations to define a trajectory through N-dimensional motor space. As such, it can be used to perform scans similar to at least simple versions of Constants, mixing axes that scan together with those that scan against each other. The motors are the keys of the Cycler object. Since there is no "variadic cycle" for Scan ND, `wright-plans` unit support is provided via a separate keyword argument, `axis_units`, a dictionary mapping the motors as keys to strings, the unit to use for that motor. `detectors`, `constants`, `per_step` and `md` are all as described above in 5.2.2.

```
from cycler import cycler
cy = cycler(motor1, [1, 2, 3]) * cycler(motor2, [4, 5, 6])
scan_nd([sensor], cy, axis_units={motor1: "ps", motor2: "mm"})
```
(5.10)

`motortune`

The remaining plans that are provided by `wright-plans` enable specialized tuning routines designed for Optical Parametric Amplifiers (OPAs). The first, `motortune` is a flexible generic plan which can do a wide variety of tasks related to tuning OPAs. The remaining four plans are reparameterizations of `motortune` which make common acquisitions easier and more directly specified.

The signature for `motortune` is:

```
def motortune(detectors, opa, use_tune_points, motors, spectrometer=None, *,
↪  md=None):
```
(5.11)

`detectors` and `md` are as described above in 5.2.2. `opa` is the Bluesky device specifying an OPA. It is expected to have subdevices for the motors specified in `motors`. `use_tune_points` is a boolean which if true inserts an axis for the OPA as the outermost (slowest) axis of the scan at the points specified by the tuning curve. If motors are specified, then only the points specified by the relevant tunes are used, otherwise all points are used. `motors` is a dictionary mapping the *names* of the motors (as strings representing the attribute of the `opa` object) to a dictionary of parameters for each motor. The motor

parameters consist of the following:

- `method`: Must be the string "static" or "scan" the former meaning that the motor should be held constant for the duration of the scan, the latter meaning that the motor is an axis in the scan.
- `center`: Specifies the position for method "static" or the center of the axis for method "scan" ignored for method "scan" if `use_tune_points` is true, at which point center is relative to the tune point motor position.
- `width`: The width, in native motor units, to scan for method "scan".
- `npts`: The number of points for the motor axis in method "scan".

`spectrometer` is a dictionary of parameters which control a monochromator. `spectrometer` is optional as many tuning operations do not require spectral resolution or detectors are not mounted on a monochromator. Its keys consist of the following:

- `device`: The Bluesky device representing the spectrometer.
- `method`: Must be the string "static", "zero", "track", "set", "scan", or "none".
  - "static" means that the monochromator should be set to a specific position at the beginning of the scan, as specified by "center".
  - "zero" is equivalent to static, except that the position is always zero nanometers, provided as a convenience.
  - "track" introduces a Constant which sets the spectrometer setpoint to be equal to the OPA's setpoint.
  - "set" is equivalent to "static" for acquisitions where `use_tune_points` is false and "track" where `use_tune_points` is true.
  - "scan" introduces an axis for the monochromator to be scanned.
  - "none" is an alternate way of specifying no spectrometer at all.
- `center`: Specifies the position for method "static" or the center of the axis for method "scan", ignored for method "scan" if `use_tune_points` is true, at which point center is relative to the tune point.
- `width`: The width, in native motor units, to scan for method "scan".
- `npts`: The number of points for the motor axis in method "scan".

- `units`: specifies the units for method "static" or method "scan", if not provided, "nm" is assumed.

```
# Equivalent to an intensity scan
motortune(
    [det],
    w1,
    True,
    {"delay_2":{"method": "scan", "width": 2.0, "npts": 31}},
    {"device": mono, "method": "zero"}
)
# Equivalent to a setpoint scan
motortune(
    [det],
    w1,
    True,
    {"crystal_2":{"method": "scan", "width": 30, "npts": 31}},
    {"device": mono, "method": "scan", "width": -250, "npts": 51, "units": "wn"}
)
# Equivalent to a tune test scan
motortune([det],
    w1,
    True,
    {},
    {"device": mono, "method": "scan", "width": -250, "npts": 51, "units": "wn"}
)
# Equivalent to a holistic scan
motortune([array],
    w1,
    True,
    {"delay_1":{"method": "scan", "width": 1.0, "npts": 31}},
    {"device": mono, "method": "track"}
)

# Pure motorspace scan
motortune([det],
    w1,
    False,
    {"delay_2":{"method": "scan", "width": 2.0, "npts": 31}}
)
```

(5.12)

`run_intensity`

`run_intensity` is a plan designed to acquire data which will be processed with `attune.intensity`. Since it is known that such acquisitions will have exactly one motor which is scanned, the complexity of the `motors` argument to `motortune` is reduced to three required arguments: `motor`, `width`, and `npts`. The resultant `motortune` will always have `use_tune_points` set to True, and thus will always be a two

dimensional scan (before accounting for detector axes).

The signature for `run_intensity` is:

```
def run_intensity(detectors, opa, motor, width, npts, spectrometer, *, md=None):
```
(5.13)

`spectrometer` is passed directly to `motortune`, and thus must be specified in the same manner.

```
run_intensity(
    [det],
    w1,
    "delay_2",
    2.0,
    31,
    {"device": mono, "method": "zero"}
)
```
(5.14)

`run_setpoint`

The actual implementation of `run_setpoint` is nearly identical to `run_intensity`, as the scans required are exactly the same as far as motors are concerned. The difference is that `run_setpoint` is expected to have a spectral axis, as that is required for processing with `attune.setpoint`. This spectral axis may originate from either an array detector or by scanning the spectrometer.

The signature for `run_setpoint` is:

```
def run_setpoint(detectors, opa, motor, width, npts, spectrometer, *, md=None):
```
(5.15)

`spectrometer` is passed directly to `motortune`, and thus must be specified in the same manner.

```
run_setpoint(
    [det],
    w1,
    "crystal_2",
    3.0,
    31,
    {"device": mono, "method": "scan", "width": -250, "npts": 51, "units": "wn"}
)
```
(5.16)

`run_tune_test`

`run_tune_test` is the simplest of the tuning plans, with no motors specified. It is designed to be processed by `attune.tune_test`, and thus a spectral axis is expected and as with `run_setpoint` may originate from either an array detector or scanning a spectrometer.

The signature for `run_tune_test` is:

```
def run_tune_test(detectors, opa, spectrometer, *, md=None):
```
(5.17)

`spectrometer` is passed directly to `motortune`, and thus must be specified in the same manner.

```
run_tune_test(
    [det],
    w1,
    {"device": mono, "method": "scan", "width": -250, "npts": 51, "units": "wn"}
)
```
(5.18)

`run_holistic`

Finally, `run_holistic` defines an acquisition designed to be processed with `attune.holistic`. Here, one of the motors, `motor0` is not actually scanned, but rather inferred from the tune points themselves. It is specified so that the metadata can reflect the logical axes of the scan. The second motor, `motor1`, is specified in the same manner as the other derived tuning plans: as a name, width, and number of points.

The signature for `run_holistic` is:

```
def run_holistic(detectors, opa, motor0, motor1, width, npts, spectrometer, *,
↪  md=None):
```
(5.19)

`spectrometer` is passed directly to `motortune`, and thus must be specified in the same manner.

```
run_holistic(
    [array],
    w1,
    "crystal_1",
    "delay_1",
    1.0,
    31,
    {"device": mono, "method": "track"}
)
```
(5.20)

5.2.3 `bluesky-in-a-box`: Background Services to Collect Scientific Data

`bluesky-in-a-box` runs all of the services using Docker[131][132]. Specifically, `bluesky-in-a-box` uses Docker Compose[133] to simultaneously start and stop all services, accounting for dependencies between them. A summary of the architecture of `bluesky-in-a-box` is provided in Figure 5.3. There are six docker containers: Redis, Mongo, WT5, zmq-proxy, hw-proxy and re-manager. These all transmit data to one another, and with additional programs running outside of Docker, as signified by the arrows between boxes. A summary of the exposed TCP Ports is provided in Table 5.1.

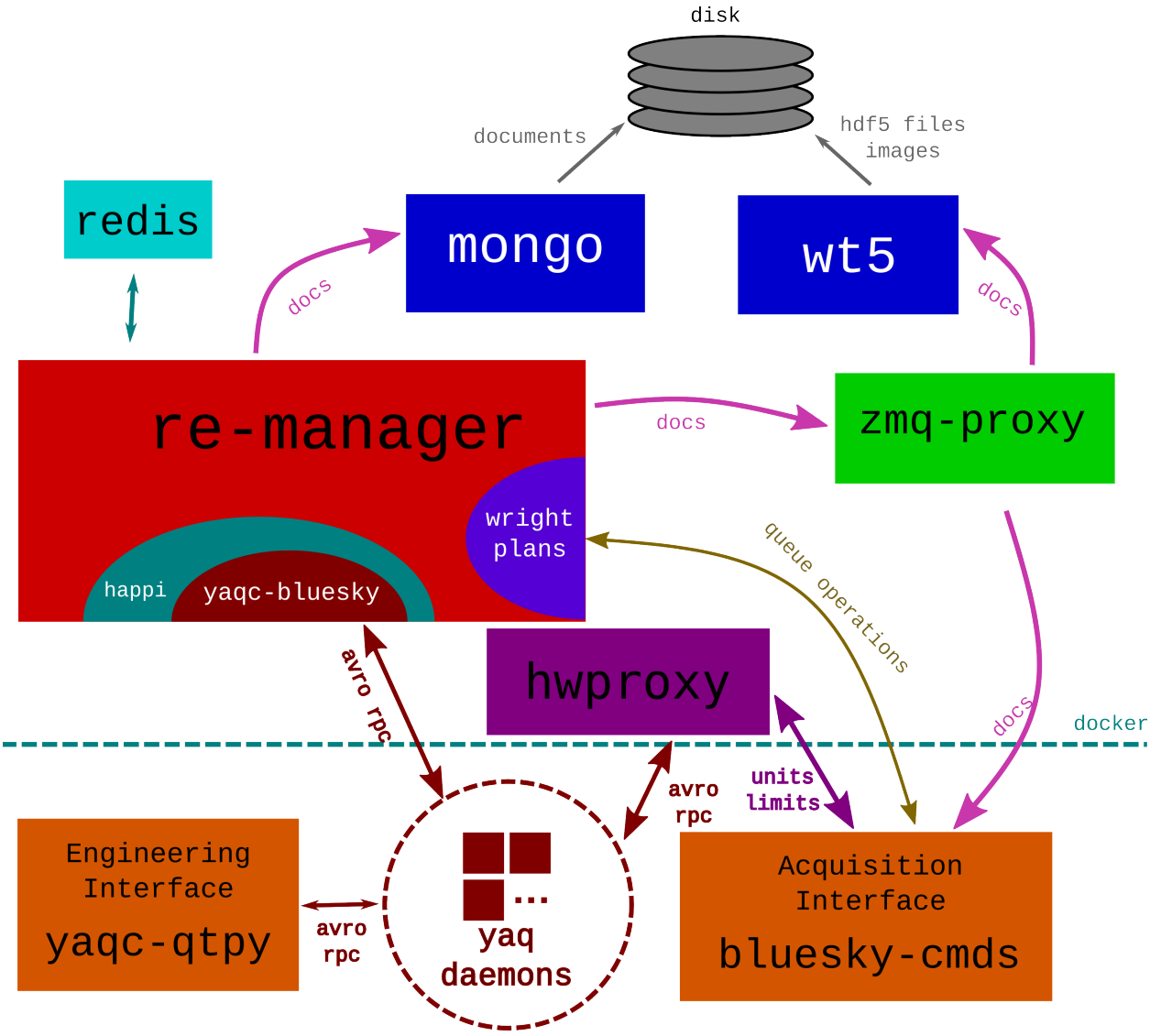

Figure 5.3: A summary of the parts of the `bluesky-in-a-box` and the flow of data between them. The parts above the line are all run inside of docker, and can be brought online and offline simultaneously. The parts below the line are run on the host system and represent the more direct hardware access and the graphical programs to used to interact with the system.

Port	Protocol	Content
5567	zmq	Event Model Documents publishing
5568	zmq	Event Model Documents subscribing
6379	redis	Normally only used internally, storage of the queue and history
27017	mongo	Event Model Documents storage and access via databroker
60615	JSON/zmq	re-manager control and query
60620	JSON/zmq	hw-proxy queries
60625	JSON/zmq	re-manager log entry subscribing

Table 5.1: Summary of open ports in `bluesky-in-a-box` .

Installation

Set up hardware virtualization Docker is a technology that allows running services in a controlled environment that is isolated from the programs on the host machine. To do this, it takes advantage of hardware options on modern Central Processing Units (CPUs). The hardware options are available on most CPUs, but are often turned off by default and must be turned on in the BIOS settings. The name and location of this setting will depend on the make and model of the hardware, but often are identified by options with names like "KVM" or "Intel Virtualization Technology". Technically, this should only be required for Windows and MacOS installations.

Set up WSL (Windows only) The docker containers for `bluesky-in-a-box` define a Linux environment for each service. Microsoft has invested in the Windows Subsystem for Linux (WSL)[134] to make running Linux environments on Windows machines efficient. To install on Windows 10 or above, you simply need to open a command prompt as an administrator (cmd.exe or powershell) and run `wsl --install`. The install process will prompt you to reboot the machine. It is recommended to install a version of Linux into WSL explicitly, by running `wsl --install -d debian` (or whatever distribution is your preference).

Install Docker On Windows and MacOS, the preferred method of installing Docker is to install Docker Desktop[135]. This is a standard installer wizard which will guide you through the steps to install. Once installed, you should toggle a handful of settings. Most notably, under the "Resources → WSL Integration", it should be enabled for all distributions installed. Additionally "Docker Compose V2" should be enabled. It is also recommended to consider your personal preference and purpose of using docker to determine if starting docker at login is desirable, if you want weekly tips for using Docker, or if you wish to send usage stats to Docker. Note that Docker Desktop's license terms require a paid subscription for commercial use, however academic use continues to be free.

On Linux, mostly Docker is installed via the system package manager. For a Debian or Ubuntu based system this would be:

```
sudo apt install docker.io
sudo apt install docker-compose
```
(5.21)

It is recommended to add your user account to the `docker` group to avoid needing to use `sudo` to perform docker operations:

```
sudo usermod -aG docker $USER
```
(5.22)

Configure available `yaq` hardware This section assumes that the system has `yaq` hardware running and set up to be listed by `yaqd list`. If this is a development machine, it is recommended to install 5.2.6 prior to proceeding.

`bluesky-in-a-box` uses `happi`[136], a project created by the Stanford Linear Accelerator Center (SLAC), to communicate what hardware is available to the `re-manager` and `hwproxy`. It provides a standard way of declaring available hardware and how to create a Bluesky-compatible Python object for each. Using `happi` allows the possibility of easily and transparently using hardware supported by additional control layers beyond `yaq`.

`happi` is a Python package that can be installed with either `conda install happi` or `pip install happi`. This will install a Command Line Interface (CLI) Application to manage a database (stored as a JSON file) of available hardware. In addition, for `happi` to be able to work with `yaq` hardware you must `conda install yaqc-bluesky` or `pip install yaqc-bluesky`.

To get started, set up a configuration file for `happi`. On Windows, start by creating a folder `~\AppData\Local\happi\happi` Inside of this folder, create a file, `happi.ini` with the following contents (replacing `<USERNAME>` with the username that is appropriate for your system):

```
[DEFAULT]
path = C:\Users\<USERNAME>\AppData\Local\happi\happi\happidb.json
```
(5.23)

Follow this by running the following in a command prompt:

```
setx /s %COMPUTERNAME% /u %USERNAME% HAPPI_CFG
↪  %LOCALAPPDATA%/happi/happi/happi.ini
```
(5.24)

This sets up an environment variable which is persistent between sessions that tells happi where to load and save changes.

The Linux equivalent would be to create `~/.config/happi.ini` with the contents of path pointing to `~/.local/share/happi/happidb.json`.

Once set up, you can add all `yaq` hardware by doing:

```
yaqd list --format happi | happi update -
```
(5.25)

This exports all of the daemons that are recognized by `yaqd-control` into the happi database. If hardware is added to the system (or information like the host or port are updated) then the same line will update happi again.

Configure data folder You should create a data folder where `bluesky-in-a-box` will place the generated `wt5` files. The exact location does not particular matter, but `~/bluesky-cmds-data` is what will be used in this document.

Download `bluesky-in-a-box` Docker configuration To download, first clone the git repository for `bluesky-in-a-box` :

```
git clone https://github.com/wright-group/bluesky-in-a-box
```
(5.26)

There are a few options for the Docker Compose files, a template for the file is provided in the root of the `bluesky-in-a-box` repository. Copy the template file, `.env-example` to `.env`, then edit it. `HAPPI_DB_PATH` should be set to the full path of `happidb.json` as specified in `happi.ini` file above. `WT5_DATA_PATH` should be set to the full path of the data folder specified above. Finally, `TZ` should be set to the appropriate timezone where it is running, for example `America/Chicago`.

Start and update containers Start the docker engine, either by opening Docker Desktop (Windows) or by enabling the background docker service (Linux). Navigate to the `bluesky-in-a-box` folder in a command prompt. If needed, pull new versions or switch to appropriate branches that contain the version of `bluesky-in-a-box` that you need. To build for the first time or to perform a standard update, run the following:

```
docker compose up --build
```
(5.27)

That will suffice for most cases, however if you want to force a full rebuild, you can use:

```
docker compose build --no-cache
```
(5.28)

This is most useful as a development tool to ensure that you are not accidentally relying on the state of your local system in ways that would not work on other machines.

Once the containers are built, you can start them using Docker Desktop by pressing the start button for `bluesky-in-a-box` in the "Containers/Apps" tab. This will start the pre-built containers without rebuilding them. Rebuilding using the CLI is needed whenever the containers themselves are changed, most commonly for bumping the versions of installed software.

`re-manager`

The RE Manager is the beating heart of the `bluesky-in-a-box` system. It is the service which accepts parameterized plan descriptions, organizes them into a queue, allows manipulation of the queue itself, and, most importantly, actually conducts the experiment. The RE Manager starts a Bluesky Run Engine (hence the name) and consumes the plans from the queue. These plans encode the steps for taking data: moving motors, triggering sensors, and emitting Event Model documents.

The RE Manager is responsible for connecting to the devices which are listed by the HAPPI database, providing its own list of available devices (which filters out those that are offline or are subdevices of other hardware such as OPA motors). At present, `bluesky-in-a-box` expects that all hardware

specified is `yaq` hardware, though support for additional interface layers requires only that they are specified in HAPPI and that the necessary dependencies get added to the `re-manager` container. It is also responsible for providing a list of available plans. For `bluesky-in-a-box` , all of the `wright-plans` are available, as well as the built-in Bluesky `count`, `sleep`, and `mv` plans. `count` generates a time series of readings, separated by a wait period. `sleep` simply waits for a specified period of time, given in seconds. `mv` allows for setting of hardware (to positions specified in their native units) in a manner which can be enqueued. All of the plans which generate data are wrapped with the Bluesky `baseline` decorator, which reads *all* movable hardware once before and once after the plan itself is consumed. This allows for a full record of the state of the system without recording all hardware for every data point, even when it not told to move for a given plan.

The RE Manager container provides a few ways of interacting, which are largely based on intefaces over a ZeroMQ[137] message system. ZeroMQ (abbreviated ZMQ or 0MQ) provides a standard way of either doing request/reply Remote Procedure Calls (RPCs) or publish/subscribe (pub/sub) data flows where one end provides a stream of data and the other is expecting that stream of data. The former, RPC style requests, is used to enqueue plans, interact with the queue itself, and to query the state of the RE Manager. The latter, pub/sub style requests, is used to publish the Event Model documents as well as (separately) publishing the console output of the RE Manager so that it can be displayed in client applications. These ZMQ interfaces are considered trusted interfaces, and as such do no authentication of users. If such authentication is needed, it can be provided by additional tools by the authors of `bluesky-queueserver` which wrap the ZMQ interface in a FastAPI[138] interface served over TCP, with authentication done by that service. This is recommended in the future if remote access is desired to instruments running `bluesky-in-a-box` , however initial development is focused on running a self contained system which provides access primarily to the direct machine running the instrument.

`zmq-proxy`

`zmq-proxy` is a small program provide by Bluesky which simply routes Event Model documents that it reads from on port and writes them to any number of subscribers on a second port. The same purpose in larger systems where caching, latency, and network reliability are important may be achieved

by an industry standard service like Apache Kafka[139]. Because `bluesky-in-a-box` is running on a single machine, network reliability and latency are not anticipated problem and we do not currently require caching, so the simpler program is used. `zmq-proxy` provides a server for each end of the communication, which allows multiple clients to act as sources or sinks of the documents which are being shuttled. In practice, usually only one service, the RE Manager, should ordinarily be the source of the documents. However, this does provide an easy way for testing or special purpose scripts to serve as a source.

`WT5`

The WT5 container consumes the Event Model Document stream from the `zmq-proxy` container and creates WrightTools Data files. These files are stored in a folder shared with the host platform, such that they can be browsed and opened using platform native tools. In addition, the WT5 container writes a number of text files into the data folder and generates images for many scans. These include the "start" "descriptor" and "stop" documents from the event model for each Bluesky Run, written as JSON text files. Additionally each generated WT5 file also writes the output of `data.print_tree(verbose=True)` to a file so that users can quickly see information such as the shape and axes of the data file without actually opening the file.

One and Two dimensional scans produce the "quick" plots provided by WrightTools. Larger scans have caused significant memory problems when attempting to plot many slices, and so have been disabled.

`mongo`

MongoDB[140] is an industry standard database for storing JSON-like documents. Mongo is widely used by the upstream authors of Bluesky at various National Labs as the immediate store of data being recorded from instruments. As such, there are pre-existing tools in the ecosystem that pair well with a Mongo database. The Mongo Docker container is provided to allow exploration of this ecosystem of tools with the data collected by `bluesky-in-a-box` . It also provides a back-up of the data should the WT5 data writer fail. At present, the Mongo database does not serve as a long term archival data

back-up, but rather as a short term fail-safe. This decision is not one of fundamental design, but rather a decision of allocation of development time. It is quick and easy to add a Mongo container to the Docker system and route the data from the RE Manager to it, ensuring it is accessible short term. It is a more complex task to ensure that the data gets written in a manner appropriate for archival storage, and there has been no need for it at this time.

`redis`

Redis[141] is an industry standard service for in-memory key-value data store. It is used by the RE Manager container to track the state of the queue of plans. Redis is required for the operation of the RE Manager, and thus must be running. However, there is little to no direct interaction with redis, so unless you are actively developing `bluesky-queueserver`, it can largely be ignored.

`hwproxy`

`bluesky-hwproxy`[142] is a small project started for inclusion in `bluesky-in-a-box` . Its primary purpose is to provide read-only access to the hardware which is available to the RE Manager. This allows for rich graphical programs like `bluesky-cmds` to query information about hardware such as limits and units. `hwproxy` wraps the device loading portions of `bluesky-queueserver` and exposes the standard Bluesky Protocol methods over a ZMQ interface very similar to that used by the RE Manager itself. This allows any tool that can communicate with the RE Manager to communicate with `hwproxy` just as easily.

Troubleshooting

Docker as a technology for running `bluesky-in-a-box` was chosen primarily to make standard usage as easy as possible. It allows for all of the connected services to be brought online and offline quickly and easily, encoding all of their dependent relationships on each other. However, the same factors that make Docker a good choice for standard operation mean that a few extra steps are needed when troubleshooting or debugging. In particular, Docker is designed to provide a consistent environment for

all of the services to run, with static code and dependencies. Troubleshooting is a task which necessitates rapid updating and a tight feedback loop.

There are some particular approaches that are useful when debugging `bluesky-in-a-box` in particular (and Docker containers more generally). The first step is to determine *where* the origin of the problem exists. This is not as trivial as it may seem at first. Generally, it is a good strategy to start by carefully reading the logs provided by the Docker containers. The two most relevant containers will be the RE Manager and the WT5 containers. The other containers are all fairly standard tools that are not nearly as customized and thus are not likely to be the problem. Most of the time, the container which has an error message is a good candidate for where the problem exists, but this is not always the case. Some problems may exist in the network itself which connects the containers to each other and to the host system. These tend to be relatively rare, but can also be some of the trickiest to track down and resolve. Others may present as errors down stream but the root cause is the input to that container. For example, faulty parameters may cause a plan to fail during execution. Such a failure may be expected, given the inputs, so this example means that either user training or user interfaces could be improved. Alternatively, if the metadata is malformed that may allow the actual data acquisition to proceed as expected, but cause the WT5 data writer to fail. In this instance, the root cause would be the Bluesky plan, which is a dependency of the RE Manager, though the error would present in the WT5 container.

For smaller tests, particularly those involving testing of the network connections, Docker allows opening a command shell which runs inside of the container. If using Docker Desktop, starting a command shell inside of one of the Docker containers is as easy as pressing a button in the graphical application. On any platform, you can use the Docker command line to open a shell in the Docker container:

```
docker exec -it <container name> bash
```
(5.29)

Note that the container name here is as listed by `docker ps`, not the name in the Docker Compose file. For `bluesky-in-a-box` , this is most useful for information gathering tasks such as answering the question "Can the RE Manager container communicate with the yaq daemon running on the host machine?" or "Can my WT5 container communicate with the ZMQ Proxy container?".

If the problem exists with the code that is in the `bluesky-in-a-box` repository (mostly the startup scripts, configuration, and the WT5 data writing code) then a more traditional editing the code and running the containers is reasonably achievable. Do `docker compose up --build` to run the Docker containers. Without doing this, the old version will run instead. This makes for a reasonable, if a little slower than ideal, development feedback loop.

If, instead, the problem exists in your dependencies, that makes developing even more indirect. There is not a particularly good way to build the Docker containers pointing to local directories with code changes. Thus the procedure is instead to push the code changes to a remotely accessible Git repository and install Python dependencies from that Git repository. Note, however, that while you *can* point the Git installation in `requirements.txt` to the branch that you create, this is not fully recommended because subsequent updates to the branch will not cause the Docker container to rebuild. Docker will only redo a step in the installation if it can tell that the step requires it. Thus it will only redo the step that installs the Python dependencies if it sees a change to what it is asked to install. Since the requested branch is the same, the Docker build system determines that it is the same version being requested and thus will use the previously built version. Instead, it is recommended to use the SHA hash pointing to a particular commit instead, and updating it every time the code is pushed to the remote repository.

An example of a `requirements.txt` that points to a Git repository:

```
bluesky==1.8.3
#WrightTools==3.4.4
pyzmq==22.3.0
numpy==1.22.3
toolz==0.11.2
git+https://github.com/wright-group/WrightTools@abcdef12
```
(5.30)

The last line adds a particular Git version of WrightTools which takes the place the commented out version specified earlier in the requirements file.

Finally, and perhaps most powerfully, the modularity of the Docker system can be used to run parts of the system outside of docker itself, thus enabling more traditional development and feedback cycles. Any of the Docker containers can be independently halted and superseded by a version of the service running

outside of Docker, though clients of that service may need to be reconfigured to point to the correct external service. While it can be used if there are problems discovered with the MongoDB, Redis, or ZMQ Proxy containers, for `bluesky-in-a-box` this is most useful to perform tests using a smaller Run Engine running outside of the RE Manager container all together. B.5.1 provides an example of a script that can be run directly on the host machine, performing data acquisitions which are still processed by the WT5 data writing pipeline (via the ZMQ Proxy container). This particular script does not require *any* change to the configuration of `bluesky-in-a-box` . In fact, even the RE Manager container can be running, though it should not be actively acquiring data, as downstream tools such as the WT5 writer and live plotting are likely to behave poorly if there are two simultaneous acquisitions happening. This script allows for a much smaller and easier to troubleshoot system to do the actual orchestration of the acquisition. It removes some of the complexities of of the RE Manager container, allowing for more direct access to the Run Engine. This comes, of course, at the cost of not having a queue or the user interfaces built to interact with the RE Manager.

5.2.4 `qserver`: Command Line Front-end to `bluesky-queueserver`

`qserver` is a command line program for interacting with `bluesky-queueserver`. It is provided by the project itself, and serves as a valuable tool for both troubleshooting and as an "escape hatch" for things that more narrowly focussed interfaces do not make possible.

Installation

`qserver` is included with the installation of the Python package `bluesky-queueserver`. As such it is installed via:

```
conda install bluesky-queueserver
```
(5.31)

or:

```
pip install bluesky-queueserver
```
(5.32)

Usage

The most common use for `qserver` is to simply query the server for its current state:

```
$ qserver status
Arguments: ['status']
17:29:47 - MESSAGE:
{'devices_allowed_uid': '60b8e736-2b3e-4531-831f-d46dec9a477b',
 'devices_existing_uid': 'f203f778-5e9d-454d-9c8d-30a3c193c8f9',
 'items_in_history': 19,
 'items_in_queue': 0,
 'lock': {'environment': False, 'queue': False},
 'lock_info_uid': '581a37ac-d2d9-4967-b998-d044f1570d25',
 'manager_state': 'idle',
 'msg': 'RE Manager v0.0.16',
 'pause_pending': False,
 'plan_history_uid': '587f5b9f-ffe0-404b-be11-37b723c628ad',
 'plan_queue_mode': {'loop': False},
 'plan_queue_uid': '19b7952c-9c2f-4bd5-a621-6d5750912916',
 'plans_allowed_uid': '0650be18-94d3-4966-969c-a0b1264a2473',
 'plans_existing_uid': '08851d8e-8156-4407-9f60-6d7e7979bc10',
 'queue_stop_pending': False,
 're_state': 'idle',
 'run_list_uid': '2a9e07dc-f8c7-4c39-a9b6-c90f6c9b9833',
 'running_item_uid': None,
 'task_results_uid': 'efc52454-d902-4358-9470-80de6a7416db',
 'worker_background_tasks': 0,
 'worker_environment_exists': True,
 'worker_environment_state': 'idle'}
```

(5.33)

This gives a fair amount of helpful information, such as the size of the queue and history, and the state of various parts of the queue server itself.

Additional examples of how to use `qserver` are provided by the built-in help text:

```
$ qserver help
```

(5.34)

This will print an extensive list of available uses for the program, including adding items to the queue and controlling the running state of the queue. All of the interactions that are possible as a client of the QueueServer are possible through this interface, though many of them are easier to interact with in a more specialized interface such as the graphical one detailed in the next section.

5.2.5 `bluesky-cmds` : Graphical Front-end to `bluesky-queueserver`

`bluesky-cmds` is a graphical application to interact with a running Bluesky Queueserver, such as that provided by `bluesky-in-a-box` . While everything *can* be accomplished using the command line tool above, that is not the most ergonomic tool for users. The CLI requires that users type out exactly the correct command, including properly formatting JSON where it is required. As a graphical app, `bluesky-cmds` encodes much of the tedious structure of requests to the Queueserver in the user interface itself, thus minimizing potential for simple typographical errors.

`bluesky-cmds` inherits much of its style, layout, and capabilities from the prior graphical data acquisition apps, `yaqc-cmds`and PyCMDS. It does, however, remove all of the direct interaction with hardware that the older programs had in order to focus solely on being the best application to manipulate the RE Manager queue. This functionality is provided by alternative applications such as `yaqc-qtpy`. Additionally, the actual code to perform acquisitions and store data is no longer part of the same application, replaced by requests of the Queueserver. This means that the graphical app can be closed (intentionally or unintentionally) and the acquisition continues.

Installation

`bluesky-cmds` is included with the installation of the Python package `bluesky-cmds` . However, the package is not yet available on conda, so conda systems should install the dependencies via conda first. As such it is installed via:

```
conda install bluesky-queueserver-api appdirs click pyqtgraph pyside2 qtpy toml
↪  toolz wrighttools sympy bluesky-widgets
pip install bluesky-hwproxy bluesky-cmds --no-deps
```
(5.35)

or:

```
pip install bluesky-cmds
```
(5.36)

or, for development purposes (conda users should still install dependencies as above):

```
git clone https://github.com/wright-group/bluesky-cmds
cd bluesky-cmds
flit install --pth-file
```
(5.37)

Configuration

If running locally, then the default configuration options should suffice. To populate or edit the configuration file you can execute:

```
bluesky-cmds edit-config
```
(5.38)

This will prompt you to populate the file with the default template if it does not already exist, and open the configuration file in a text editor.

The default template, appropriate for connecting to a `bluesky-in-a-box` instance running on the local machine is shown below:

```
[bluesky]
re-manager = "tcp://localhost:60615"
re-info = "tcp://localhost:60625"
hwproxy = "tcp://localhost:60620"
zmq-proxy = "localhost:5568"
```
(5.39)

To run remotely, the host and port for each connected service can change as needed.

Usage

Starting `bluesky-cmds` is done by running the following command:

```
bluesky-cmds
```
(5.40)

In the future, planned integrations with the operating systems (Windows, Linux, MacOS) will allow `bluesky-cmds` to appear as an ordinary application that can be started from application menus.

When the application opens, it communicates with the RE Manager, retrieving lists of available devices and plans. Additionally the application opens the Run Engine worker environment. The startup behavior is repeated whenever the application reconnects to the RE Manager (e.g. when the RE Manager is restarted)

The application window has a progress bar across the top, and a tab view which provides the main user interaction. The tab view has three tabs: Queue, Plot, and Logs. The progress bar gives an indication of the progress of a running acquisition. For the progress bar to work, the expected number of points must be known at the beginning of the scan and the application must be open when the acquisition begins. The progress bar provides an elapsed time counter on the left hand side and an estimation of remaining time on the right hand side. The bar is green when the acquisition starts or completes successfully, and turns red if the acquisition fails.

The Queue Tab

The Queue tab is the default tab when the application starts up. Figure 5.4 shows the Queue tab. Along the left hand side, there is a pane to control the queue, allowing enqueueing of plans and starting the queue. On the right, there is a table which displays the queue and the history.

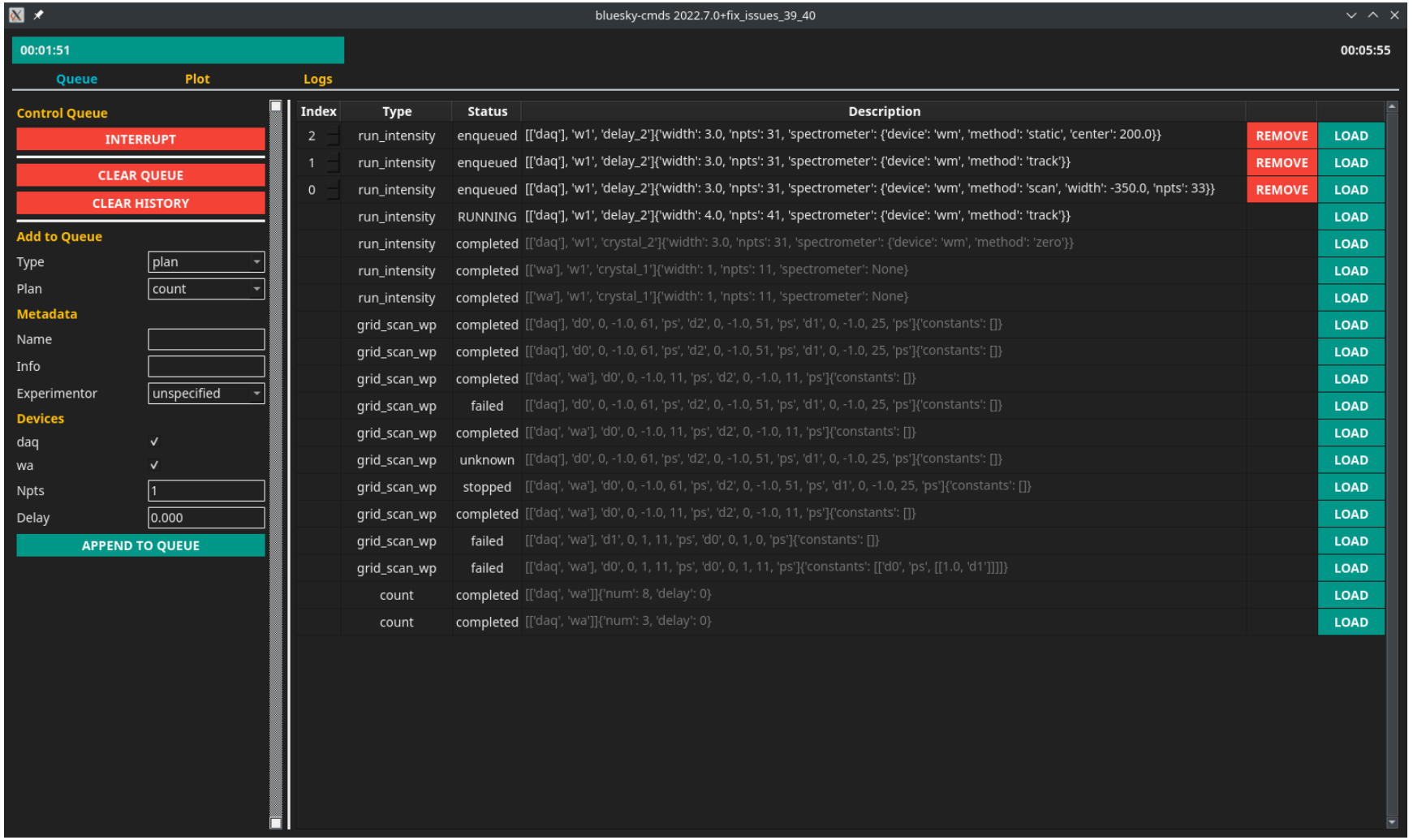

Figure 5.4: `bluesky-cmds` with the queue tab open.

To begin an enqueued scan, click the teal "Start Queue" button. While the queue is running, this button turns into a red "Interrupt" button. When pressed, the RE Manager will pause execution of the plan and a dialog window presents options of how to proceed: "Resume", "Stop after plan", "Skip", and "Stop now". Figure 5.5 shows the interruption workflow. "Resume" will cause the paused plan to proceed from where it left off. "Stop after plan" will instruct the queue to resume the current plan, but not proceed to the next enqueued plan. "Skip" instructs the RE Manager to halt execution of the current plan, marking the plan as successful and not re-enqueuing it. "Stop Now" instructs the RE Manager to halt execution of the current plan, but mark the plan as unsuccessful and re-enqueue it to run again.

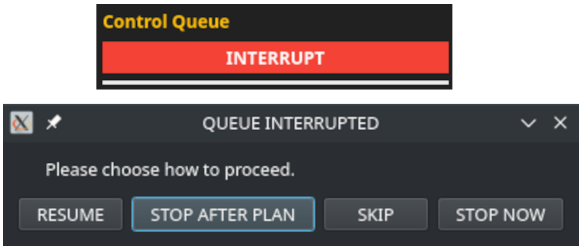

Figure 5.5: When an acquisition is in progress, the Queue Start button becomes an Interrupt button which pauses the acquisition and presents options for how to proceed.

The queue and history can be cleared by clicking the appropriate button in the left hand pane of `bluesky-cmds` . Each of these will open a confirmation dialog as shown in Figure 5.6 to prevent accidentally removing enqueued or recently run plans. Doing so is rarely absolutely necessary, though it is easier than removing enqueued items individually and the user interface can be slower than desired if there are many entries in the queue and history.

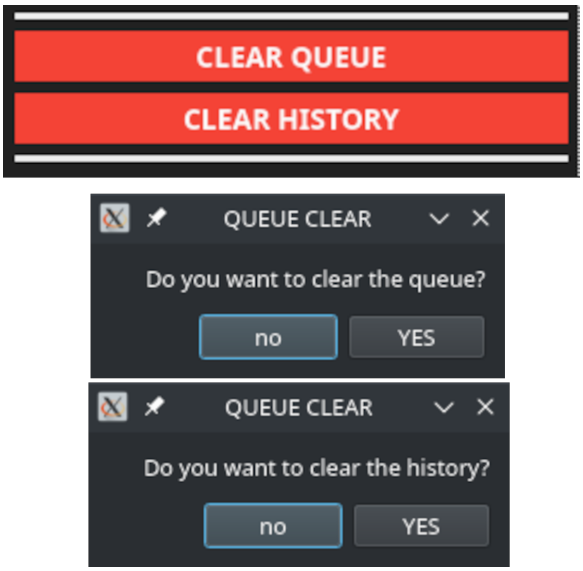

Figure 5.6: The buttons for clearing the queue and the history and the pop-up windows that open for each.

Below the queue and history clearing controls, there is a drop down box with three options: "plan", "instruction", and "preset". This controls the remainder of the left hand bar to allow adding various items to the queue.

Enqueueing a plan The primary way to enqueue a plan in `bluesky-cmds` is to use the "plan" option of the "Type" selector. The next selector provided is the "Plan" selector, which provides a list of allowed plans. Each plan then has its own interface for setting the arguments to that specific plan. These interfaces are currently collated by hand and provided by `bluesky-cmds` , though in the future may be able to be built automatically by information provide by the RE Manager. There is a fall-back option if no custom interface is provided where JSON arguments and keyword arguments can be provided directly. Figure 5.7 shows the interface for the `grid_scan_wp` plan.

There are common patterns used in building these interfaces. For instance, all of the plans with variadic cycles have buttons to add and remove sets of inputs that must be provided together. There are also reusable interface elements for providing metadata, detectors, or constants.

Plans are appended by clicking the teal button at the bottom of the plan interface.

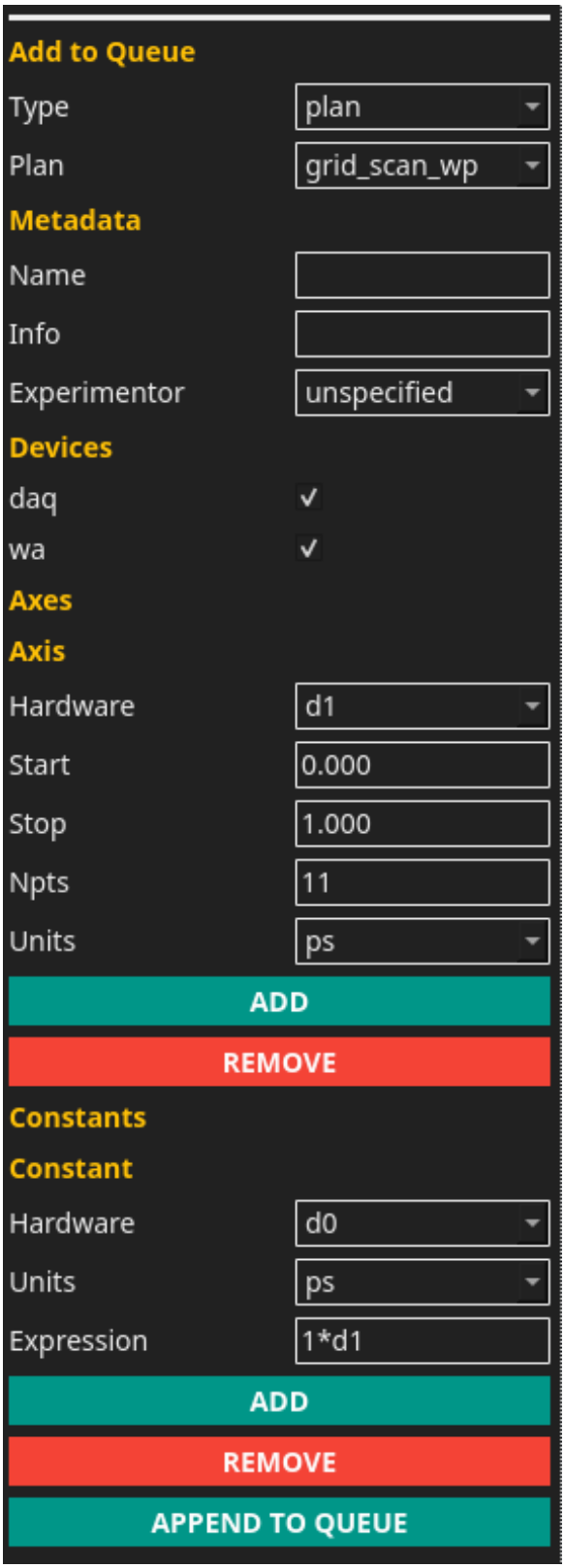

Figure 5.7: The user interface for enqueueing a grid scan plan.

Enqueueing an instruction Instructions are items that can be enqueued that modify the behavior of the queue itself. At present, there is only a single option for instructions to append to the queue: Queue Stop. There are no parameters for Queue Stop, so it appears as a simple button which when pressed enqueues a Queue Stop at the end of the active queue. This instructs the queue to return to an idle state rather than proceeding to the next enqueued plan, requiring the user to explicitly restart the queue to continue. This can be useful if there is some manual user interaction required, but the next experimental steps are known so the plans can be enqueued before that interaction can occur. Figure 5.8 shows the instruction pane in the user interface.

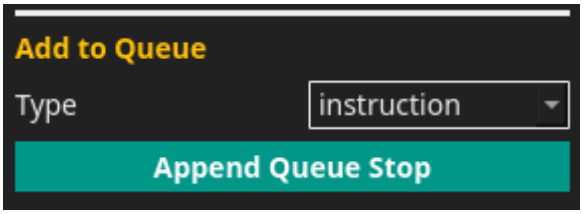

Figure 5.8: The "instruction" pane for appending to the queue.

Enqueueing a preset A "preset" is simply an ordered list of plans (or instructions) which can saved and enqueued sequentially. This is useful if there are repetitive tasks such as tuning operations or running the same battery of scans on different samples. Under the hood, presets are files stored in a configuration folder where the file name identifies the preset name and the contents are JSON objects, separated by lines, which represent a series of plans or instructions. The preset system is functionality added by `bluesky-cmds` , it is not built-in to the RE Manager directly. When a preset is selected to be appended to the queue, `bluesky-cmds` opens the preset file and reads each line and appends each entry to the queue.

Figure 5.9 shows the preset enqueueing pane. The second drop-down box allows selecting one of the existing preset options. The teal "Append Preset Plans" button enqueues all of the plans from the preset. The yellow "Edit Preset Plans" button opens a text editor to edit the JSON file directly. This is an advanced feature, but provides the power and flexibility to those who need it. While it can be used to completely add new entries to the preset, it is most useful for smaller tweaks. Changing the value of parameters, order of enqueuing, or removing plans from presets are all relatively simple tasks in the text editor view.

An example of a preset file:

```
{"item_type": "plan", "name": "count", "args": [["daq"], 3]}
{"item_type": "plan", "name": "count", "args": [["daq"]], "kwargs": {"num": 5,
↪  "delay": 0}, "meta": {"Name": "", "Info": "", "Experimenter": "Kyle"}}
{"item_type": "plan", "name": "count", "args": [["daq"], 5]}
{"item_type": "plan", "name": "mv", "args": ["d0", 0, "d1", 0]}
{"item_type": "instruction", "name": "queue_stop"}
```

(5.41)

There is presently no built in user interface to remove a preset entirely. To do so, you must delete the file from the folder of presets, which is located in `~/.local/share/bluesky-cmds/presets` on Linux and `~\AppData\Local\bluesky-cmds\bluesky-cmds\presets` on Windows.

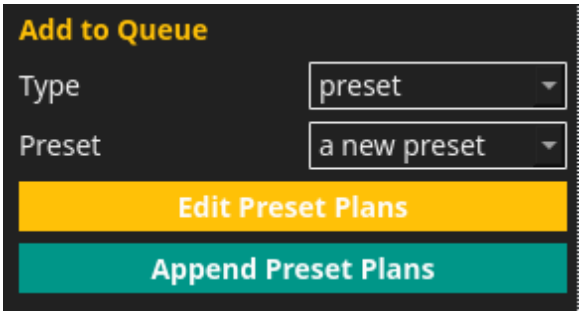

Figure 5.9: The "preset" pane for appending to the queue.

To add to a preset, right click on the description field of an entry in the queue or history, and select "Append to preset...". This will open a dialog box with a selector for existing presets, which if selected the plan will be appended to the end of the preset. If you select "New Preset...", then an additional dialog will open, where a name for a new preset can be provided. This workflow is shown in Figure 5.10.

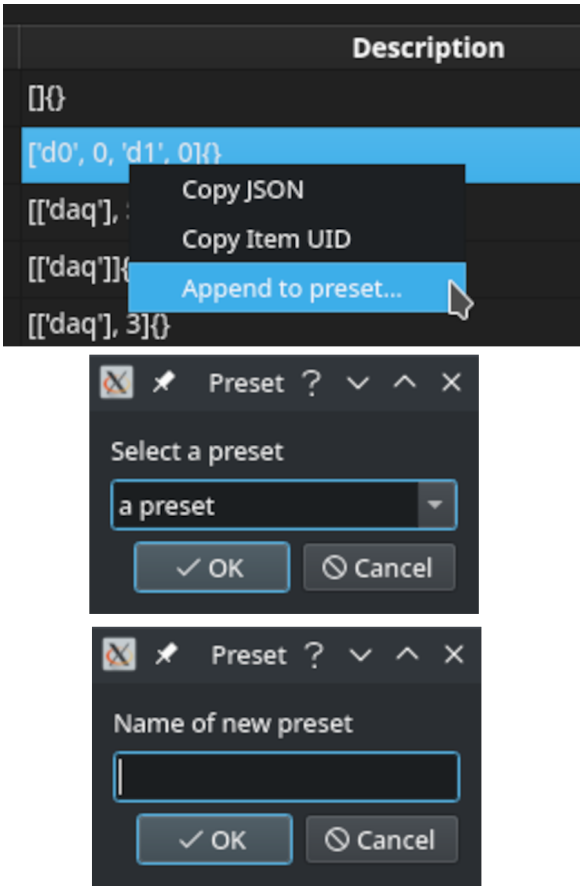

Figure 5.10: The workflow for appending to presets.

In the main body of the window, a table represents the current state of the queue and history. More recently enqueued items appear towards the top of the table, older items towards the bottom. The items waiting in queue, thus, appear at the top, and the items in the history appear at the bottom. The currently running item (if it exists) appears at the boundary between the queue and the history. The enqueued and running items have descriptions shown in bright white, while the completed items appear with gray descriptions. The right most column indicates the position in the queue, given as a zero-based index. Items can be reordered by editing the value in that column. On left, there are two columns with buttons: remove and load. Items in the queue (but not the history) can be removed individually with the red "REMOVE" buttons. All items can be loaded into the sidebar where the parameters can be modified and a similar (or identical) item enqueued. The other information provided include the plan name, the status (enqueued, RUNNING, completed, failed, aborted, stopped or unknown) and the description field, which lists the arguments passed to the plan. Hovering over the description shows the full JSON of the item (see Figure 5.11, including an error message if applicable. The description has a contextual menu accessed via right clicking which provides additional options as shown in Figure 5.12. Most of the items in the context menu simply copy information to the clipboard: The full JSON as shown by the hover text, the item UUID which identifies the item in the queue, and the run UUID which identifies the data that is collected by the scan.

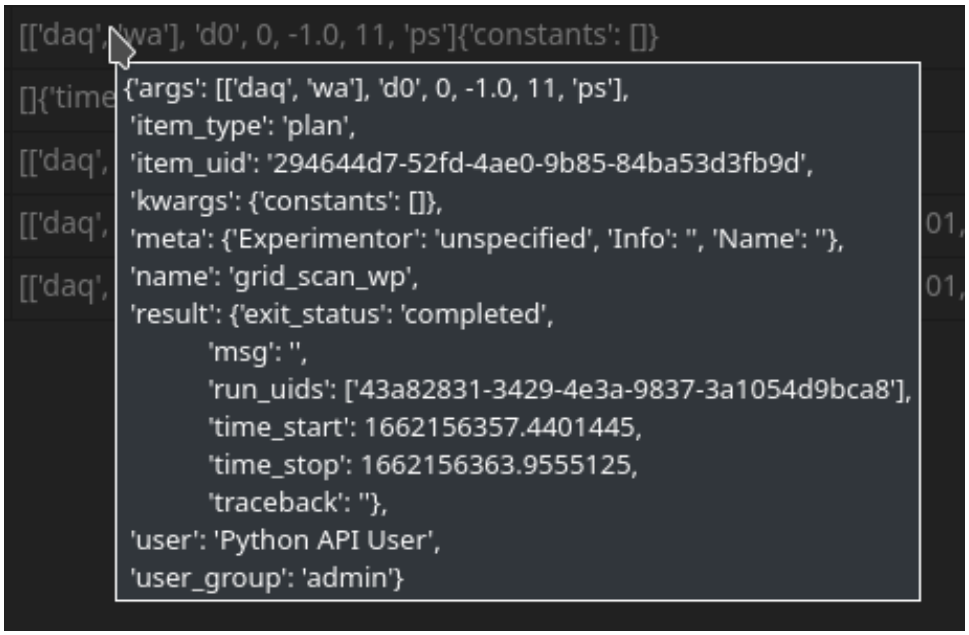

Figure 5.11: The hover text of a queue item.

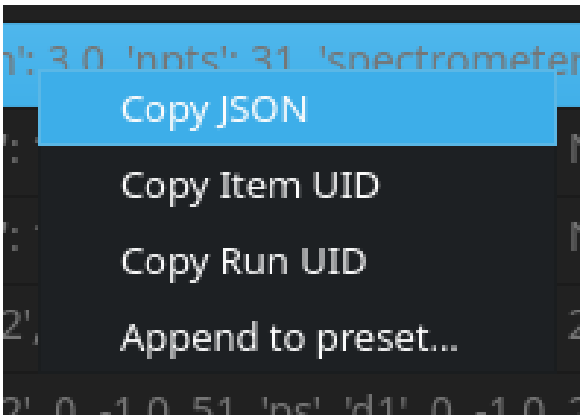

Figure 5.12: The right click menu of a queue item.

The Plot Tab

The plot tab (Figure 5.13 presents a live representation of recent data collected for the current plan. The primary window is a PyQtGraph[143] plot showing the most recent five slices of collected data. The current slice is the brightest cyan slice, with the previous slices shown as duller colors progressively until the last one which is gray. At the top, the currently plotted channel name and most recent collected value are shown in large font size. The plotting does handle array detectors, for which the top number shown is the maximum value of the most recent array collected. At present, cameras and other higher dimensional sensors are not able to be plotted using the live plot tab.

On the right hand side, there are controls to adjust the plot. At the top, a selector allows for choosing which channel to plot. The second selector determines which values appear along the X-axis of the graph. In particular, to switch between a scalar channel and an array detected channel, both selectors will need to be adjusted, as the appropriate axes differ. Additionally, there is a selector which allows for changing the units of the X-axis. Finally, there is a Scan Index indicator, which indicates which indexed pixel was the most recent collected.

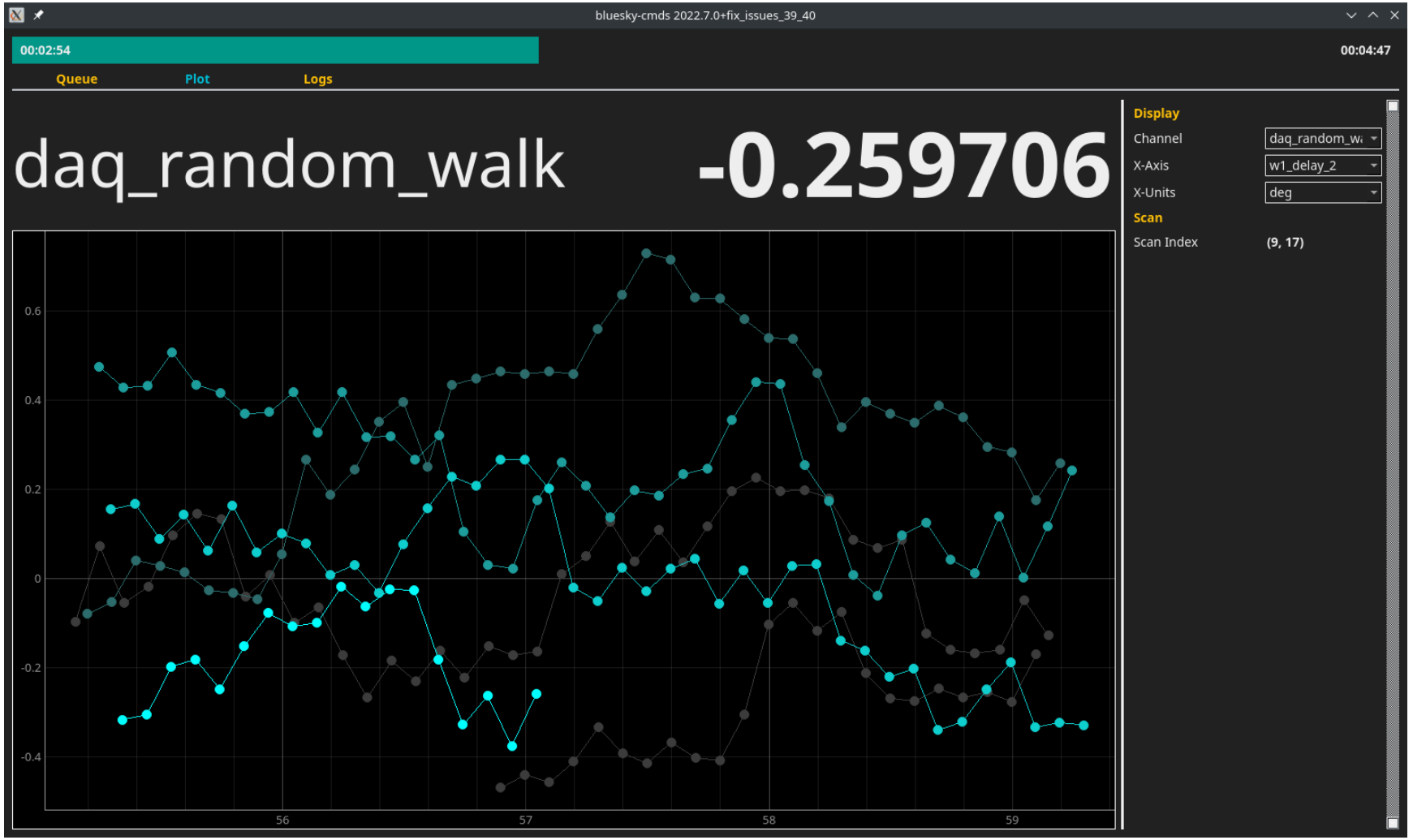

Figure 5.13: `bluesky-cmds` with the Plot tab open.

The Logs Tab

Figure 5.14 shows the Logs tab of `bluesky-cmds` . This is a text field which shows log entries from both the local application and the logs emitted from `bluesky-in-a-box` 's RE Manager. Log levels are color coded as are the logging instances which indicate what system is providing the log entry. At the bottom of the Logs tab, there is a button which clears the contents of the logging text box. The Logs tab is a good first step towards troubleshooting when something unexpected happens.

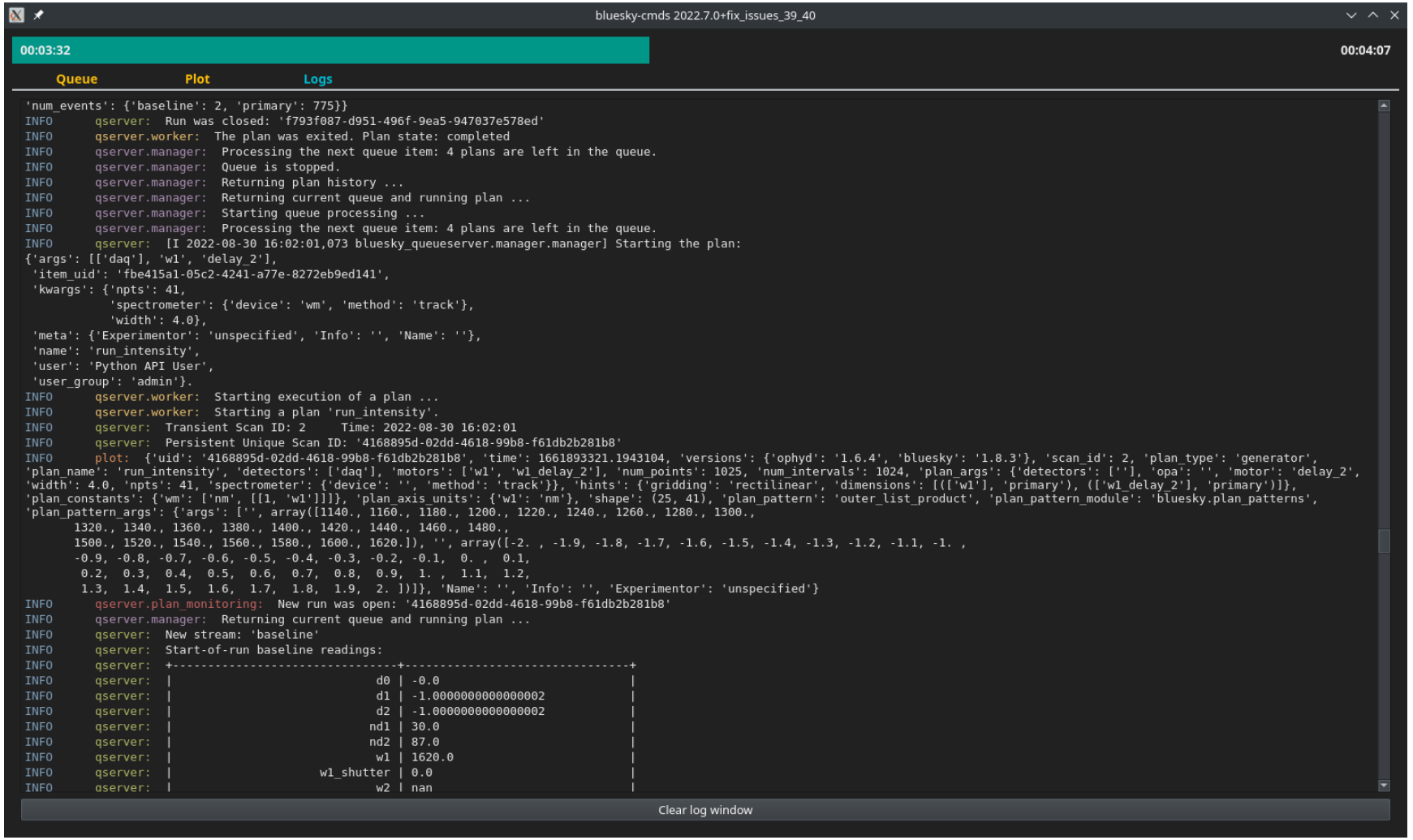

Figure 5.14: `bluesky-cmds` with the Logs tab open.

5.2.6 `wright-fakes`: Simulated Hardware for Development

`wright-fakes` is a collection of docker containers each running a single `yaq` daemon. `wright-fakes` provides an entire system, mimicking a laser table, but using only fake daemons to represent motors. It includes detectors, a monochromator, delays, neutral density wheels, and OPAs (including the component motors of the OPAs and delays).

This serves as a quick and easy way to start a suite of `yaq` daemons which is consistent and provides the variety of hardware needed to thoroughly test the data acquisition software.

Installation

These instructions assume that Docker is already installed, for more information see the 5.2.3.

Clone the git repository for `wright-fakes` :

```
git clone https://github.com/wright-group/wright-fakes
```
(5.42)

Inside of this repository there are folders for each available system (at time of writing, this is only `fs`). Navigate in a command prompt to the folder for the system you wish to start and run:

```
docker compose up --build
```
(5.43)

This builds the system of simulated daemons and starts running the daemons through docker. If the port that is expected for a `yaq` daemon is already in use, the container will fail to run, much the same as if you tried to run a second instance of a `yaq` daemon

If you are using Docker Desktop, then you can restart the fake daemons from the "Containers/Apps" tab of the Docker Desktop application, in the same manner as `bluesky-in-a-box` .

Once installed, you need to update the `yaqd-control` cache so that it knows about all of the daemons:

```
yaqd scan
```
(5.44)

On Windows, this can be a long process, so you may wish to search only the ranges where actual daemons are running

```
yaqd scan --start=38001 --stop=38003
yaqd scan --start=38401 --stop=38403
yaqd scan --start=38451 --stop=38453
yaqd scan --start=38500 --stop=38501
yaqd scan --start=38550 --stop=38551
yaqd scan --start=38999 --stop=39000
yaqd scan --start=39301 --stop=39303
yaqd scan --start=39501 --stop=39508
yaqd scan --start=39601 --stop=39608
yaqd scan --start=39701 --stop=39703
yaqd scan --start=39876 --stop=39877
```
(5.45)

5.2.7 Bluesky Community and Ecosystem

While an impressive library in its own right, the real power of Bluesky comes from the ecosystem that surrounds the library. There is a community of people who are all working towards making various pieces of the ecosystem better. Some are adding hardware interfaces, much like `yaqc-bluesky`, which expand the available hardware that can be used by Bluesky. The "default" such library, Ophyd[127], is currently undergoing a major transformation with the goal of making it easier to adapt to new hardware and interface libraries[144]. While `yaq` is a key piece of enabling the Wright Group to use Bluesky, the future could include using one of the alternate libraries for some key pieces of hardware, which might not even need any work on the part of Wright Group members themselves, other than configuration.

In addition to hardware, other components of the ecosystem are constantly being improved and worked on, including ones that we do not currently use. This includes components for building GUIs, Bluesky Widgets, and access to data after it is recorded, Databroker and Tiled. These three projects, in particular, are worth monitoring for their features and periodically considering if it is worth the time investment to adapt workflows of the Wright Group.

Bluesky Widgets

Bluesky Widgets[145] is a library which provides reusable graphical components for interacting with Bluesky and specifically for Bluesky QueueServer. In addition to the reusable components, the library provides a set of example applications which show practical versions of the components the library provides. Bluesky Widgets is actually already a dependency of `bluesky-cmds` , though only for its data model for ingesting event documents, and not for any of the Qt widgets which are provided. Some of the example applications, such as the "Queue Monitor" application may prove to be generally useful either along side of or instead of programs like `bluesky-cmds` .

Databroker

Databroker[146] is the ecosystem provided tool for accessing data from Bluesky acquisitions. Primarily, it returns data as Xarray[147] Dataset objects, which can then be manipulated, plotted, and stored for future use. In particular, Databroker can read data directly from the Mongo database that stores each event in sequence. This is part of the reason for even having the Mongo database in the first place. Ultimately, the initial decision was to ease the transition for users by providing WT5 files much like they are already used to. However, the tools provided by databroker and Xarray may yet prove to easily allow some data manipulation that is hard or impossible with WrightTools and WT5. Additionally, Databroker is in the process of converting to being built on top of the next project, Tiled, so perhaps it would be better for users to use Tiled directly even if the further tools such as Xarray are desired.

Tiled

In many ways, Tiled is the successor to Databroker. As a project, Tiled is quite ambitious, aiming to provide a unified interface to many different forms of data. Tiled provides a server, which ingests data from folders or databases and provides metadata about the tree of data provided. Tiled also provides a client Python library, which includes the ability to search data, download data in a variety of formats, including loading only portions of the available data. Additionally, there is a web server which provides the ability to browse data and view previews as images or graphs interactively. Tiled is a powerful tool

that is well designed and built on standards. Tiled was not immediately used by the Wright Group because a) it was actively being developed in parallel, b) it represents a larger change to user workflow than we wished to introduce in one step, and c) it incurs additional systemic complexity of running an additional server component. In the future, it may be possible to get a large portion of the benefits of Tiled, with minimal disruption of user workflow, by using plug-ins to allow tiled to read and provide WT5 files directly.

Part III

Applications

Chapter 6

Active Correction as Daemons

6.1 Introduction

Blaise Thompson extensively discussed multiple forms of active corrections applied to CMDS instruments in Chapter 6 of his dissertation[13]. Most prominently, this includes Spectral Delay Correction (SDC). At the time, active correction for SDC was performed by the orchestration layer, PyCMDS, as the "autonomic" system. The implementation of the autonomic system was tightly coupled to the custom orchestration provided by PyCMDS. This proved inflexible, and caused the hardware definitions to be intractable in many cases, leading to hard to understand behavior. Herein, I provide two examples of how I applied these ideas using `yaq` daemons. The first is Spectral Delay Correction itself. The second is a more generalized N-Dimensional interpolator which was written to aid Emily Kaufman with her project regarding active phase matching correction.

When the Wright Group was considering using Bluesky as an orchestration layer, the behavior of the autonomic system was one of the key pieces that made that transition seem hard at first. There did not seem to be an easy and correct way of inserting the logic for this kind of offsetting into the Bluesky Run Engine, as it was implemented for PyCMDS. Additionally, Bluesky meant separating the behavior as collected, which runs through the Run Engine, from the behavior of hardware in an interactive session, as when attempting to initially find signal. Upon reflecting, and taking a step back to consider the desired outcome, the solution became obvious: instead of inserting this logic as a modifier to the Run Engine, that logic can be implemented as a daemon which wraps the hardware daemon. This intermediate daemon provides the controls of offsets being applied or ignored. Additionally, daemons can be more specialized to the particular task, and thus are easier to use and describe compared to the general purpose autonomic system of PyCMDS. To the client program, whether that is an interactive client like `yaqc-qtpy` or the Bluesky Run Engine, this daemon looks and acts like a simple motor that gets told to go to a particular position and does so.

Herein, we will look at two instances of active correction as daemons. First, SDC itself and the `attune-delay` daemon which implements it. And second, the `ndinterp` daemon which allows for more complicated relationships among controlling hardware.

6.2 attune-delay

Spectral delay occurs largely due to transmissive optics which have a wavelength-dependent index of refraction and changes of the optical path length within the tunable light sources. When pulses are less a picosecond long (or even shorter), even small variations will cause overlap in time to diminish rapidly. Spectral Delay Correction (SDC) is an active correction technique which employs the delay motors already included in the instrument to adjust the path length of each beamline to account for any variation in arrival time as a function of wavelength.

SDC "curves" are represented as Attune `Instrument` objects. This allows reuse of utilities that were used for OPA tuning curves previously. For an OPA `Instrument` object, the whole thing represents one light source, each `Arrangement` represents one configuration of motors to produce a unique range of light, and each `Tune` represents the mapping of a single motor position (or another `Arrangement`) from the input color. In contrast, for an SDC `Instrument` object, the levels are all shifted one down: the light source `Instrument` keys are the `Arrangement` keys of the SDC curve, and the `Arrangement` keys from the light sources are the `Tune` keys. The `Tune` outputs are not motor positions directly, but rather offset in picoseconds for the particular light source using the particular `Arrangement`. This allows a single `Instrument` to fully describe all possible required offsets for a laser system which may use multiple `Arrangements` to do experiments. The total offset is the sum of all offsets from each light source.

The objective is to create a daemon which performs the necessary offset. Such a daemon will wrap a daemon for a translation stage, but can also perform the logical transform from displacement in length units to displacement in time units from some arbitrary zero mark which can be configured. The offset will change whenever one of the values used to compute it changes: either a light source changes color or `Arrangement`. When creating a daemon to compute and control the delay and offset, there is a natural inclination to have this daemon be a client of the light source and simply call for the color of the light source every time it computes the offset. The problem with this is that the daemon has no way to tell that the color of a light source has changed. To remedy this, a delay offset computing daemon would have to poll each light source for their position so that it could react as it needs to. This tight polling is expensive, and requires balancing polling rate and responsiveness. Additionally, when the light

source changes color, it is possible that the delay will be the slowest motor, and so any client program will want the light source to remain busy until the associated delay is done moving. All of these factors combined mean that instead, the Attune daemon which controls the light sources must be a client of the Attune Delay offsetting daemon. The Attune Delay daemon simply caches the position and active arrangement of each controlling hardware and uses that cache to compute offsets. It must recompute and reapply the offset whenever any controlling light source moves. The Attune daemon can poll its associated delays for busy status whenever it moves, maintaining its own busy state that a client might be polling. The client-daemon relationships are summarized in Figure 6.1

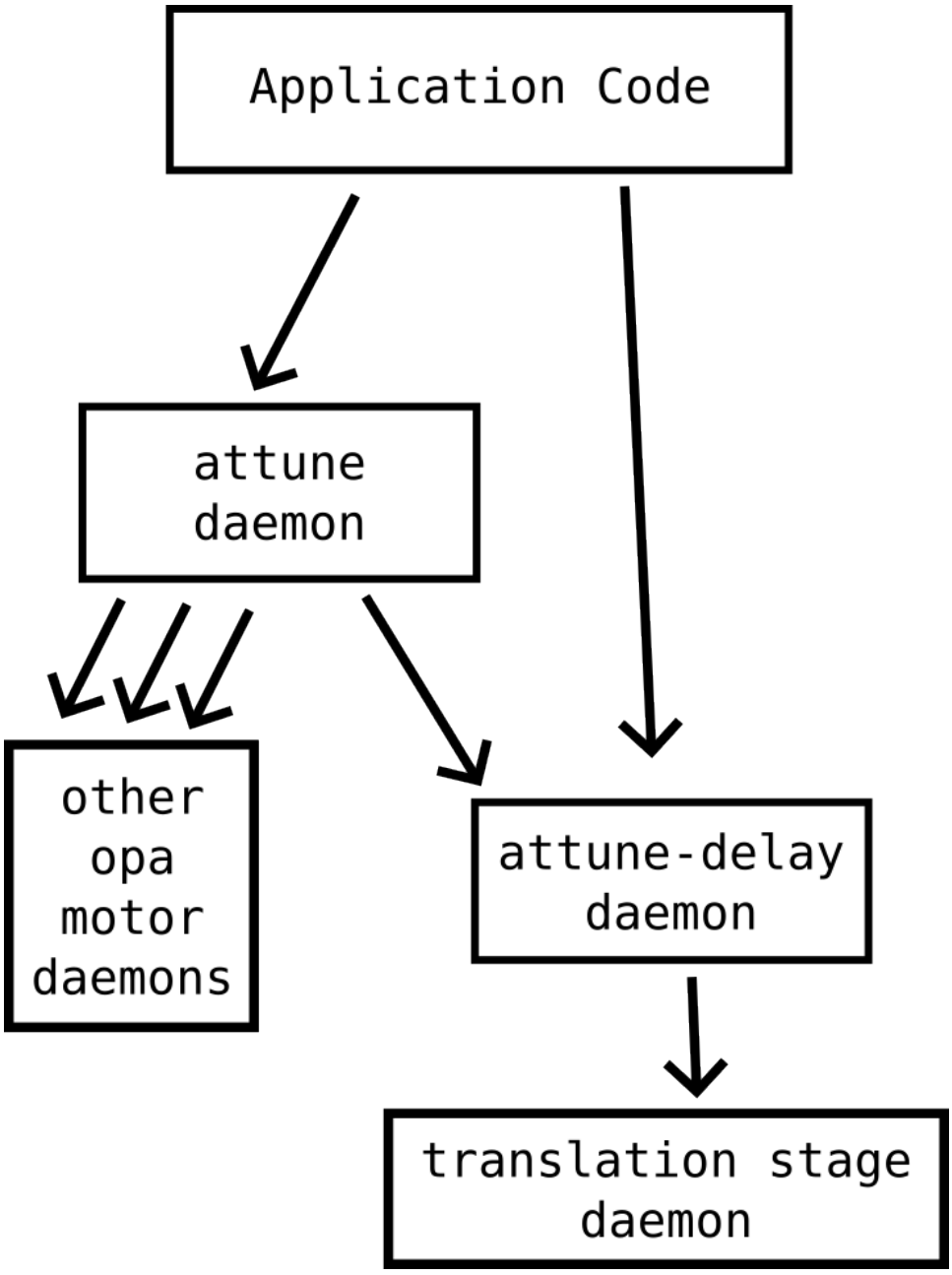

Figure 6.1: The relationships between clients and daemons with an Attune Delay daemon.

To make this work, the Attune Delay daemon exposes a set of messages that the light source Attune daemon can call to communicate changes. Specifically, `set_control_tune` and `set_control_position` are called by the light source daemon at initialization and whenever the active arrangement or position, respectively, are changed. When the Delay daemon receives these calls, it will recompute and reapply the offset. The former accepts a string for the control key, i.e. the name of the light source daemon, and a string for the tune, which is the arrangement of the light source. The latter similarly accepts the control key and a double precision floating point position. Those are the only two messages that are used by the Attune daemon.

In addition, there are messages used by engineering interfaces or specialized clients to control the behavior of the Attune Delay daemon. `set_control_active` allows for selectively turning off offsets from each light source. This is useful for collecting the SDC offset curves themselves, which are typically collected without active correction applied to avoid needing to add two corrections together. The position and tune caches are updated even if the controlling hardware is inactive. This allows the correct offset to be applied if the hardware is toggled to active. `set_zero_position` takes a position in the coordinates of the underlying translation stage and calls that position "zero" for the logical delay. `set_instrument` allows overwriting of the entire `Instrument` object. This is used by data processing scripts to apply newly collected SDC curves. The `Instrument` is stored in the Attune Store, where it can be reloaded. Client connections are interrupted so that they are aware that assumptions about the applied `Instrument` are broken and they must refresh their cached copies.

Each of the setting methods mentioned above have associated methods for retrieving the information, as well as methods to get related quantities such as limits or units.

In all, the Attune Delay daemon looks just like any other `yaq` daemon to a client program which does not require knowledge of the details of the offsetting behavior. While all of the information is available to clients, if one wishes to do a simple one dimensional scan, it behaves just as any other motor. Its only functional difference in that regard from the underlying translation stage is that the units and zero position are changed to represent the logical behavior of the stage as part of a larger instrument rather than the physical behavior of the stage alone.

6.3 ndinterp

Phase matching is an important factor contributing to the constructive interference of non-linear mixing processes. Some mixing processes cannot be phase matched, and therefore have destructive interference which causes diminished signal with increased path length. To properly account for phase matching, relative angles of beam propagation combined with the energies and indices of refraction for each incident beam and the generated output beam must be taken into account. Since it is wavelength dependent, a desired experiment may span a region where the optimal phase matching requires incoming light at significantly differing angles. The Wright Group, and in particular Emily Kaufman, have been attempting to actively correct for angle of incidence to ensure proper phase matching for all wavelengths of a desired experiment. To accomplish this task, the idea is to move the retroreflector laterally to displace the beam horizontally while maintaining the output angle from the retroreflector. The beam will therefore hit a different position on the focusing optic. The focussing optic will maintain focus for the parallel incoming beams to the same spot on the sample, but with differing angles.[1] Depending on the free aperture of the mirrors and the focal length of the focussing optic, this can provide several degrees of available offset while maintaining spatial overlap at the sample. Figure 6.2 shows the effect of translating a retroreflector as described above. This figure is not to scale, simply a cartoon illustrating the principle. The ideal angle can be calculated, but is dependent on multiple laser colors in complicated ways. Unlike SDC, phase matching corrections are not separable for each light source and additive.

[1] This is technically only true for parabolic mirrors, but in practice is often a good enough approximation for spherical mirrors used approximately on axis

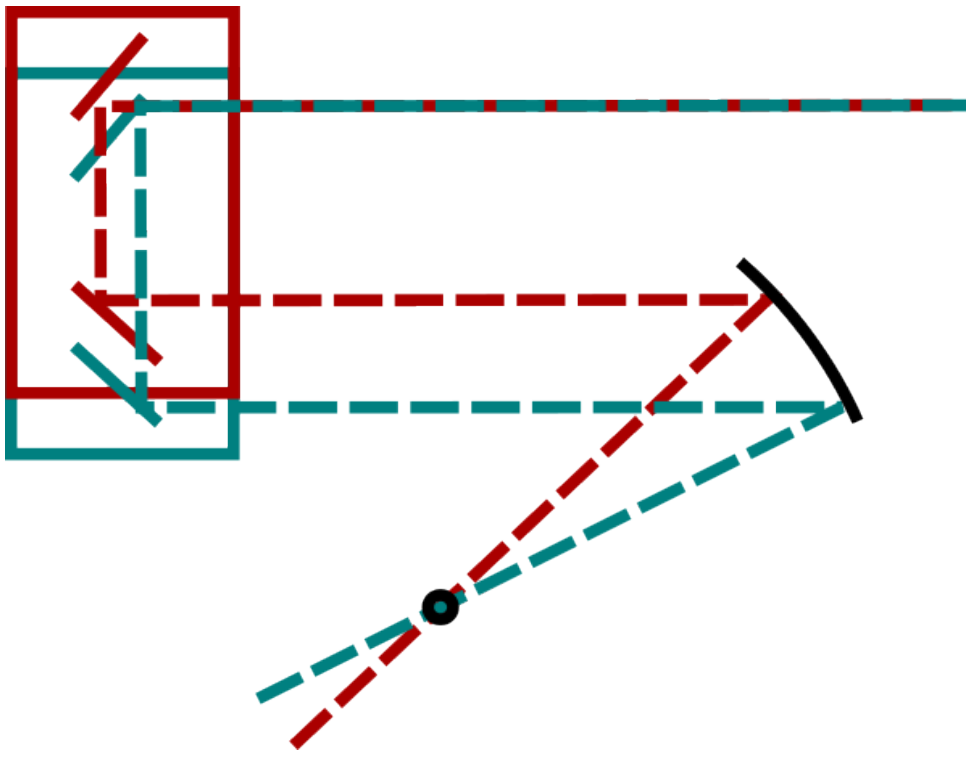

Figure 6.2: The same incident beam angle will be translated if the retroreflector is translated. A focussing optic causes the beams to converge to the same location in space but at different angles.

The hardware required for active phase matching is a translation stage mounted perpendicular to the beam path with a retroreflector. Because a retroreflector is already required for delays, this lateral translation stage is mounted on top of the delay This does require some tight consideration for physical size of the translation stage, as it must both support the mass of the retroreflector and be short enough for mirrors to sit on top.

For this process, the information required is almost precisely the same as the Attune Delay daemon detailed above. So, to accomplish the task, we must implement a daemon that looks to the light source Attune daemon exactly like an Attune Delay daemon. This is not strictly required for this daemon specifically, but does allow it to be quickly implemented without changing existing daemons. The daemon must be notified of when controlling hardware changes position and recompute its offsets. This daemon, called NDInterp, performs N-Dimensional interpolation to be a maximally flexible active correction daemon. However, the `Tune` information is not used, so a method which does nothing is provided so that the calling client daemon does not have to be changed. Instead of being backed by an Attune `Instrument` which accepts independent offsets which are additive, a single multidimensional offset is used. This multidimensional offset is represented as a WrightTools Data file. The data file is configured to have a single `WrightTools.Data` object which has axes of the control hardware and a channel which represents the required offset. Similar to Attune Delay, NDInterp has a zero position from which all offsets are relative.

Since the offset is fundamentally multidimensional, independent control hardware cannot be toggled on and off like they can for the Delay daemon. Instead, there is a single boolean `yaq` property which can disable the offset behavior entirely to allow for experiments which do not require the active phase matching.

This approach to active phase matching allowed us to quickly and easily write a daemon that is capable of performing the active correction required.

Chapter 7

OPA400: A custom OPA built with `yaq`

7.1 Introduction

An Optical Parametric Amplifier (OPA) is a device which allows for tunable control of the color of light produced. In principle, a "pump" photon is split into two new photons, conventionally called "signal" and "idler" for the higher and lower energy photons, respectively. The signal and idler photons observe conservation of energy, meaning that the sum of the two photon energies is equal to the pump photon energy. The energy split is determined by a phase matching condition, which in our case is the angle with respect to the optical axis of a birefringent crystal, β-barium borate (BBO). This process can be seeded by photons of the signal color, which increases efficiency of conversion. To obtain this seed, an initial preamplification step is performed by using white light mixed with the pump laser in the birefringent crystal to produce a broad spectrum output. This broad spectrum output is reflected off of a grating to select a narrow range of frequencies to be amplified using a second pass through the crystal. Once signal and idler photons are created, additional mixing processes can be performed in additional crystals to produce new tunable wavelength ranges. Commonly this includes doubling or even quadrupling the signal or idler photons, or adding in residual pump photons to either signal or idler.

For the picosecond laser system operated by the Wright Group, the upstream "pump" laser is 800 nm (12,500 cm^{-1}). For OPAs pumped from this light, signal and idler are in the mid-infrared region, requiring additional mixing if visible light is desired. What sets OPA400 apart from the other OPAs on the picosecond system is that rather than performing the additional mixing on the output photons, the pump laser is instead doubled to 400 nm (25,000 cm^{-1}) prior to splitting into signal and idler. This means that the signal photons are themselves visible light between 750 nm and 450 nm, and the idler are correspondingly near infrared photons. This provides a conveniently large tunable range in the visible light region with a single, smooth, tuning curve which avoids issues of needing to stitch together acquisitions from multiple tuning ranges.

This project was primarily undertaken by Dr. Daniel Kohler. I played a particular role in ensuring that the software and custom electronics made for this project could perform the required tasks.

7.2 Design

The design of OPA400 encompasses both the optical components, the mechanical components used to control the photon split, and the driving electrical components for the mechanisms. Many of the details of the design, including the custom electronic and mechanical parts, are available for this project

7.2.1 Optical Design

The optical design of OPA400 mirrors closely the same principles of designing the laser table as a whole. Optical path length is a critical dimension, so care must be taken when laying out optics and delay stages for fine adjustments are required. The path lengths are folded for compact spatial design and having proper degrees of freedom for alignment. High intensities are required for efficient generation of non-linear mixing, so focusing optics are required. Overlap of multiple beams in the BBO crystal is crucial in both space and time. Chromatically selective beam splitters are used pick off specific colors of light while allowing other colors, such as the parametric output, to pass through. The layout of OPA400 is shown in schematic form in Figure 7.1 and in photograph form in 7.2.

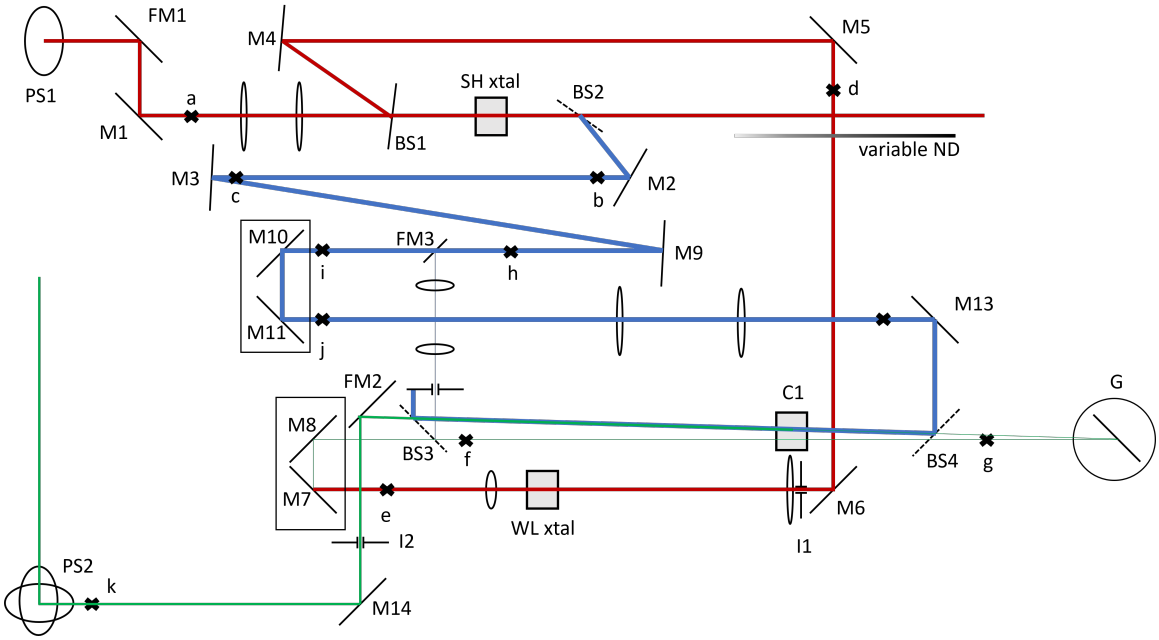

Figure 7.1: Schematic view of OPA400. Input 800 nm light shown in red, doubled 400 nm light shown in blue, parametric output shown in green.

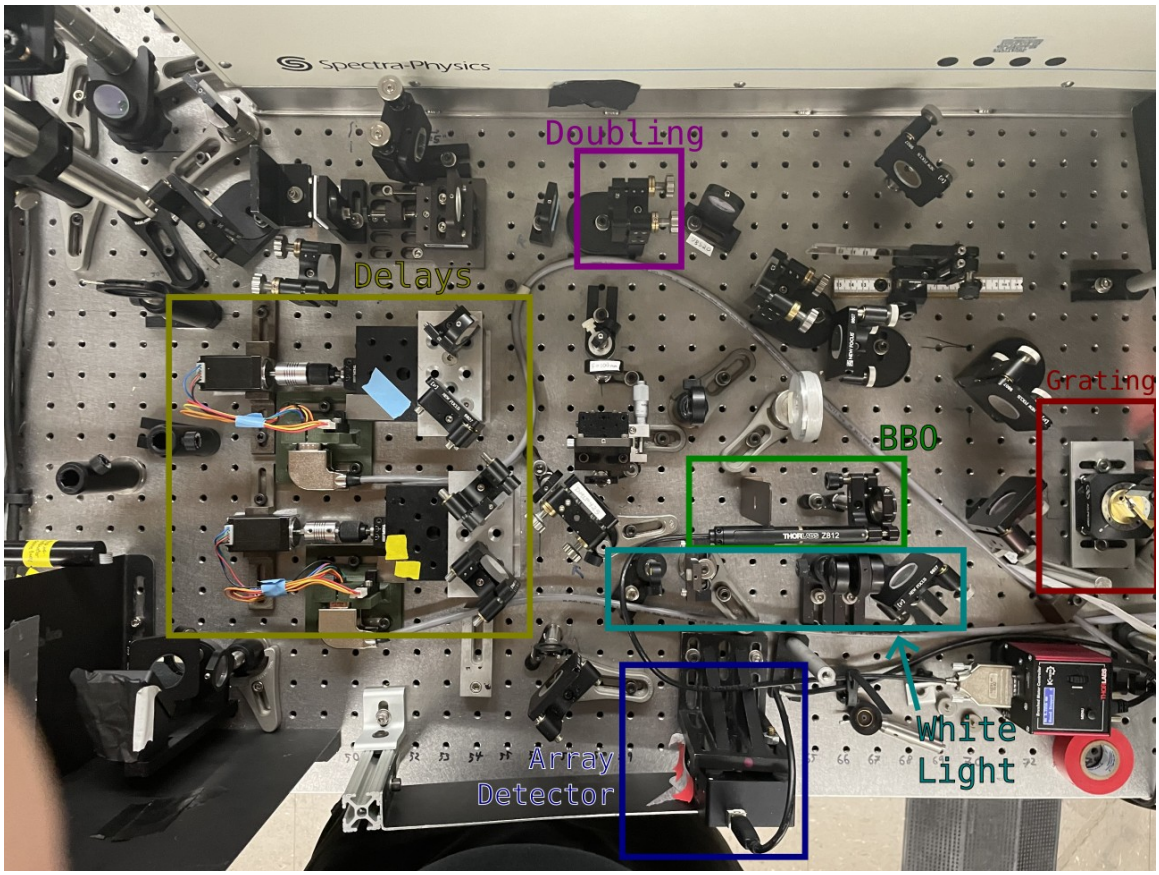

Figure 7.2: Aerial view of OPA400. The 800 nm light enters at the top left and is doubled to 400 nm (purple). A small portion of 800 nm light is used to generate white light (teal). The white light and a portion of the 400 nm pump light are combined initially in the BBO (green). This creates a broad spectrum output that is bounced off of the grating (red) back to the BBO (green). The delays (yellow) ensure that light arrives in the BBO at the same time, which is a requirement for mixing to occur. An array detector (blue) allows monitoring of the output visible signal photons.

7.2.2 Hardware Selection

The hardware selected follows a "homebuilt" approach. Where possible, namely for the delay stages, standard NEMA stepper motors were used because they are cheap and highly available. Where precision and/or compactness was a strict requirement, commercially available motors were used. This includes the Thorlabs Z812[148] motor for controlling the angle of the BBO Crystal and the Newport CONEX-AGP[149] rotational mount for the grating.

Creating a complex component such as an OPA is an iterative process, with needs evolving as testing is performed. For example, initially we opted not to include a grating, relying on phase matching angle alone for selectivity of the signal photon color. This proved possible, but not as efficient or stable as we desired, so the grating was added as a revision. The array detector, an Ocean Optics USB2000, is an optional but convenient addition which provides a tighter feedback loop when aligning the OPA. Additionally, the original design used a Newport stepper motor to control the crystal angle. The intent was to drive this motor with alternate stepper motor drivers. This motor did not ultimately work as desired, so it was replaced with the Thorlabs motor and the associated manufacturer provided motor driver.

7.3 Stepper Control Box

The control system for this OPA centers around a Raspberry Pi 4[150]. The stepper motors are driven by Adafruit Stepper Motor HATs[151]. These stepper motor drivers come in a convenient form of a circuit board that fits directly on top of the Raspberry Pi. Each board can support two stepper motors, driven at a voltage provided to a power input that is not shared to the Raspberry Pi itself. These boards can stack, allowing as many stepper motors as needed to be controlled from the same computer.

On top of the stepper motor driver boards, another hat is used to provide limit switch inputs. This board is custom designed, containing eight digital inputs which accept 5 Volt input signal and provide a 3.3 Volt signal to the Raspberry Pi General Purpose Input Output (GPIO) pins. The correspondence between input number as provided by the hat and digital GPIO pin used by the Raspberry Pi is provided in Table 7.1. Each input has three pins associated with it: 5 Volts, Ground, and a signal pin. The signal pin has a pull-up resistor for 5 Volts. If a simple mechanical switch is used for the input, a nominally open switch between the signal and ground is sufficient. The 5 Volt line is provided so that sensors such as optical interrupts which require power to operate can be used easily. Figure 7.3 shows a schematic representation of the schematic for the level conversion and LED indicator for a single input. This motif is repeated eight times. Figure 7.4 shows a rendering of the circuit board itself.

Interrupt Number	GPIO Pin
1	17
2	22
3	5
4	6
5	13
6	26
7	23
8	24

Table 7.1: The pin correspondence the interrupt hat.

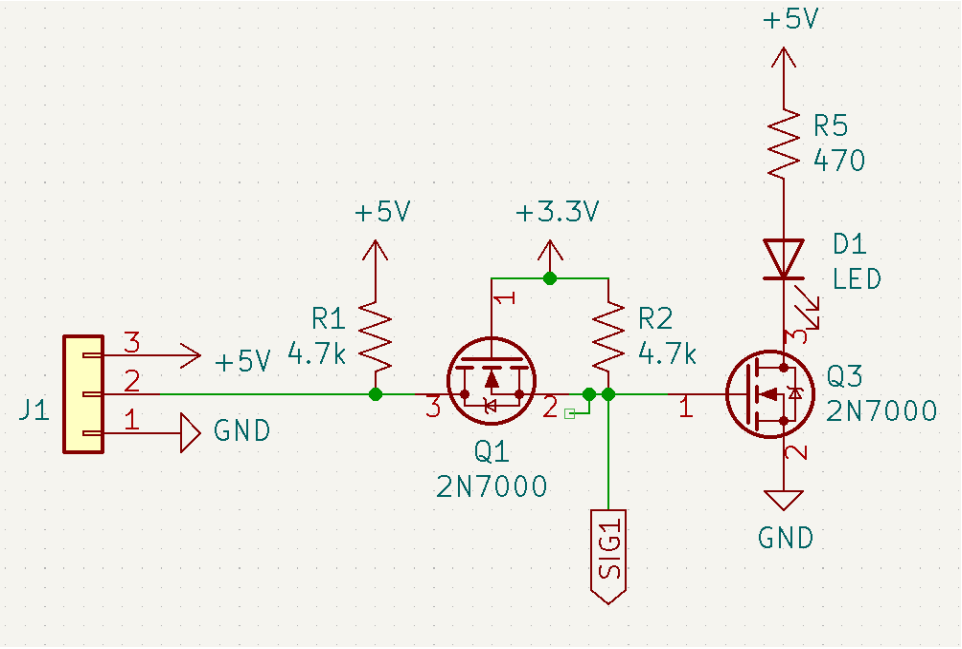

Figure 7.3: A schematic view of a single input for the Interrupt Hat.

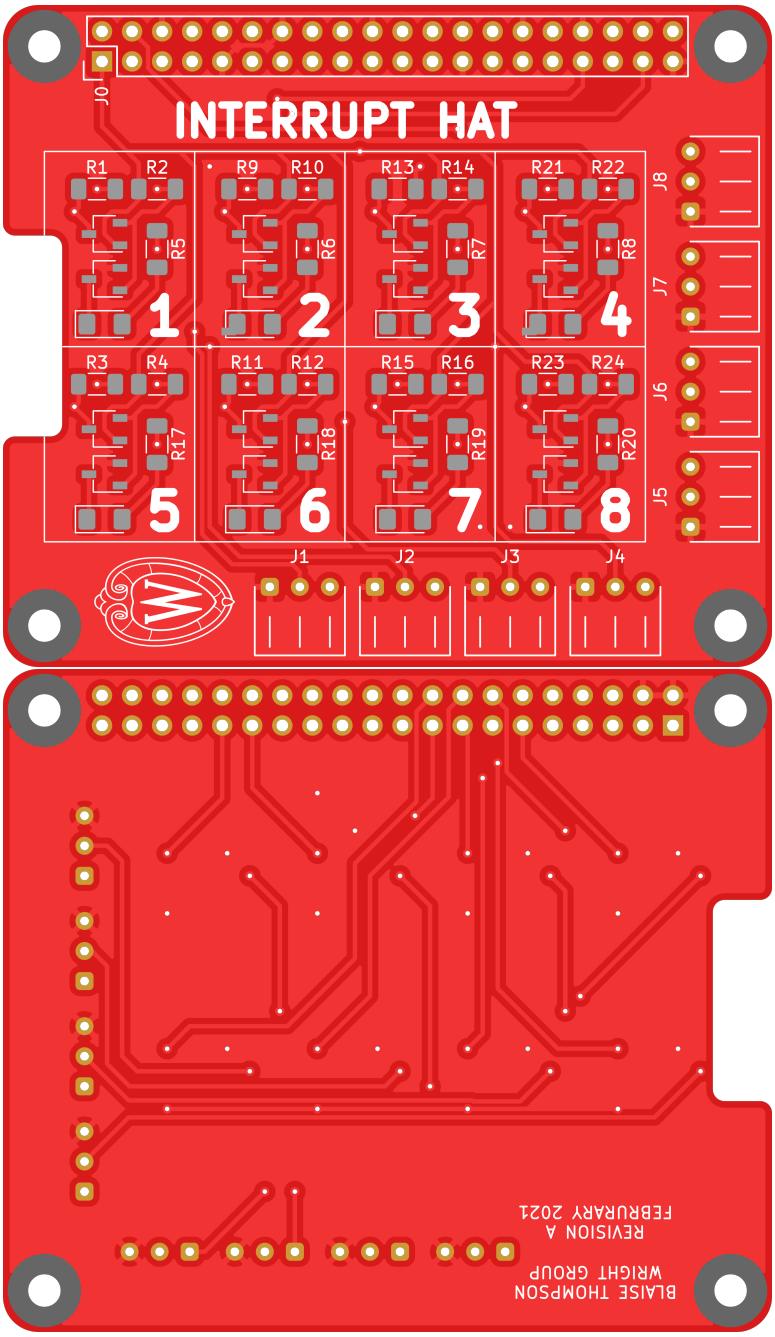

Figure 7.4: Rendering of the interrupt circuit board, top view and bottom view. Rendering performed using tracespace[152].

The stepper control box is designed to work with the Mean Well RS-15 series of DC power supplies[153]. Each hat level (two steppers) can accept an independent voltage level, so stepper motors with differing specifications can be used and controlled by the same system. The box is designed so that three power supplies can be installed as needed. One of the power supplies should be a 5 Volt power supply so that the Raspberry Pi itself can be powered from the same source.

The box is constructed of metal base with mounting holes drilled for the power supplies and the Raspberry Pi assembly. At the corners of the base, 15 mm aluminum extrusion posts provide support for the panels. The side panels and top are clear acrylic cut to size to slot in to the aluminum extrusion. The top is secured with screws, with tapped 3 mm screw holes in the center of the posts. This allows visual access to the LED indicators of the Interrupt Hat. The front and rear panels are 3D printed plastic, with mounting for connectors. The rear panel has AC power input and a power switch. The front panel has a cut out for access to the Raspberry Pi USB and Ethernet ports as well as connectors for the stepper motors. The pinout for the Wright Group's standard DE-9 D-subminiature stepper motor port is provided in Table 7.2. Figure 7.5 shows a photograph of the completed control box.

Pin	Connection
1	NC
2	Limit Signal
3	+5 V
4	GND
5	NC
6	B2
7	B1
8	A1
9	A2

Table 7.2: The pinout for the stepper motor DE9 connector.

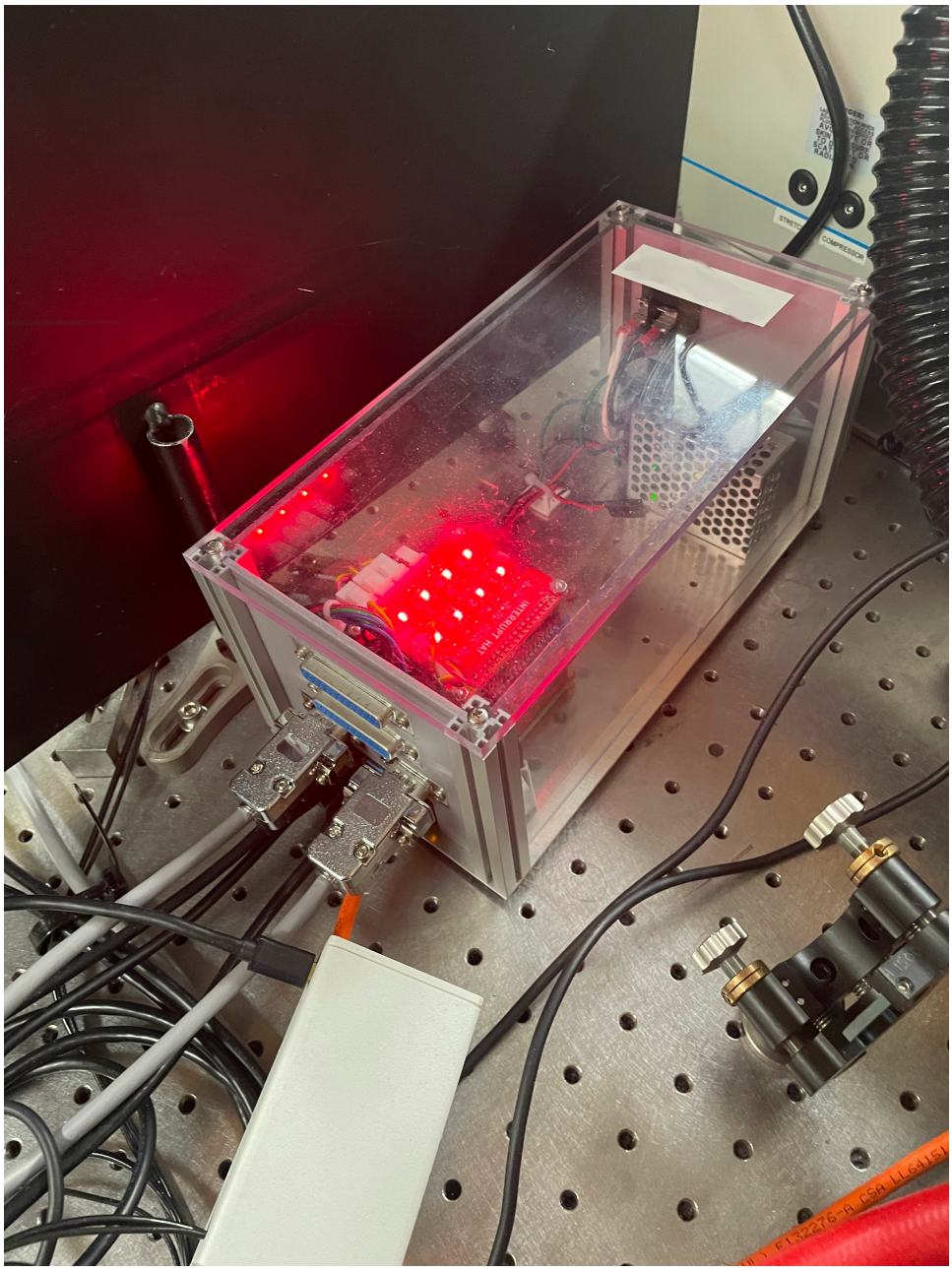

Figure 7.5: Photograph of the stepper control box.

7.4 Delay stages

The delay stages themselves have custom components designed for this application. A 3D printed part provides a mount for a connector and the optical interrupt which is used for homing the motor. A small piece of sheet metal is the flag which interrupts the optical interrupt. The delay mechanism itself is a simple manual translation stage purchased from Thorlabs[154]. This stage has a fine lead screw which provides the necessary precision to properly overlap the picosecond light pulses. Below the stage, a machined aluminum block raises the stage to a height where the motor can directly drive the lead screw. There is an additional adapter plate on top of the translation stage which provides a low profile mount for two mirrors at 90 degrees from each other, forming a planar retroreflector. The custom machined parts in this system are important because space, particularly vertical space, is limited to have the mirrors in the desired position. These parts were machined out of aluminum rather than 3D printed with consideration of the thermal expansion and rigidity of the parts which are directly supporting the optics.

The motors are coupled to the stage using a small aluminum rod machined to size and epoxied in place and a spring coupler which was purchased as a part for a 3D printer. The motors themselves are mounted using two L-brackets pinching the motor. Originally, the design included mounting the motor to the 3D printed part which also houses the connector and interrupt. This proved problematic, as the forces exerted on the motor in the designed configuration meant it was unable to turn the stage. Aligning the motor so that it can turn the stage is not difficult, but required more flexibility than a single mounting point of the 3D printed part.

7.5 Software

To control OPA400, a series of `yaq` daemons were implemented. Two of the motors were commercial products, the Thorlabs motor for the BBO crystal and the Newport motor for the grating. These two had extant daemons already implemented prior to their use for OPA400[155][156]. The Newport motor was already in use in other experimental setups, and had been written with the particular hardware that ended up installed in OPA400. The Thorlabs motor was purchased specifically for this project, but uses a common protocol with other similar motors. The daemon was already implemented, though it was tested specifically with the controller prior to installation. These daemons can run directly on the Raspberry Pi, which makes for simpler cable management.

In addition to the previously created daemons, a new daemon was made to control the Adafruit stepper motor drivers. This daemon uses two python libraries provided by Adafruit, the CircuitPython MotorKit[157] and CircuitPython Motor[158] libraries, as well as one library for interacting with the Raspberry Pi GPIO pins, GPIO Zero[159]. The first two are used to control the stepper motor, instructing each individual step. The latter is used to read the state of the optical interrupt. The daemon, implemented in about 100 lines of Python code, allows users to interact with the delay stages as they would with a daemon implemented for a commercial product. Originally, this daemon was going to use another daemon for reading the interrupt state, but that was abandoned because of timing considerations and the fact that a daemon built for a Raspberry Pi hat is already a well constrained environment. These daemons must be run directly on the Raspberry Pi, as they need direct hardware access to the GPIO pins.

Once each individual motor has an appropriate daemon implemented, the motors can be included in an Attune Instrument, just like other OPA models. This Attune Instrument has its own daemon that can run on main instrument machine, which allows for easy access to update and read the Attune Instrument history.

When starting a new Attune Arrangement (or in this case a new Instrument altogether) a good initial strategy is to manually find appropriate motor positions for a small number of points. Approximately five points, including points near to both ends of the expected usable range, is usually good enough to

allow a general sense of the curvature of each Tune. This rough tuning curve can be scaled up to more points (often around 20 points, though the degree of curvature and range will vary how many points are required for satisfactory interpolation) using `attune.map_ind_points`. This upscaled Tune will not be fully accurate, but will usually be close enough that the standard tuning procedures can capture the offsets.

The standard tuning procedure for OPA400 is as follows. First, a careful alignment of the optics, following standard operating procedures to align to apertures and obtain single color output. Second a `run_intensity` for the pre-amp delay (D1, the white light line) with the grating at normal incidence. Third, a `run_tune_test` of the pre-amp color, using the array detector. Fourth, a `run_intensity` for the grating positions, detecting first pass amplification through an aperture.Finally, the second pass delay (D2, the delay for 400 nm light) is offset appropriately. In theory, this delay needs to compensate for some additional spectral delay inside of the OPA. In practice, a static offset is usually good enough for the whole range of colors.

Chapter 8

Waldo: A case study on building a full instrument with `yaq`

8.1 Introduction

This chapter serves as a case study on building an entire instrument with `yaq` from the beginning. In particular, we will look at the process of designing the system with `yaq` in mind. Herein, we examine the desired specifications for a new laser system, called "Waldo". With an understanding of desired behavior, we examine the choices made regarding hardware acquired for Waldo. Additionally, we will look at the implementation of `yaq` daemons for the selected hardware. Ultimately, this leads up to the assembly and implementation of experimental orchestration software for the system.

The design and implementation of Waldo was primarily led by Dr. Kent Meyer. Dr. John Wright worked closely with Light Conversion to provide a system with the requested specifications. My own role was primarily in providing the software to run the instrument, which included adding additional support for new hardware purchased for Waldo. Dr. Blaise Thompson provided hardware support for the Gage Data Acquisition card while I provided most other hardware support mentioned.

8.2 Design Requirements

Waldo was commissioned to serve an intermediate pulse duration between those of the existing laser systems used by the Wright Group. The Wright Group's older CMDS systems have nominal pulse durations of 35 femtoseconds and 1 picosecond. Both of these systems have an upstream light source that is 1 kHz 800 nm pulses of their respective pulse duration. The light from the upstream source is split into two OPAs for the femtosecond system and four OPAs for the picosecond system. The two systems combined allow for study of a wide variety of molecular and material systems, however there are many cases where the ideal pulse duration is in between the two so either system is a compromise.

8.2.1 Specifications

The desired system for Waldo has a pulse duration of approximately 500 fs. An increased, but controllable, repetition rate of up to 100 kHz is desired. The increased repetition rate allows for more signal averaging over the same amount of laboratory time, therefore increasing the signal to noise ratio. This laser system must have three tunable light sources: one primarily producing visible wavelength ranges, and two primarily producing infrared wavelengths. This arrangement of lasers is based on the picosecond system and is primarily designed to interrogate vibrational modes coupled to electronic modes in molecular systems using CMDS techniques such as Doubly Vibrationally Enhanced (DOVE) or Triple Sum Frequency (TSF) spectroscopies.

Since this system is designed for fully coherent experiments, a relatively short travel delay stage on two of the beam paths is sufficient. Approximately 50 mm is appropriate, with sufficient resolution to allow optimal temporal overlap of the sub-picosecond pulses.

The monochromator must be capable of discriminating both infrared and visible light for tuning and experimental data collection. The signals for tuning are quite strong lasers, while the signals for many experiments are comparatively weak, so it must be able to handle both use cases.

The detectors and electronics for measuring the signal from the detectors must both have instrument response functions that decay between laser shots. While the 1 kHz systems had one millisecond between

successive laser shots, this system will have only 10 microseconds. Thus detectors with relaxation times that were appropriate for the other laser systems may not be appropriate for this system. Additionally, there is less time to sample multiple signal channels serially, so it becomes important that the data acquisition card can do multiple simultaneous digitizations.

8.2.2 Hardware Selection

When selecting hardware for a new laser system, it is often a good idea to start with equipment that you already know. Even if the precise equipment you have is not sufficient, having common factors such as communication protocols will make integrating the new components much easier. It is often reasonable to use hardware from the same manufacturer as other hardware, as they often use the same low level interfaces.

In this instance, using this principle means that many of the `yaq` daemons for a brand new system are already written, or at least similar enough that they require only small perturbations from extant daemons.

While Python in many ways is a "glue" language that *can* make calls to external libraries, a manufacturer provided Python library is a distinct advantage towards confidence that a `yaq` daemon will be quickly and easily implemented. Ideally, these libraries are themselves Open Source and distributed on PyPI[113] and conda[45]. Being distributed on these standard platforms makes listing the libraries as dependencies easy and makes distribution of the `yaq` daemon simpler. Being Open Source obviates any potential copyright issues that could be taken if code from the manufacturer is distributed to make the `yaq` daemon work. It also allows for changes to be made if bugs are discovered or feature enhancements are desired to make using the library easier.

For Waldo, the upstream laser system was custom designed to the desired specifications by Light Conversion. The pump laser is a CARBIDE 1030 nm 100 kHz laser[160]. This pump light is fed into two systems: an ORPHEUS-HP for the visible tunable light and an ORPHEUS-TWINS which provides two separately tunable infrared lines. Each of the picosecond and femtosecond systems have at least one Light Conversion OPA.

While originally the desire was that the increased repetition rate and monochromator rejection would mean that active chopping was not required, two Thorlabs Chopping systems[161] were added to the infrared lasers to provide active scatter rejection. These are similar models to those used on the other laser systems, though fitted with a different blade arrangement.

The two infrared lines have motorized delay stages from Thorlabs[162]. Each stage provides 50 mm of travel with a resolution of 0.1 micrometers.

A Horiba iHR 320 monochromator[163] was selected with gratings for mid infrared, near infrared, and visible light. This monochromator is from the same company as the smaller monochromators on the other laser systems.

The detectors primarily used are a combination Mercury-Cadmium-Telluride (MCT) detector and Indium-Antimonide (InSb) detector for infrared light and a Hammamatsu photomultiplier tube (PMT) for visible light. Fast photodiodes can also be used, though are not commonly required. The signals generated from these detectors are read by a Gage Compuscope data acquisition card[164]. This differs from the other laser systems and is the only required daemon that is from a new manufacturer entirely. This card was chosen for its acquisition speed and available features such as on board averaging. It has four physical channels that can be measured simultaneously in parallel.

8.3 Implementing Daemons

Much of the hardware arrives before the core upstream laser is installed on the table, since the upstream laser components are custom ordered and require professional installation. As hardware arrives, `yaq` daemons can be written and tested *before* the hardware is installed on the laser table. While in many cases much of the necessary commands can be surmised from hardware documentation, the daemon as a whole must be tested thoroughly. `yaq` makes testing the interface easy, as there are limited, well documented, ways to interface with the hardware. This approach means that time is more efficiently spent dealing with the smaller hardware before the bigger hardware is ready. Thus, once the upstream lasers arrive, installing the full system can proceed more rapidly.

Much of the hardware was intentionally chosen to be similar to hardware with existing `yaq` daemon support. The delays were a different model, but use the same protocol and the same command set. The configuration of having a multi-channel control box meant that some details which were consistent for all previously seen Thorlabs hardware were not quite identical, so a small amount of working out configuration and proper communication was required. When first installed, there were some recurring issues relating to the homing direction being reversed from expected, but a combination of restoring known settings using Thorlabs provided utilities and consistently using the hardware mean that these problems no longer occur.

The Light Conversion hardware uses largely the same protocol as the previously implemented OPAs. Our control system does not actually use Light Conversion's mapping of desired wavelength onto motor positions. We chose to implement our own mapping because it provides increased control over the tuning operations and allows for consistent control over OPAs that are not Light Conversion products (including OPA400, our custom built OPA). As such, the interface we require from Light Conversion is the one to directly control each motor position as well as the shutters. One feature of the new Light Conversion OPAs that previous systems did not use is the ability to have discrete motor positions labeled. Light Conversion's description of discrete motor positions was an inspiring factor in the creation of the `is-discrete` `yaq` trait, however it had not actually been required until the new laser system was installed. The discrete motors are used for binary settings such as inclusion or exclusion of a wave plate and for multi-position motors such filter wheels providing a bandpass of allowed light. The inclusion

of discrete motors incurred changes to not only the `yaqd-attune`[165] daemon, but also the upstream Attune library itself to manage hardware that is fundamentally discrete as part of a tuning curve.

The Horiba Monochromator was largely similar to the existing monochromators in terms of its control interface. However, the new monochromator has additional features including motorized mirrors for selecting input and output slits and motorized slits. The communication protocol for these additions had to be determined. Unfortunately, Horiba does not provide easy access to control the hardware from external programs. Prior to `yaq`, PyCMDS used a generated Python file to interface with a dynamically linked library and provide access to the hardware. This method did not work when the laboratory machine was upgraded from Windows 7 to Windows 10. This incompatibility was one of the driving factors in the inception of `yaq`, and made the Horiba MicroHR daemon one of the first daemons implemented. The direct communication protocol was reverse engineered by observing the USB traffic when the device was instructed to do certain tasks such as initializing or setting the wavelength. This process had to be repeated for the additional features of the new monochromator.

There were two new daemons that had to be implemented, the Gage Compuscope daemon and a daemon for an integrated array detector located inside of the ORPHEUS-HP. While some work can be done using the Gage acquisition card alone, its primary use-case is in a tightly integrated system, where trigger signals directly from the upstream laser are important and signals are required to ensure correct interpretation of the data collected. While some of this could be done with standalone pulse generators, ultimately it is challenging to replicate a full laser system sufficiently to implement the core detection code. The chopping scheme has a significant tight coupling with the detection electronics. While the NI data acquisition cards used by the previous systems had sufficient inputs that both choppers could use a standalone channel for their digital input, the Gage board selected is limited to four channels. Thus some additional electronics were added to do weighted addition of the digital signals to be binned and read by a single analog input for the Gage system. Additionally, the chopping for the previous systems was synchronized to the repetition rate of the laser such that only one or two laser shots passed through each blade section. At 100 kHz, this is impractical given limitations of the rotation of the chopper wheels, and comparison to the beam size. Thus instead, the chopping scheme for this table is performed asynchronously from the trigger signal. The phase of the chopper is read, and multiple shots pass for each blade segment. The on board averaging further complicates chopping, requiring that all

shots in an averaged segment have the same chopper phase to be meaningful. Thus, segments with impure chopper phase are omitted from further calculation.

The second new daemon, for an RGB Photonics Qmini spectrometer[166], also had to wait for the upstream laser to be installed, since it is integrated into one of the OPAs. This detector is functionally similar to the previously implemented Ocean Optics daemons[167], though not identical. RGB Photonics provided a python library for interfacing with the hardware, though it was not distributed and therefore was included in the daemon's source repository.

8.4 Assemble the Instrument

Ultimately, the laser system cannot be fully installed until the upstream lasers are installed. This step, which is performed by a technician from Light Conversion, is the first big step towards a working system. While individual daemons may be ready, there is no substitute for first light on the new laser table. Once the laser is installed, downstream hardware can be installed. In Waldo, the first hardware encountered outside of the OPA for the infrared light is the chopper wheels. These are positioned so that the sensor is directly opposite of the laser path. Following that, there are focusing optics to ensure collimation and reasonable beam size through the remaining optics. Other than alignment mirrors, the next hardware encountered is the delay stage. Following the delay stage are focusing optics for the sample and finally focussing optics into the monochromator. The visible light beam path is comparatively simple, with no chopper and a non-motorized delay which allows for small manual adjustments. The visible line sets the "standard", such that the infrared lines must overlap in time with it, rather than attempting to adjust all three beams to some arbitrary zero mark.

While the system is being built, there are often lasers that are not as fully enclosed as we would ordinarily like. Extra precautions must be taken to ensure that beams are blocked and not reflecting at dangerous angles. Once the design settles a proper enclosure for the laser system is required. This enclosure not only protects from accidental stray laser light, but also provides a controlled environment that can be pumped with dry air to avoid effects of water molecules in the air.

Once the hardware is installed, the promise of `yaq` is that it makes using the existing orchestration ecosystem easy and no additional work is required to make it function. Waldo started by using `yaqc-cmds`, as that was the standard orchestration software for the other two systems at the time. A summary of installed `yaq` daemons on the Waldo laser system is provided in Table 8.1. Waldo was, however, the first system to regularly use the newly implemented Bluesky interfaces for collecting experimental data. There were a few bumps along the way, bugs that needed correcting and issues with memory management. However, the Bluesky codebase is used by many more scientists and is separable into more modular systems which makes for more rapid development to address any bugs that do exist.

host	port	kind	name
127.0.0.1	39011	thorlabs-bsc203	d1_stage
127.0.0.1	39012	thorlabs-bsc203	d2_stage
127.0.0.1	38601	lightcon-topas4-shutter	twin1_shutter
127.0.0.1	38602	lightcon-topas4-shutter	twin2_shutter
127.0.0.1	38603	lightcon-topas4-shutter	hp_shutter
127.0.0.1	38701	lightcon-topas4-motor	hp_Delay_1
127.0.0.1	38702	lightcon-topas4-motor	hp_Crystal_1
127.0.0.1	38703	lightcon-topas4-motor	hp_Delay_2
127.0.0.1	38704	lightcon-topas4-motor	hp_Crystal_2
127.0.0.1	38705	lightcon-topas4-motor	hp_SHG_Crystal
127.0.0.1	38706	lightcon-topas4-motor	hp_RP5_Stage
127.0.0.1	38707	lightcon-topas4-motor	hp_WS_Wheel_1
127.0.0.1	38708	lightcon-topas4-motor	hp_RP6_Stage
127.0.0.1	38709	lightcon-topas4-motor	hp_Crystal_Stage_2
127.0.0.1	38710	lightcon-topas4-motor	hp_WS_Wheel_2
127.0.0.1	38801	lightcon-topas4-motor	twin1_Delay_1
127.0.0.1	38802	lightcon-topas4-motor	twin1_Crystal_1
127.0.0.1	38803	lightcon-topas4-motor	twin1_Delay_2
127.0.0.1	38804	lightcon-topas4-motor	twin1_Crystal_2
127.0.0.1	38805	lightcon-topas4-motor	twin1_RP_Stage
127.0.0.1	38806	lightcon-topas4-motor	twin1_DFG_CS_1
127.0.0.1	38901	lightcon-topas4-motor	twin2_Delay_1
127.0.0.1	38902	lightcon-topas4-motor	twin2_Crystal_1
127.0.0.1	38903	lightcon-topas4-motor	twin2_Delay_2
127.0.0.1	38904	lightcon-topas4-motor	twin2_Crystal_2
127.0.0.1	38905	lightcon-topas4-motor	twin2_SHG_Crystal
127.0.0.1	38906	lightcon-topas4-motor	twin2_RP_Stage
127.0.0.1	38907	lightcon-topas4-motor	twin2_DFG_CS_1
127.0.0.1	38001	attune	hp
127.0.0.1	38002	attune	twin1
127.0.0.1	38003	attune	twin2
127.0.0.1	39876	horiba-ihr320	mono
127.0.0.1	39977	rgb-qmini	orpheus_hp_qmini
127.0.0.1	39001	attune-delay	d1
127.0.0.1	39002	attune-delay	d2
127.0.0.1	39003	gage-compuscope	daq
127.0.0.1	38911	thorlabs-pm-triggered	thorlabs_pm100d
127.0.0.1	38401	thorlabs-ell18	hp_FH_IDL
127.0.0.1	38500	wright-stepper-box	act
127.0.0.1	38501	wright-stepper-box	pm1
127.0.0.1	38502	wright-stepper-box	pm2
127.0.0.1	38503	wright-stepper-box	pmtest
127.0.0.1	36000	system-monitor	massive
127.0.0.1	38510	newport-conex-agp	filter1_angle
127.0.0.1	38010	attune	filter1

Table 8.1: Summary of installed daemons on Waldo

8.5 Conclusions

Building custom instrument is never "done". There are always new ideas of new experiments that require some additional aspect of control. Some experiments are temporary, providing proof of concept for some potential future direction. In the future, there are already plans to expand the list of daemons that are able to control aspects of Waldo. The upstream laser has variable repetition rate, which could be controlled by a daemon, allowing acquisitions at differing rates to occur more easily. As a related change, the desired frequency of the choppers is an important factor for experiments with differing repetition rates, so a daemon to control the choppers is also relevant. Finally, it could be useful to put the upstream laser on standby at the end of a queue. Shutters can already be used to limit unwanted light from potentially damaging samples, but if experiments are completed, there is no need for the upstream laser to be on.

In all, `yaq` made for an easier implementation of a new laser system. Prior to `yaq`, it was much more difficult to implement the hardware interfaces, often requiring all of the hardware to be present to be able to debug a single piece of hardware.

Part IV

Appendix

Appendix A

Glossary

Glossary

Avro Protocol (avpr) A complete listing of all messages suported by an avro RPC.

Bluesky (Library) The central Library of the Bluesky Project which provides an engine for specification and orchestration of scientific experiments.

Bluesky (Project) A collaboration primarily consisting of members of National Labs which provides libraries in Python do perform experimental science.

bluesky-cmds Graphical application for interacting with a Bluesky Queueserver.

Bluesky-in-a-box A Docker compose application which runs all services needed to run a Queueserver.

CLI Command Line Interface, an application that is run in a text-based terminal.

CMDS Coherent Multi-Dimensional Spectroscopy.

daemon A small background program that performs a single task.

Docker A system to run applications in a reproducible way.

OPA Optical Parametric Amplifier, a device which changes the wavelength of light by adjusting motors controlling crystal and grating angles and delay times.

plan A Python function which defines a series of steps of an experimental procedure.

property (yaq) A collection of messages which provide access to one logical value, such as getter and setter, along with associated messages such as units and options.

PyCMDS A monolithic Python program for collecting CMDS data.

Queueserver A background service which runs Bluesky plans in a queue.

RPC Remote Procedure Call.

trait (yaq) A collection of messages, config, state, and properties that provides consistent interaction with multiple daemons.

UUID Universally Unique Identifier.

variadic cycle A repeating cycle of parameters which are logically grouped in sets.

`wright-plans` Bluesky Plans built for the Wright Group.

yaq-traits A command line application to generate Avro Protocols from more human friendly TOML files.

yaqc-cmds Pronounced "Yak C Commands". A custom program for collecting CMDS data using `yaq` to communicate with hardware.

yaqc-qtpy An application to interact with `yaq` daemons which is built on traits and serves engineering purposes.

yaqd-control A command line application to manage `yaq` daemons.

Appendix B

Useful Scripts

B.1 Introduction

This appendix contains an assortment of scripts which are useful in the Wright Group.

Many of these scripts are in active use on a regular basis, with only minor modifications for inclusion here. Such modifications include reformatting the code for a line length of 79 characters, removing commented out or otherwise unreachable code, and adjusting some of the comments.

B.2 WrightTools

B.2.1 Quick 2D

```
import WrightTools as wt
import matplotlib.pyplot as plt
import pathlib

fil_dir = pathlib.Path(r"C:\Users\Wright\path\to\data\folder")
data = wt.open(fil_dir / "primary.wt5")
data.convert("wn")

wt.artists.quick2D(
    data, xaxis=0, yaxis=1, channel="daq_ai0_diff_abcd", pixelated=False
)
plt.show()
data.print_tree()
```

B.3 Attune

B.3.1 Undo Tune

```
import attune

opa = "w2"
instr = attune.restore(
    opa, time="2022-08-19T19:45:47.205570-05:00"
)  # , can check date format of data with d.attrs["created"] on your data object
```

B.3.2 Intensity workup

```
import pathlib
import WrightTools as wt
import attune
import numpy as np

folder = pathlib.Path(r"C:\Users\John\path\to\folder")


# data.wt5 (yaqc) or primary.wt5 (bluesky)
data_filename = "primary.wt5"

# only matters if using ingaas.
# Should be wavelengths (yaqc) or array_wavelengths (bluesky)
array_axis = "array_wavelengths"


datapath = folder / data_filename
opa = "w1"
channel = "daq_sample"
arrangement = "non-sh-sh-idl"
tune = "mixer_2"
apply = False


d = wt.open(datapath)
d.transform(
    f"{opa}_points", f"{opa}_{tune}_points"
)
# The `_points` arrays represent the squeezed versions of the arrays
# Some methods require this.


instr = attune.load(opa)

if channel == "ingaas" or channel == "array_ingaas":
    d.transform(f"{opa}_points", f"{opa}_{tune}_points", array_axis)
    # second argument might be "w1_Delay_2", "w1_Delay_2_centers"
    # data.print_tree()
    d.level(channel, -1, 5)  # leveling along array signal before moment

    d.moment(
        axis=array_axis,
        channel=channel,
        moment=0,
```

```
        resultant=wt.kit.joint_shape(d[opa], d[f"{opa}_{tune}_points"]),
    )
    channel = -1
    d.transform(
        f"{opa}_points", f"{opa}_{tune}_points"
    )


instr2 = attune.intensity(
    data=d,
    channel=channel,
    arrangement=arrangement,
    tune=tune,
    instrument=instr,
    gtol=0.01,
    autosave=apply,
    save_directory=datapath.parent,
)


# This section sets the instrument as the active instrument for the OPA
if apply:
    import yaqc

    c = yaqc.Client(39300 + int(opa[1]))
    c.set_instrument(instr2.as_dict())
```

B.3.3 Spectral Delay Correction workup

```
import WrightTools as wt
import attune
import pathlib

data_dir = pathlib.Path(
    r"C:\Users\John\path\to\folder"
)
data = wt.open(data_dir / "primary.wt5")

opa = "w2"
delay = "d1"
channel = "daq_signal_mean"
arrangement = opa
tune = "non-sh-non-sig"
curve = f"autonomic_{delay}"
store = False
autosave = store

data.transform(f"{opa}_points", f"{delay}_points")
data.print_tree()
data.level(1, 1, -4)

instr = attune.intensity(
    data=data,
    channel=channel,
    arrangement=arrangement,
    level=False,
    tune=tune,
    ltol=0.01,
    gtol=0.01,
    # s=1,
    # k=1,
    autosave=autosave,
    save_directory=data_dir,
)

# This should be fixed in the workup method itself
instr.arrangements[opa][tune]._dep_units = data[f"{delay}_points"].units

instr = attune.update_merge(attune.load(arrangment), instr)

if store:
    # Requires restart of the associated daemon to take effect
    attune.store(instr)
```

B.3.4 Setpoint workup

```
import pathlib
import WrightTools as wt
import attune
import numpy as np

folder = pathlib.Path(
    r"C:\Users\john\path\to\folder"
)

# data.wt5 (yaqc) or primary.wt5 (bluesky)
data_filename = "primary.wt5"

datapath = folder / data_filename
opa = "w2"

# ingaas (yaqc) or array_ingaas (bluesky)
channel = "array_ingaas"

arrangement = "non-non-non-sig"
tune = "crystal_2"
apply = True
simple_filter = True

# "wavelengths" in yaqc-cmds, array_wavelengths in bluesky
array_axis = "array_wavelengths"

d = wt.open(datapath)

instr = attune.load(opa)

if channel == "ingaas":
    d.transform(f"{opa}_points", f"{opa}_{tune}_points", "wavelengths")
    # second argument might be "w1_Delay_2", "w1_Delay_2_centers"
    # data.print_tree()
    d.level("ingaas", -1, 5)  # leveling along array signal before moment

    if simple_filter == True:
        if arrangement == "non-non-non-sig":
            for i, w2_color in enumerate(d[f"{opa}_points"][:, 0]):
                for k, array_pixel_color in enumerate(
                    d["wavelengths"][i, 0, :]
                ):
                    if (
                        array_pixel_color >= w2_color + 50
```

```
                        or array_pixel_color <= w2_color - 50
                    ):
                        for j, signal in enumerate(d.ingaas[i, :, k]):
                            d.ingaas[i, j, k] = np.nan
if channel == "array_ingaas":
    d.transform(f"{opa}_{tune}_points", f"{opa}_points", "array_wavelengths")
    # second argument might be "w1_Delay_2", "w1_Delay_2_centers"
    # data.print_tree()
    d.level("array_ingaas", -1, 5)  # leveling along array signal before moment

    if simple_filter == True:
        if arrangement == "non-non-non-sig":
            d[channel][
                abs(d[array_axis][:] - d[f"{opa}_points"][:]) > 50
            ] = np.nan


d.moment(
    axis=array_axis,
    channel=channel,
    moment=0,
    resultant=wt.kit.joint_shape(d[opa], d[f"{opa}_{tune}_points"]),
)
d.moment(
    axis=array_axis,
    channel=channel,
    moment=1,
    resultant=wt.kit.joint_shape(d[opa], d[f"{opa}_{tune}_points"]),
)
channel = -1
gtol = 0.005
d.channels[-1][d.channels[-2] < gtol * d.channels[-2].max()] = np.nan

d.transform(f"{opa}_points", f"{opa}_{tune}_points")
d.channels[-1].clip(min=d[f"{opa}"].min() - 1000, max=d[f"{opa}"].max() + 1000)
d.channels[-1].null = d[
    array_axis
].min()  # setting effective zero to be wa min

instr2 = attune.setpoint(
    data=d,
    channel=channel,
    arrangement=arrangement,
    tune=tune,
    instrument=instr,
    autosave=apply,
    save_directory=datapath.parent,
```

```
    s=1,
    k=3,
)

if apply:
    import yaqc

    c = yaqc.Client(39300 + int(opa[1]))
    c.set_instrument(instr2.as_dict())
```

B.3.5 Tune Test workup

```
import numpy as np
import pathlib

import attune
import WrightTools as wt
import matplotlib.pyplot as plt


folder = pathlib.Path(
    r"C:\Users\john\path\to\folder"
)
datapath = folder / "primary.wt5"

opa = "w1"
channel = "daq_signal_mean"
arrangement = "non-sh-sh-idl"
filtering = False
apply = False
sub_baseline = False
save_plot = apply

d = wt.open(datapath)


instr = attune.load(opa)

if channel == "array_ingaas":
    d.level(channel, -1, 5)  # leveling along array signal before moment

    if sub_baseline:
        baseline = wt.open(baseline_datapath)
        baseline.level("ingaas", -1, 5)
        d[channel][:] -= baseline[channel][:]

    if filtering == True:

        if arrangement == "non-non-non-sig":
            for i, w2_color in enumerate(d[f"{opa}_points"][:, 0]):
                if w2_color > 1490:
                    for j, array_pixel_color in enumerate(
                        d["wavelengths"][i, :]
                    ):
                        if array_pixel_color > np.min([1700, w2_color + 40]):
                            d.ingaas[i, j] = np.nan
```

```
                if w2_color > 1580:
                    for j, array_pixel_color in enumerate(
                        d["wavelengths"][i, :]
                    ):
                        if array_pixel_color < np.min([1700, w2_color - 20]):
                            d.ingaas[i, j] = np.nan

        if arrangement == "non-non-sh-idl":
            for i, w2_color in enumerate(d[f"{opa}_points"][:, 0]):
                if w2_color > 1120:
                    for j, array_pixel_color in enumerate(
                        d["wavelengths"][i, :]
                    ):
                        if (
                            1080
                            < array_pixel_color
                            < np.min([1105, w2_color - 10])
                        ):
                            d.ingaas[i, j] = np.nan
    d.transform(f"{opa}_points", f"array_wavelengths-{opa}")

else:
    d.transform(f"{opa}_points", f"wm-{opa}")

instr2 = attune.tune_test(
    data=d,
    channel=channel,
    arrangement=arrangement,
    instrument=instr,
    gtol=0.005,
    ltol=0.00,
    autosave=apply,
    save_directory=datapath.parent,
    s=3,  # "smoothness"
    k=1,  # number of nodes
)

if apply:
    import yaqc

    c = yaqc.Client(39300 + int(opa[1]))
    c.set_instrument(instr2.as_dict())
```

B.3.6 Holistic workup

```
import pathlib
import attune
import WrightTools as wt

data_dir = pathlib.Path(
    r"C:\Users\john\bluesky-cmds-data\folder"
)

# data.wt5 (yaqc) or primary.wt5 (bluesky)
data_filename = "primary.wt5"

# wavelengths (yaqc) or array_wavelengths (bluesky)
array_axis = "array_wavelengths"

# ingaas (yaqc) or array_ingaas (bluesky)
channel_name = "array_ingaas"

opa = "w2"
apply = True
old_instrument = attune.load(opa)

data = wt.open(data_dir / data_filename)

data.transform(f"{opa}_crystal_1", f"{opa}_delay_1", array_axis)
data.level(channel_name, -1, 5)

new_instrument = attune.holistic(
    data=data,
    channels=channel_name,
    arrangement="non-non-non-sig",
    tunes=["crystal_1", "delay_1"],
    instrument=old_instrument,
    save_directory=data_dir,
    autosave=apply,
    level=False,
    gtol=0.04,
)


if apply:
    import yaqc

    c = yaqc.Client(39300 + int(opa[1]))
    c.set_instrument(new_instrument.as_dict())
```

B.4 Hardware Communication

B.4.1 NI DAQ Round Robin

This script produces a plot that helps to visualize the measurements of the DAQ and make informed choices about pixel index for measuring the state and for determining the phase offset for choppers.

```
import os
import time
import itertools
import collections

import numpy as np

import WrightTools as wt

import matplotlib.pyplot as plt
import matplotlib.patches as patches

import PyDAQmx

conversions_per_second = 1e6  # a property of the DAQ card
shots_per_second = 1100.0  # from laser (ensure OVERestimate)


def round_robin(channels, index, shots=7, plot=True, cycle="rgbk", freq=False):
    """Collect and plot data from a NI DAQ card.

    Parameters
    ----------
    channels : dictionary
        Dictionary of {name, index}.
    index : integer
        Plot a black vertial line at this index for every shot.
    shots : integer (optional)
        Number of shots to acquire. Default is 7.
    plot : boolean (optional)
        Toggle plotting. Default is True.
    cycle : iterable (optional)
        Color cycle for shot highlighting in background. Default is rgbk.
    """
    # coerce inputs
    channels = collections.OrderedDict(channels)
    task_handle = PyDAQmx.TaskHandle()
    read = PyDAQmx.int32()
    num_channels = len(channels)
    virtual_samples = int(
```

```
        conversions_per_second / (shots_per_second * num_channels)
    )
    us_per_virtual_sample = (1.0 / conversions_per_second) * 1e6
    try:
        # task
        PyDAQmx.DAQmxCreateTask("", PyDAQmx.byref(task_handle))
        # create global channels
        total_virtual_channels = 0
        for _ in range(virtual_samples):
            for channel in channels.keys():
                channel_name = "channel_" + str(total_virtual_channels).zfill(
                    2
                )
                PyDAQmx.DAQmxCreateAIVoltageChan(
                    task_handle,  # task handle
                    "Dev1/ai%i" % channel,  # physical chanel
                    channel_name,  # name to assign to channel (must be unique)
                    PyDAQmx.DAQmx_Val_Diff,  # the input terminal configuration
                    -10,
                    10,  # minVal, maxVal
                    PyDAQmx.DAQmx_Val_Volts,  # units
                    None,
                )  # custom scale
                total_virtual_channels += 1
        # timing
        PyDAQmx.DAQmxCfgSampClkTiming(
            task_handle,  # task handle
            "/Dev1/PFI0",  # sorce terminal
            1000.0,  # sampling rate (samples per second per channel) (float 64)
            PyDAQmx.DAQmx_Val_Rising,  # acquire samples on the rising edges
            PyDAQmx.DAQmx_Val_FiniteSamps,  # acquire a finite number of samples
            shots,
        )  # number of samples per global channel (unsigned integer 64)
    except PyDAQmx.DAQError as err:
        print("DAQmx Error: %s" % err)
        PyDAQmx.DAQmxStopTask(task_handle)
        PyDAQmx.DAQmxClearTask(task_handle)
        return
    # get data
    samples = np.zeros(
        shots * virtual_samples * num_channels, dtype=np.float64
    )
    for _ in range(1):
        try:
            start_time = time.time()
            PyDAQmx.DAQmxStartTask(task_handle)
            PyDAQmx.DAQmxReadAnalogF64(
```

```
                task_handle,  # task handle
                shots,  # number of samples per global channel (uint64)
                10.0,  # timeout (seconds) for each read operation
                # fill mode (specifies if the samples are interleaved)
                PyDAQmx.DAQmx_Val_GroupByScanNumber,
                samples,  # read array
                len(
                    samples
                ),  # size of the array, in samples, into which samples are read
                PyDAQmx.byref(read),  # reference of thread
                None,
            )  # reserved by NI, pass NULL (?)
            PyDAQmx.DAQmxStopTask(task_handle)
            # create 2D data array
            out = np.copy(samples)
            out.shape = (shots, virtual_samples, num_channels)
            print(
                "Acquired %d shots in %f seconds"
                % (read.value, time.time() - start_time)
            )
        except PyDAQmx.DAQError as err:
            print("DAQmx Error: %s" % err)
            PyDAQmx.DAQmxStopTask(task_handle)
            PyDAQmx.DAQmxClearTask(task_handle)
            return
    # plot
    def get_plot_arrays(channel):
        y = out[:, :, channel].flatten()
        x = np.zeros(len(y))
        for i in range(shots * virtual_samples):
            offset = us_per_virtual_sample * channel
            shot_count, sample_count = divmod(i, virtual_samples)
            x[i] = (
                offset
                + shot_count * 1000
                + sample_count * (us_per_virtual_sample * num_channels)
            )
        mag = max(-y.min(), y.max())
        y /= mag
        return x, y

    if plot:
        # data
        fig = plt.figure(figsize=(10, 4))
        ax = plt.gca()

        for i, item in enumerate(channels.items()):
```

```
            plt.plot(*get_plot_arrays(i), lw=4, alpha=0.5, label=item[1])
        # boxes
        cs = itertools.cycle(cycle)
        for i in range(shots):
            rect = patches.Rectangle(
                (i * 1000, -1.1),
                1000,
                2.2,
                zorder=0,
                facecolor=next(cs),
                edgecolor="none",
                alpha=0.1,
            )
            ax.add_patch(rect)
        # lines
        for i in range(shots):
            plt.axvline(i * 1000 + index, c="k")
        plt.legend(loc="upper left", bbox_to_anchor=(1.04, 1))
        ax.set_xlim(0, shots * 1000)
        ax.set_ylim(-1.1, 1.1)
        ax.set_xlabel(r"$\mu$s", fontsize=18)
    if freq:
        fig, gs = wt.artists.create_figure(
            width=10, nrows=3, default_aspect=0.25
        )

        for i, item in enumerate(channels.items()):
            ax = plt.subplot(gs[i])
            x, y = get_plot_arrays(i)
            x *= 1e-6
            x2, y2 = wt.kit.fft(x, y)
            y2 = np.abs(y2)
            y2 /= y2.max()
            ax.loglog(x2, y2, alpha=0.5, label=item[1])
            ax.set_xlim(100, 10000)
            plt.legend(loc="upper left", bbox_to_anchor=(1.04, 1))
            ax.grid()
        wt.artists.set_fig_labels(xlabel="frequency(Hz)", ylabel="abs")

    # finish
    if task_handle:
        PyDAQmx.DAQmxStopTask(task_handle)
        PyDAQmx.DAQmxClearTask(task_handle)
    return out


channels = collections.OrderedDict()
```

```
# channel numbers here are the physical channels
channels[0] = "signal"
channels[4] = 'chopper 1'
channels[5] = "chopper 2"  # 1=open, 0=blocked
channels[3] = 'Pyro 2'
channels[2] = 'Pyro 1'

round_robin(channels, index=590, shots=10, plot=True, freq=False)
plt.show()
```

B.4.2 InGaAs Plotting

```
import numpy as np
import matplotlib.pyplot as plt
import yaqc

ingaas = yaqc.Client(38989)
fig, ax = plt.subplots()
fig.subplots_adjust(bottom=0.2)
t = np.arange(-2.0, 2.0, 0.001)
(l,) = ax.plot(t, np.zeros_like(t), lw=2)


def submit():
    try:
        ingaas.measure()
        l.set_xdata(ingaas.get_mappings()["wavelengths"])
        l.set_ydata(ingaas.get_measured()["ingaas"])
    except ConnectionError:
        pass
    ax.relim()
    ax.autoscale_view()
    plt.draw()


timer = fig.canvas.new_timer(interval=200)


@timer.add_callback
def update():
    submit()


timer.start()
plt.show()
```

B.5 Bluesky Run Engine

B.5.1 Run a Run Engine, connecting to `bluesky-in-a-box`

This script allows you to run an instance of a Bluesky Run Engine, with the devices read from HAPPI and the readings being sent to the WT5 writer of `bluesky-in-a-box` .

The recommended usage of this script is to run as an interactive python shell:

```
$ python -i bluesky_play.py
[Errno 111] Connection refused
Adding d0
Adding d1
Adding d2
Adding daq
YAQDevice() takes no arguments
Adding nd1
Adding nd2
[Errno 111] Connection refused
Adding w1
Adding w1_shutter
Adding w2
Adding w2_shutter
Adding wa
Adding wm
>>> RE(count([daq], 10, 1))
```

(2.1)

```
import configparser
import os
import socket
import happi

from wright_plans import (
    list_scan_wp,
    rel_list_scan_wp,
    list_grid_scan_wp,
    rel_list_grid_scan_wp,
    scan_wp,
    grid_scan_wp,
    rel_grid_scan_wp,
    rel_scan_wp,
)
from wright_plans.attune import (
    motortune,
    run_tune_test,
    run_intensity,
    run_setpoint,
    run_holistic,
```

```
)

import bluesky
from bluesky.plans import count
from bluesky.plan_stubs import mv, sleep
from bluesky.preprocessors import baseline_decorator
from bluesky.protocols import Movable
from bluesky.callbacks.zmq import Publisher
from bluesky.callbacks.best_effort import BestEffortCallback

happi_cfg = configparser.ConfigParser()
happi_cfg.read(os.getenv("HAPPI_CFG"))
happi_client = happi.Client(
    database=happi.backends.backend(happi_cfg["DEFAULT"]["path"])
)

# Movable devices to read baseline before and after scans
movables = []

# Cache of describe keys used to determine if sub-devices should be
# ejected from the namespace
all_device_keys = {}

for device in happi_client.all_items:
    # Skip devices marked inactive in happi
    if not device.active:
        continue
    try:
        dev = happi.from_container(device)
        dev_keys = set(dev.describe().keys())
        for prev_dev_name, prev_dev_keys in all_device_keys.items():
            # Do not add this device (break for loop, skip else clause)
            # if all of my keys are in another device
            if dev_keys.issubset(prev_dev_keys):
                break
            # Eject (set to None) a subdevice previously added
            if prev_dev_keys.issubset(dev_keys):
                vars()[prev_dev_name] = None
                del all_device_keys[prev_dev_name]
        else:
            # Only runs if this is a new device (no break above)
            print("Adding", device.name)
            # Add to namespace
            vars()[device.name] = dev
            # Add keys to cache
            all_device_keys[dev.name] = dev_keys
            # Add to movables list if Movable
```

```
            if isinstance(dev, Movable):
                movables.append(dev)
    except Exception as e:
        print(e)

dev = None
prev_dev = None

# Wrap all of the plans with baseline which reads movables before and after
list_scan_wp = baseline_decorator(movables)(list_scan_wp)
rel_list_scan_wp = baseline_decorator(movables)(rel_list_scan_wp)
list_grid_scan_wp = baseline_decorator(movables)(list_grid_scan_wp)
rel_list_grid_scan_wp = baseline_decorator(movables)(rel_list_grid_scan_wp)
scan_wp = baseline_decorator(movables)(scan_wp)
grid_scan_wp = baseline_decorator(movables)(grid_scan_wp)
rel_grid_scan_wp = baseline_decorator(movables)(rel_grid_scan_wp)
rel_scan_wp = baseline_decorator(movables)(rel_scan_wp)
motortune = baseline_decorator(movables)(motortune)
run_tune_test = baseline_decorator(movables)(run_tune_test)
run_intensity = baseline_decorator(movables)(run_intensity)
run_setpoint = baseline_decorator(movables)(run_setpoint)
run_holistic = baseline_decorator(movables)(run_holistic)
count = baseline_decorator(movables)(count)

RE = bluesky.RunEngine()
publisher = Publisher("127.0.0.1:5567")
bec = BestEffortCallback()
bec.disable_plots()
RE.subscribe(publisher)
RE.subscribe(bec)
```

Appendix C

Simulation

C.1 WrightSim

This section was originally published in Scipy Proceedings 2018 [40]

C.1.1 `WrightSim`: Using PyCUDA to Simulate Multidimensional Spectra

Nonlinear multidimensional spectroscopy (MDS) is a powerful experimental technique used to interrogate complex chemical systems. MDS promises to reveal energetics, dynamics, and coupling features of and between the many quantum-mechanical states that these systems contain. In practice, simulation is typically required to connect measured MDS spectra with these microscopic physical phenomena. We present an open-source Python package, `WrightSim`, designed to simulate MDS. Numerical integration is used to evolve the system as it interacts with several electric fields in the course of a multidimensional experiment. This numerical approach allows `WrightSim` to fully account for finite pulse effects that are commonly ignored. `WrightSim` is made up of modules that can be exchanged to accommodate many different experimental setups. Simulations are defined through a Python interface that is designed to be intuitive for experimentalists and theorists alike. We report several algorithmic improvements that make `WrightSim` faster than previous implementations. We demonstrated the effect of parallelizing the simulation, both with CPU multiprocessing and GPU (CUDA) multithreading. Taken together, algorithmic improvements and parallelization have made `WrightSim` multiple orders of magnitude faster than previous implementations. `WrightSim` represents a large step towards the goal of a fast, accurate, and easy to use general purpose simulation package for multidimensional spectroscopy. To our knowledge, `WrightSim` is the first openly licensed software package for these kinds of simulations. Potential further improvements are discussed.

Simulation, spectroscopy, PyCUDA, numerical integration, Quantum Mechanics, multidimensional

Introduction

Nonlinear multidimensional spectroscopy (MDS) is an increasingly important analytical technique for the analysis of complex chemical material systems. MDS can directly observe fundamental physics that are not possible to record in any other way. With recent advancements in lasers and optics, MDS experiments are becoming routine. Applications of MDS in semiconductor photophysics [25], medicine [168], and other domains [169] are currently being developed. Ultimately, MDS may become a key research tool akin to multidimensional nuclear magnetic resonance spectroscopy. [22]

A generic MDS experiment involves exciting a sample with multiple pulses of light and measuring the magnitude of the sample response (the signal). The dependence of this signal on the properties of the excitation pulses (frequency, delay, fluence, polarization etc.) contains information about the microscopic physics of the material. However, this information cannot be directly "read off" of the spectrum. Instead, MDS practitioners typically compare the measured spectrum with model spectra. A quantitative microscopic model is developed based on this comparison between experiment and theory. Here, we focus on this crucial modeling step. We present a general-purpose simulation package for MDS: `WrightSim`[1].

[1] Source code available at https://github.com/wright-group/WrightSim, released under MIT License.

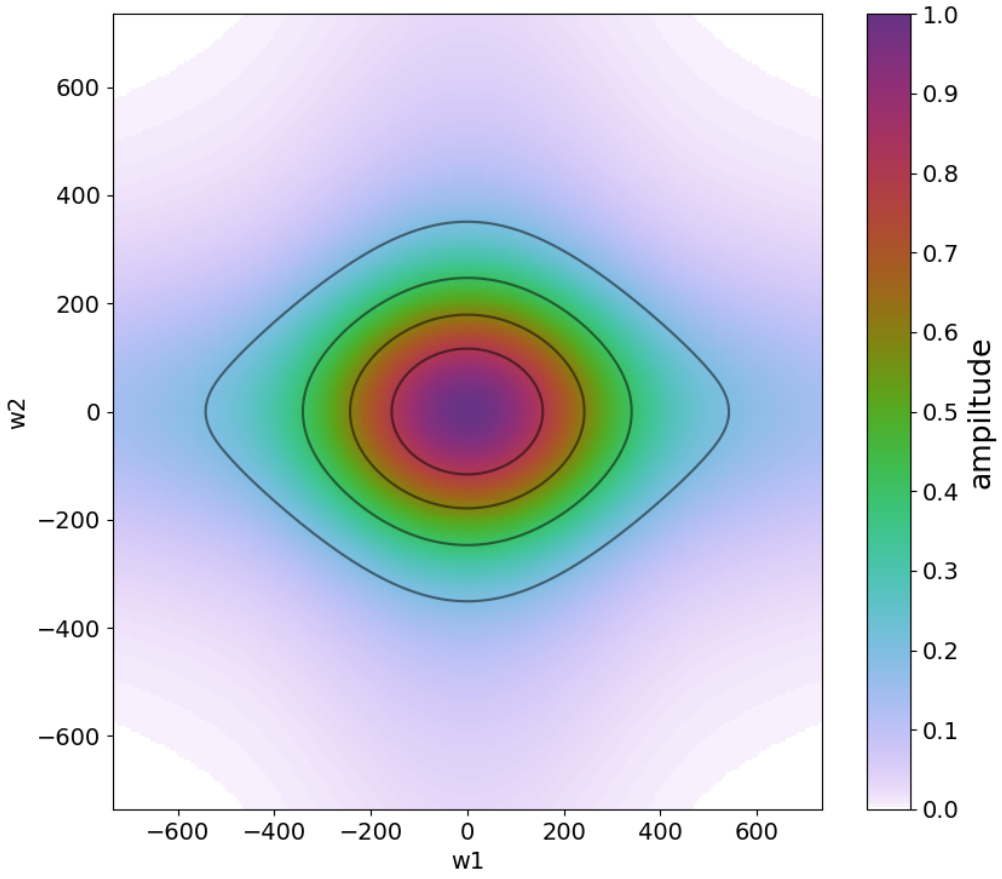

Figure C.1: Simulated spectrum at normalized coordinates

Figure C.1 is a visualization of a spectrum in 2-dimensional frequency-frequency space. The axes are two different frequencies for two separate input electric fields. The system that we have chosen for this simulation is very simple, with a single resonance. The axes are translated such that there is a resonance around 0.0 in both frequencies. This two-dimensional simulation is representative of `WrightSim`'s ability to traverse through many aspects of experimental space. Every conceivable pulse parameter (delay, fluence, frequency, chirp etc.) can become an axis in the simulation.

`WrightSim` is designed with the experimentalist in mind, allowing users to parameterize their simulations in much the same way that they would collect a similar spectrum in the laboratory. `WrightSim` is modular and flexible. It is capable of simulating different kinds of MDS, and it is easy to extend to new kinds.

`WrightSim` uses a numerical integration approach that captures the full interaction between material and electric field without making common limiting assumptions. This approach makes `WrightSim` flexible, accurate, and interpretable. While the numerical approach we use is more accurate, it does demand significantly more computational time. We have focused on performance as a critical component of `WrightSim`. Here we report algorithmic improvements which have significantly decreased computational time (i.e. wall clock time) relative to prior implementations. We also discuss parallelization approaches we have taken, and show how the symmetry of the simulation can be exploited. While nascent, `WrightSim` has already shown itself to be a powerful tool, greatly improving execution time over prior implementation.

A Brief Introduction of Relevant Quantum Mechanics

This introduction is intended to very quickly introduce *what* is being done, but not *why*. If you are interested in a more complete description, please refer to Kohler, Thompson, and Wright. [37]

`WrightSim` uses the density matrix formulation of quantum mechanics. This formulation allows us to describe mixed states (coherences) which are key players in light-matter-interaction and spectroscopy. This involves numerically integrating the Liouville-von Neumann equation [170]. This strategy has been described before [171], so we are brief in our description here.

`WrightSim` calculates multidimensional spectra for a given well-defined Hamiltonian. We do not make common limiting assumptions that allow reduction to analytical expressions. Instead, we propagate all of the relevant density matrix elements, including populations and coherences, in a numerical integration. This package does **not** perform *ab initio* computations. This places `WrightSim` at an intermediate level of theory where the Hamiltonian is known, but accurately computing the corresponding multidimensional spectrum requires complicated numerical analysis.

Now, we focus on one representative experiment and Hamiltonian. In this case, we are simulating the interactions of three electric fields to induce an output electric field. For three fields, there are $3! = 6$ possible time orderings for the pulses to interact and create superpositions or populations in the material system (Figure C.2, columns). Within each time ordering, there are several different pathways (Figure C.2, rows). In total, there are 16 pathways, represented in Figure C.2 as a series of wave mixing energy level (WMEL) diagrams [172]. We are restricting this simulation to have two positive interactions (solid up arrows or dashed down arrows) and one negative interaction (dashed up arrow or solid down arrow). Experimentalists isolate this condition spatially using an aperture. They can isolate the time orderings by introducing delays between pulses. Simulation allows us to fully separate each pathway, leading to

insight into the nature of pathway interference in the total signal line shape.

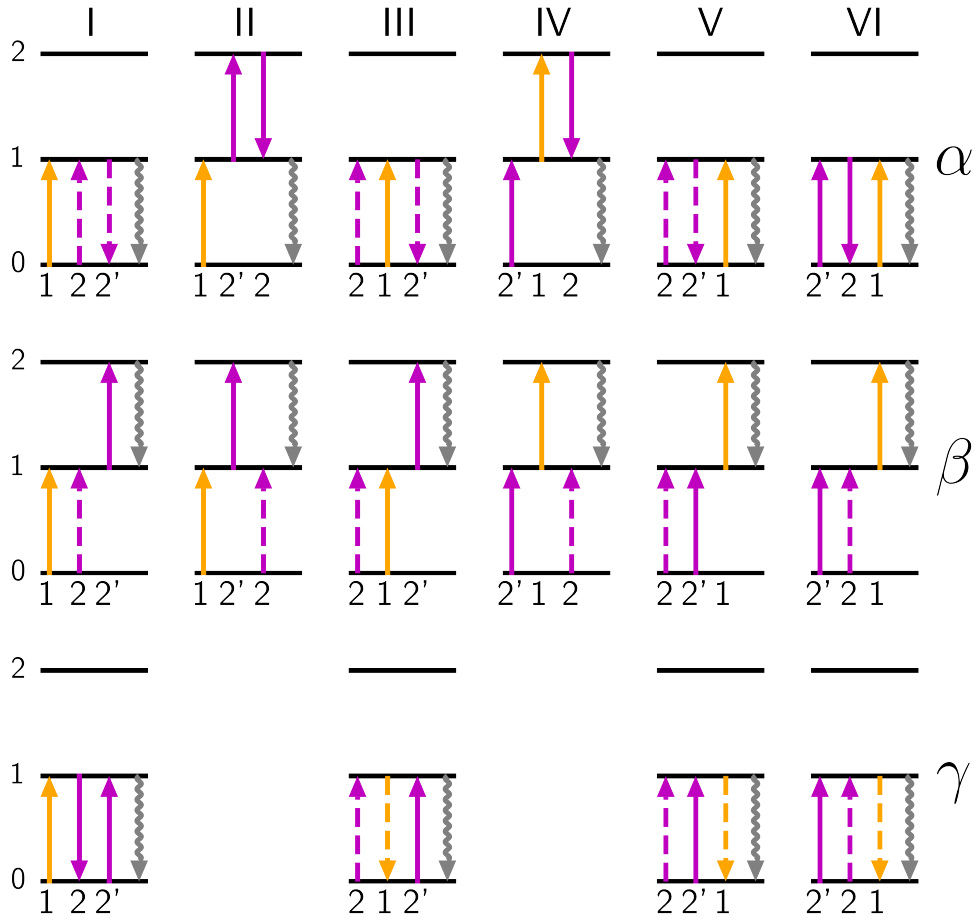

Figure C.2: Independent Liouville pathways simulated. Excitations from ω_1 are in yellow, excitations from $\omega_2 = \omega_{2'}$ are shown in purple. Figure was originally published as Figure 1 of Kohler, Thompson, and Wright [37]

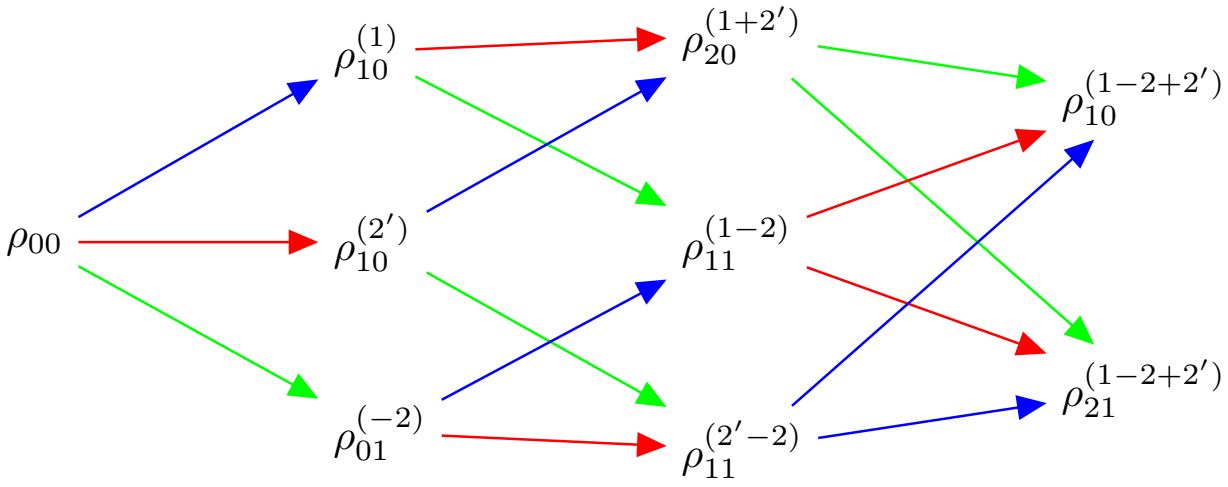

Figure C.3: Finite state automaton of the interactions with the density matrix elements. Matrix elements are denoted by their coherence/population state (the subscript) and the pulses which they have already interacted with (the superscript). Arrows indicate interactions with ω_1 (blue), $\omega_{2'}$ (red), and ω_2 (green). Figure was originally published as Figure S1 of Kohler, Thompson, and Wright [37]

Figure C.3 shows a finite state automaton for the same system as Figure C.2. The nodes are the density matrix elements themselves. All pathways start at the ground state (ρ_{00}). Encoded within each node is both the quantum mechanical state and the fields with which the system has already interacted. Interactions occur along the arrows, which generate density in the resulting state. Here, the fields must each interact exactly once. Output is generated by the rightmost two nodes, which have interacted with all three fields. These nine states represent all possible states which match the criterion described by the process we are simulating.

We take these nine states and collect them into a state density vector, $\overline{\rho}$ (Equation 1.1):

$$\overline{\rho} \equiv \begin{bmatrix} \tilde{\rho}_{00} \\ \tilde{\rho}_{01}^{(-2)} \\ \tilde{\rho}_{10}^{(2')} \\ \tilde{\rho}_{10}^{(1)} \\ \tilde{\rho}_{20}^{(1+2')} \\ \tilde{\rho}_{11}^{(1-2)} \\ \tilde{\rho}_{11}^{(2'-2)} \\ \tilde{\rho}_{10}^{(1-2+2')} \\ \tilde{\rho}_{21}^{(1-2+2')} \end{bmatrix}$$

Next we need to describe the transitions within these states. This is the Hamiltonian matrix. Since we have nine states in our density vector, the Hamiltonian is a nine by nine matrix. To simplify representation, six time dependent variables are defined:

$$\begin{aligned}
A_1 &\equiv \frac{i}{2}\mu_{10}e^{-i\omega_1\tau_1}c_1(t-\tau_1)e^{i(\omega_1-\omega_{10})t} \\
A_2 &\equiv \frac{i}{2}\mu_{10}e^{i\omega_2\tau_2}c_2(t-\tau_2)e^{-i(\omega_2-\omega_{10})t} \\
A_{2'} &\equiv \frac{i}{2}\mu_{10}e^{-i\omega_{2'}\tau_{2'}}c_{2'}(t-\tau_{2'})e^{i(\omega_{2'}-\omega_{10})t} \\
B_1 &\equiv \frac{i}{2}\mu_{21}e^{-i\omega_1\tau_1}c_1(t-\tau_1)e^{i(\omega_1-\omega_{21})t} \\
B_2 &\equiv \frac{i}{2}\mu_{21}e^{i\omega_2\tau_2}c_2(t-\tau_2)e^{-i(\omega_2-\omega_{21})t} \\
B_{2'} &\equiv \frac{i}{2}\mu_{21}e^{-i\omega_{2'}\tau_{2'}}c_{2'}(t-\tau_{2'})e^{i(\omega_{2'}-\omega_{21})t}
\end{aligned}$$

These variables each consist of a constant factor of $\frac{i}{2}$, a dipole moment term ($\mu_{10|21}$), an electric field phase and amplitude (the first exponential term), an envelope function (c, a Gaussian function here), and a final exponential term which captures the resonance dependence. These variables can then be used to populate the matrix:

$$\overline{\overline{Q}} \equiv \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -A_2 & -\Gamma_{10} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ A_{2'} & 0 & -\Gamma_{10} & 0 & 0 & 0 & 0 & 0 & 0 \\ A_1 & 0 & 0 & -\Gamma_{10} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & B_1 & B_{2'} & -\Gamma_{20} & 0 & 0 & 0 & 0 \\ 0 & A_1 & 0 & -A_2 & 0 & -\Gamma_{11} & 0 & 0 & 0 \\ 0 & A_{2'} & -A_2 & 0 & 0 & 0 & -\Gamma_{11} & 0 & 0 \\ 0 & 0 & 0 & 0 & B_2 & -2A_{2'} & -2A_1 & -\Gamma_{10} & 0 \\ 0 & 0 & 0 & 0 & -A_2 & B_{2'} & B_1 & 0 & -\Gamma_{21} \end{bmatrix}$$

The Γ values along the diagonal represent loss terms such as dephasing (loss of coherence) and population relaxation. To isolate a given time ordering, we can simply set the value of elements which do not correspond to that time ordering to zero.

At each time step, the dot product of the matrix with the $\overline{\rho}$ vector is the change in the $\overline{\rho}$ vector to the next time step (when multiplied by the differential). `WrightSim` uses a second order technique (Runge-Kutta) [173] for determining the change in the $\overline{\rho}$ vector. The core of the simulations is to take the $\overline{\rho}$ vector and multiply by the Hamiltonian at each time step (noting that the Hamiltonian is time dependant, as are the electric fields, themselves). This process repeats over a large number of small time steps, and must be performed separately for any change in the inputs (e.g. frequency [ω] or delay[τ]). As a result, the operation is highly parallelizable. The integration is performed in the rotating frame so the number of time steps can be as small as possible.

Usage

`WrightSim` is designed in a modular, extensible manner in order to be friendly to experimentalists and theorists alike. The key steps to running a basic simulation are:

- Define the experimental space
- Select a Hamiltonian for propagation
- Run the scan
- Process the results

Experimental spaces are defined in an INI format that defines a set of parameters and specifies their defaults and relationships. This can be thought of as a particular experimental setup or instrument.

We use the same experiment and Hamiltonian described above to demonstrate usage. Here, we are using a space called `trive` which provides, among other settings, two independent frequency axes and two independent delay axes, controlling a total of three incident pulses. The frequency axes are called `w1` and `w2`[2], the delays are `d1` and `d2`. To scan a particular axis, simply set the `points` array to a `NumPy` `numpy` array and set it's `active` attribute to `True`. You can also set a static value for any available axis, by setting the `points` attribute to a single number (and keeping `active` set to `False`). Finally, the `experiment` class defines the timing of the simulation. Three main parameters control this: `timestep`, which controls the size of each numerical integration step, `early_buffer`, which defines

[2]Note, while the Latin character `w` is used here because it is easier to type in code, it actually represents the Greek letter ω, conventionally, a frequency.

how long to integrate before the first pulse maximum, and `late_buffer`, which defines how long to integrate after the last pulse maximum. Here is an example of setting up a 3D (shape 64x64x32) scan with an additional static parameter set:

```
import WrightSim as ws
import numpy as np

dt = 50. # pulse duration (fs)
nw = 64 # number of frequency points (w1 and w2)
nt = 32 # number of delay points (d2)

# create experiment
exp = ws.experiment.builtin('trive')

# set the scan ranges
exp.w1.points = np.linspace(-500., 500., nw)
exp.w2.points = np.linspace(-500., 500., nw)
exp.d2.points = np.linspace(-2 * dt, 8 * dt, nt)
# tell WrightSim to treat the axis as scanned
exp.w1.active = exp.w2.active = exp.d2.active = True

# set a non-default delay time for the 'd1' axis
exp.d1.points = 4 * dt # fs
exp.d1.active = False

# set time between iterations, buffers
exp.timestep = 2. # fs
exp.early_buffer = 100.0 # fs
exp.late_buffer = 400.0 # fs
```

(3.1)

The Hamiltonian object is responsible for the density vector and holding on to the propagation function used when the experiment is run. Included in the density vector responsibility is the identity of which columns will be returned in the end result array. Hamiltonians may have arbitrary parameters to define themselves in intuitive ways. Under the hood, the Hamiltonian class also holds the C struct and source code for the `PyCUDA` implementation and a method to send itself to the CUDA device. Here is an example of setting up a Hamiltonian object with restricted pathways and explicitly set recorded element parameters:

```
## create hamiltonian
ham = ws.hamiltonian.Hamiltonian(w_central=0.)

# Select particular pathways
ham.time_orderings = [4, 5, 6]
# Select particular elements to be returned
ham.recorded_elements = [7,8]
```

(3.2)

Finally, all that is left is to run the experiment itself. The run method takes the Hamiltonian object and a keyword argument `mp`, short for "multiprocess". Any value that evaluates to `False` will run non-multiprocessed (i.e. single threaded). Almost all values that evaluates to `True` with run CPU - multiprocessed with the number of processes determined by the number of cores of the machine. The

exception is the special string 'gpu', which will cause WrightSim to run using PyCUDA.

```
## do scan, using PyCUDA
scan = exp.run(ham, mp='gpu')

# obtain results as a NumPy array
gpuSig = scan.sig.copy()
```
(3.3)

Running returns a Scan object, which contains several internal features of the scan including the electric field values themselves. The important part, however is the signal array that is generated. In this example, the complex floating point number array is of shape ($2x64x64x32$) (i.e. the number of recorded_elements followed by the shape of the experiment itself). These numbers can be easily manipulated and visualized to produce spectra like that seen in C.1. The Wright Group also maintains a library for working with multidimensional data, WrightTools WrightTools. This library will be integrated more fully to provide even easier access to visualization and archival storage of simulation results.

Performance

Performance is a critical consideration in the implementation of WrightSim. Careful analysis of the algorithms, identifying and measuring the bottlenecks, and working to implement strategies to avoid them are key to achieving the best performance possible. Another key is taking advantage of modern hardware for parallelization. These implementations have their advantages and trade-offs, which are quantified and examined in detail herein.

NISE [174] is the package written by Kohler and Thompson while preparing their manuscript [37]. NISE uses a slight variation on the technique described above, whereby they place a restriction on the time ordering represented by the matrix, and can thus use a seven element state vector rather than a 9 element state vector. This approach is mathematically equivalent to that presented above. NISE is included here as a reference for the performance of previous simulations of this kind.

Algorithmic Improvements When first translating the code from NISE into WrightSim, we sought to understand why it took so long to compute. We used Python's standard library package cProfile to produce traces of execution, and visualized them with SnakeViz [175]. Figure C.4 shows the trace obtained from a single-threaded run of NISE simulating a $32x32x16$ frequency-frequency-delay space. This trace provided some interesting insights into how the algorithm could be improved. First, 99.5% of the time is spent inside of a loop which is highly parallelizable. Second, almost one third of that time was spent in a specific function of NumPy, ix_. Further inspection of the code revealed that this function was called in the very inner most loop, but always had the same, small number of parameters. Lastly, approximately one tenth of the time was spent in a particular function called rotor (the bright orange box in Figure C.4). This function computed $cos(theta) + 1j * sin(theta)$, which could be replaced by the equivalent, but more efficient $exp(1j * theta)$. Additional careful analysis of the code revealed that redundant computations were being performed when generating matrices, which could be stored as variables and reused.

When implementing `WrightSim`, we took into account all of these insights. We simplified the code for matrix generation and propagation by only having the one 9 by 9 element matrix rather than two 7 by 7 matrices. The function that took up almost one third the time (`ix_`) was removed entirely in favor of a simpler scheme for denoting which values to record, simply storing a list of the indices directly. We used variables to store the values needed for matrix generation, rather than recalculating each element. As a result, solely by algorithmic improvements, almost an order of magnitude speedup was obtained (See Figure C.5). Still, 99% of the time was spent within a highly parallelizable inner loop.

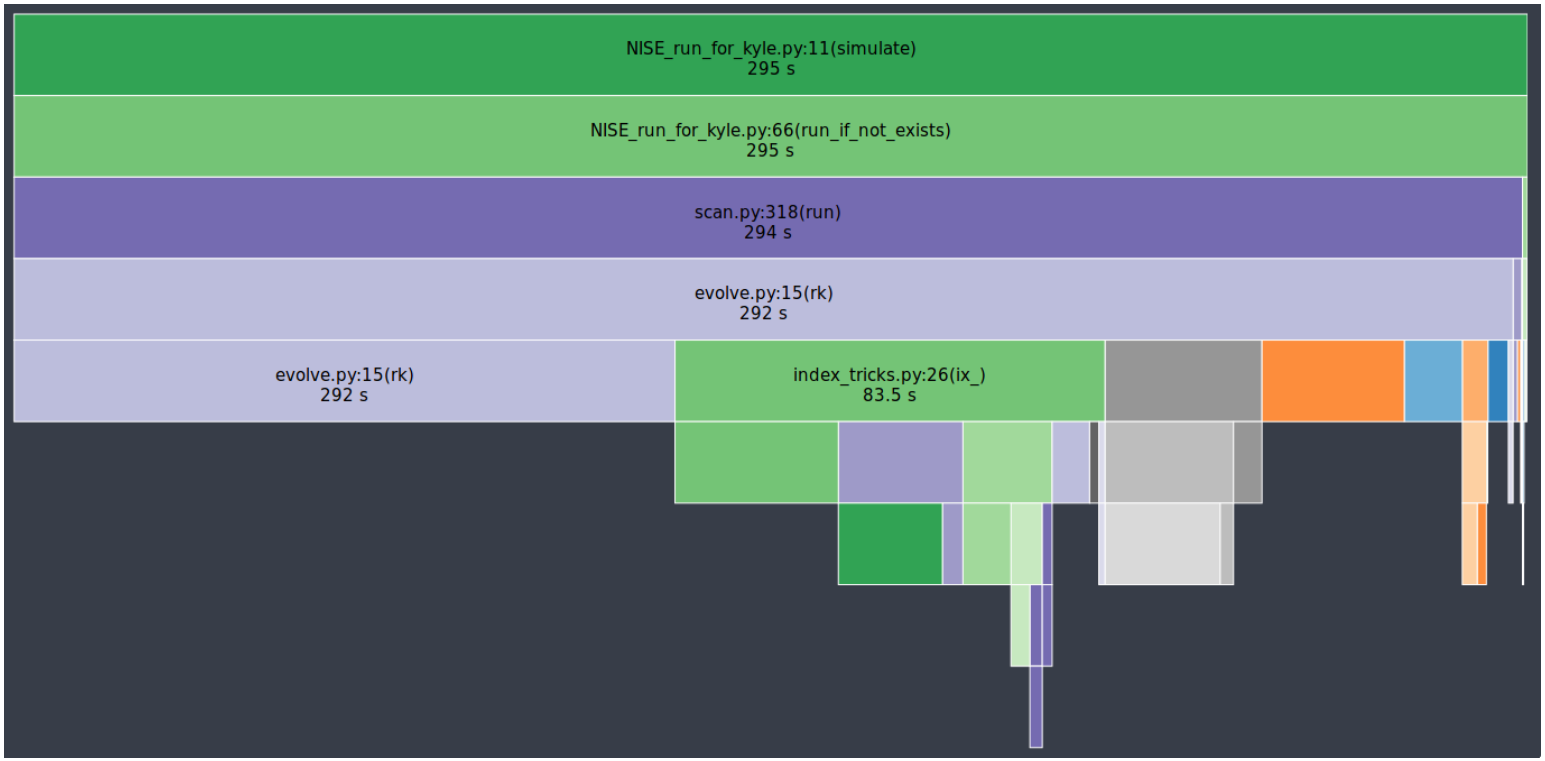

Figure C.4: Profile trace of a single threaded simulation from `NISE`.

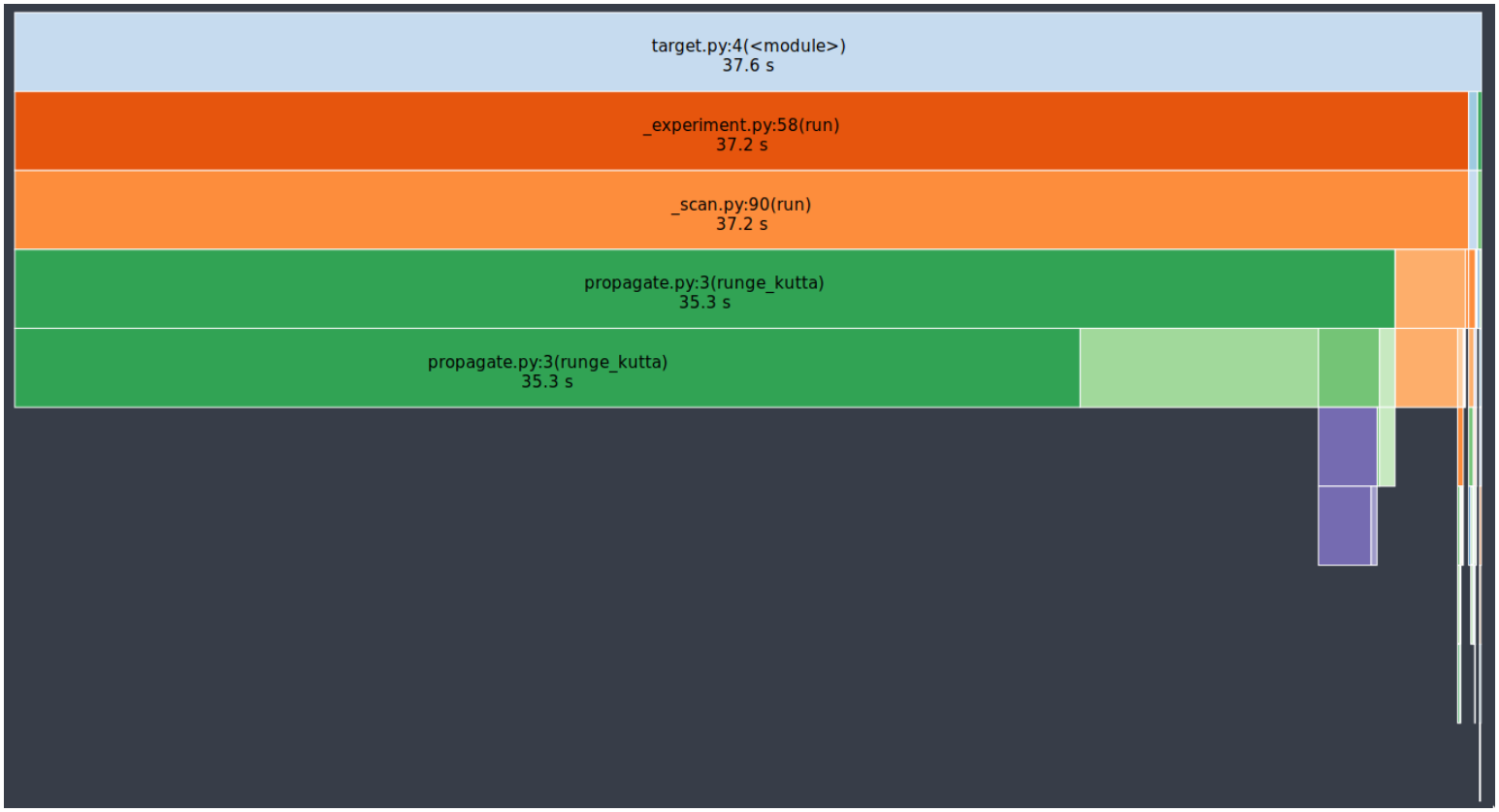

Figure C.5: Profile trace of a single threaded simulation from `WrightSim`.

CPU and GPU Parallel Implementations `NISE` already had, and `WrightSim` inherited, CPU multiprocessed parallelism using the Python standard library multiprocessing interface. Since almost all of the program is parallelizable, this incurs a four times speedup on a machine with four processing cores (limited more by the operating system scheduling other tasks than by Amdahl's law). This implementation required little adjustment outside of minor API tweaks.

In order to capitalize on the highly parallelizable nature of our multidimensional simulation, the algorithm was re-implemented using Nvidia CUDA [176]. In order to make the implementation as easy to use as possible, and maintainable over the lifetime of `WrightSim`, `PyCUDA` [177] was used to integrate the call to a CUDA kernel from within Python. `PyCUDA` allows the source code for the device side functions (written in C/C++) to exist as strings within the Python source files. These strings are just-in-time compiled (using `nvcc`) immediately prior to calling the kernel. For the initial work with the CUDA implementation, only one Hamiltonian and one propagation function were written, however it is extensible to additional methods. The just-in-time compilation makes it easy to replace individual functions as needed (a simple form of metaprogramming).

The CUDA implementation is slightly different from the pure Python implementation. It only holds in memory the Hamiltonian matrices for the current and next step, where the Python implementation computes all of the matrices prior to entering the loop. This was done to conserve memory on the GPU. Similarly, the electric fields are computed in the loop, rather than computing all ahead of time. These two optimizations reduce the memory overhead, and allow for easier to write functions, without the help of NumPy to perform automatic broadcasting of shapes.

Scaling Analysis Scaling analysis, tests of the amount of time taken by each simulation versus the number of points simulated, were conducted for each of the following: `NISE` single threaded, `NISE` Multiprocessed using four cores, `WrightSim` Single threaded, `WrightSim` Multiprocessed using four cores, and `WrightSim` CUDA implementation. A machine with an Intel Core i5-7600 (3.5 GHz) CPU and an Nvidia GTX 1060 (3GB) graphics card, running Arch Linux was used for all tests. The simulations were functionally identical, with the same number of time steps and same recorded values. The `NISE` simulations use two seven by seven matrices for the Hamiltonian, while the `WrightSim` simulations use a single nine by nine matrix. The results are summarized in Figure C.6.

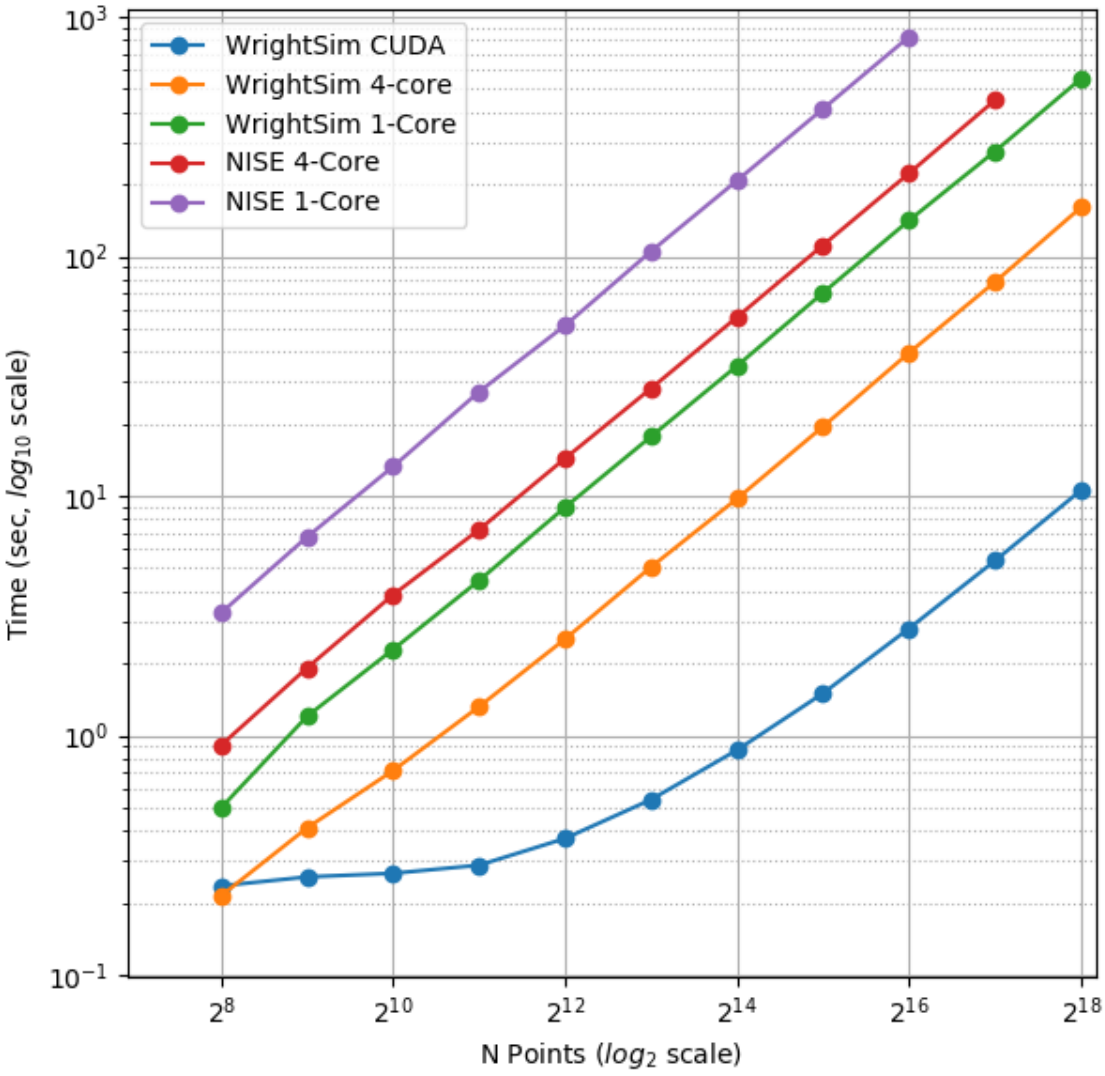

Figure C.6: Scaling Comparison of `WrightSim` and `NISE`

The log-log plot shows that the time scales linearly with number of points. All lines have approximately the same slope at high values of N, though the CUDA implementation grows slower at low N. The Algorithmic improvements alone offer doubled performance over even 4-Core multiprocessed `NISE` simulation. The CUDA implementation has a positive intercept at approximately 200 milliseconds. This is due, in large part, to the compilation overhead.

Limitations The CUDA implementation faces limitations at both ends in terms of number of points. On the low side, the cost of compilation and transfer of data makes it slower than the 4-Core CPU Multiprocessing implementation. This crossover point is approximately 256 points (for this simulation, all other parameters being equal). Incidentally, that is also a hard coded block size for the CUDA kernel call. While this could be modified to ensure no illegal memory accesses occur on smaller cases, the fact that you are not saving by using CUDA (and even single core performance is under a second) means it is not worth the effort at this time. The hard-coded block size also means that multiples of 256 points must be used in the current implementation.

With larger number of points, we are limited by the amount of memory available to be allocated on the GPU. For each pixel in the simulations presented here, 250 complex numbers represented as doubles must be allocated. Additional space is needed, however it is dominated by this array, which contains the outputs which are then transferred back to the host. Each CUDA thread additionally dynamically allocates the arrays it needs to perform the computation. The current implementation, paired with the particular hardware used, has a limit somewhere between 2^{18} and 2^{19} points. This limit could be increased by using single precision floating point numbers to represent the complex arrays, if the precision trade-off is acceptable (which is yet to be determined).

Future Work

This is still quite early days for `WrightSim`. While it is already a promising proof of concept display of how `PyCUDA` can be applied to this problem, there is still much room for improvement. In general, there are improvements to be made in terms of features, API/ease of use, and indeed further algorithmic improvements.

Features `NISE` had implemented a few additional features which were not carried over to `WrightSim` during the initial development efforts which focused on performance thus far.

There was support for chirped electric field pulses, which behave in less ideal fashions than the true sinusoids and Gaussian peaks used thus far. These non-ideal perturbations can have a real effect in spectra collected in the lab, and accurately modelling them helps to interpret these spectra.

Samples in laboratory experiments may have some amount of inhomogeneity within the sample, resulting in broader than would otherwise be expected peaks. This inhomogeneity can be modeled by storing the response array which is calculated by numerical integration, and translating the points slightly. The original `NISE` implementation would perform the simulation multiple times, where that is not needed as a simple translation will do. At one point we considered generating a library of responses in well known coordinates and saving them for future use, avoiding the expensive calculation all together. That seems to be less urgent, given the speed of the CUDA code.

NISE provided a powerful and flexible set of tools to "Measure" the signal, using Fourier transforms and produce arrays that even further mimic what is observed experimentally. That system needs to be added to WrightSim for it to be feature-complete. More naïve methods of visualizing work in this case, but a true measurement would allow for richer, more detailed analysis and interpretation.

Some new features could be added, including saving intermediate responses using an HDF5 based file format. The CUDA implementation itself would benefit from some way of saving the compiled code for multiple runs, removing the 0.2 second overhead. Current implementation compiles directly before calling the kernel, whether it has compiled it before or not. If performing many simulations in quick succession (e.g. a simulation larger than the memory allows in a single kernel call) with the same C code, the savings would add up.

The just-in-time compilation enables some special metaprogramming techniques which could be explored. The simple case is using separately programmed functions which have the same signature to do tasks in different ways. Currently there is a small shortcut in the propagation function which uses statically allocated arrays and pointers to those arrays rather than using dynamically allocated arrays. This relies on knowing the size at compilation time. The numbers could be replaced by preprocessor macros which are also fed to the compiler to assign this value "pseudo-dynamically" at compilation time. A much more advanced metaprogramming technique could, theoretically, generate the C struct and Hamiltonian generation function by inspecting the Python code and performing a translation. Such a technique would mean that new Hamiltonians would only have to be implemented once, in Python, and users who do not know C would be able to run CUDA code.

Usability One of the primary reasons for reimplementing the simulation package is to really think about our interface. As much as possible, the end user should not need to be an experienced programmer to be able to get a simulation. One of the next steps for WrightSim is to take a step back and ensure that our API is sensible and easy to follow. We wish to, as much as possible, provide ways of communicating through configuration files, rather than code. Ultimately, a GUI front end may be desirable, especially as the target audience is primarily experimentalists.

Additional Hamiltonians would make the package significantly more valuable as well. To add more Hamiltonians will require ensuring the code is robust, that values are transferred as expected. A few small assumptions were made in the interest of efficiency in the original implementation. Certain values, such as the initial density vector, represented by the Hamiltonian were hard-coded on the device code. While the hard-coded values are reasonable for most simulations, the ability to set theses at run time is desired, and will be added in the future.

Further Algorithmic Improvements While great strides were taken in improving the algorithms from previous implementations, there are several remaining avenues to gain improved performance in execution time and memory usage. The CUDA implementation is memory bound, both in terms of what can be dispatched, and in terms of time of execution. The use of single precision complex numbers (and other floating point values) would save roughly half of the space. One of the inputs is a large array with parameters for the each electric field at each pixel. This array contains much redundant data, which could be compressed with the parsing done in parallel on the device.

If the computed values could be streamed out of the GPU once computed, while others use the freed space, then there would be almost no limit on the number of points. This relies on the ability to stream

data back while computation is still going, which we do not have experience doing, and are not sure CUDA even supports. The values are not needed once they are recorded, so there is no need from the device side to keep the values around until computation is complete.

Additional memory could be conserved by using a bit field instead of an array of chars for determining which time orderings are used as a boolean array. This is relatively minimal, but is a current waste of bits. The Python implementation could potentially see a slight performance bump from using a boolean array rather than doing list searches for this same purpose.

The CUDA implementation does not currently take full advantage of shared cache. Most of the data needed is completely separated, but there are still a few areas where it could be useful.

The current CUDA implementation fills the Hamiltonian with zeros at every time step. The values which are nonzero after the first call are always going to be overwritten anyway, so this wastes time inside of of nested loop. This zeroing could be done only before the first call, removing the nested loop. Additionally, many matrices have a lot of zero values. Often they are triangular matrices, which would allow for a more optimized dot product computation which ignores the zeros in the half which is not populated. Some matrices could even benefit by being represented as sparse matrices, though these are more difficult to use.

Finally, perhaps the biggest, but also most challenging, remaining possible improvement would be to capitalize on the larger symmetries of the system. It's a non-trivial task to know which axes are symmetric, but if it could be done, the amount that actually needs to be simulated would be much smaller. Take the simulation in Figure C.1. This was computed as it is displayed, but there are two orthogonal axes of symmetry, which would cut the amount actually needed to replicate the spectrum down by a factor of four. Higher dimensional scans with similar symmetries would benefit even more.

Conclusions

`WrightSim`, as implemented today, represents the first major step towards a cohesive, easy to use, fast simulation suite for quantum mechanical numerically integrated simulations using density matrix theory. Solely algorithmic improvements enabled the pure Python implementation to be an order of magnitude faster than the previous implementation. The algorithm is highly parallelizable, enabling easy CPU level parallelism. A new implementation provides further improvement than the CPU parallel code, taking advantage of the General Purpose-GPU Computation CUDA library. This implementation provides approximately 2.5 orders of magnitude improvement over the existing `NISE` serial implementation. There are still ways that this code can be improved, both in performance and functionality. With `WrightSim`, we aim to lead by example among the spectroscopic community by providing an open-source package for general-purpose MDS simulation.

Bibliography

[1] Wei Zhao and John C. Wright. "Spectral Simplification in Vibrational Spectroscopy Using Doubly Vibrationally Enhanced Infrared Four Wave Mixing". In: *Journal of the American Chemical Society* 121.47 (Nov. 1999), pp. 10994–10998. DOI: 10.1021/ja9926414. URL: https://doi.org/10.1021%2Fja9926414.

[2] Paul M Donaldson, Rui Guo, Frederic Fournier, Elizabeth M Gardner, Ian R Gould, and David R Klug. "Decongestion of methylene spectra in biological and non-biological systems using picosecond 2DIR spectroscopy measuring electron-vibration–vibration coupling". In: *Chemical Physics* 350.1-3 (June 2008), pp. 201–211. DOI: 10.1016/j.chemphys.2008.02.050.

[3] Erin S Boyle, Nathan A Neff-Mallon, and John C Wright. "Triply Resonant Sum Frequency Spectroscopy: Combining Advantages of Resonance Raman and 2D-IR". In: *The Journal of Physical Chemistry A* 117.47 (Nov. 2013), pp. 12401–12408. DOI: 10.1021/jp409377a.

[4] Erin S Boyle, Nathan A. Neff-Mallon, Jonathan D Handali, and John C Wright. "Resonance IR: A Coherent Multidimensional Analogue of Resonance Raman". In: *The Journal of Physical Chemistry A* 118.17 (May 2014), pp. 3112–3119. DOI: 10.1021/jp5018554.

[5] Dorine Keusters, Howe-Siang Tan, and Warren S. Warren. "Role of Pulse Phase and Direction in Two-Dimensional Optical Spectroscopy". In: *The Journal of Physical Chemistry A* 103.49 (Dec. 1999), pp. 10369–10380. DOI: 10.1021/jp992325b. URL: http://dx.doi.org/10.1021/jp992325b.

[6] W. Zhao and J. C. Wright. "Doubly Vibrationally Enhanced Four Wave Mixing: The Optical Analog to 2D NMR". In: *Physical Review Letters* 84 (Feb. 2000), pp. 1411–1414. DOI: 10.1103/PhysRevLett.84.1411. URL: http://dx.doi.org/10.1103/PhysRevLett.84.1411.

[7] Andrei V. Pakoulev, Mark A. Rickard, Kent A. Meyer, Kathryn Kornau, Nathan A. Mathew, David E. Thompson, and John C. Wright. "Mixed Frequency/Time Domain Optical Analogues of Heteronuclear Multidimensional NMR". In: *The Journal of Physical Chemistry A* 110.10 (Mar. 2006), pp. 3352–3355. DOI: 10.1021/jp057339y. URL: https://doi.org/10.1021/jp057339y.

[8] John C. Wright. "Schrödinger Cat State Spectroscopy—A New Frontier for Analytical Chemistry". In: *Analytical Chemistry* 92.13 (June 2020), pp. 8638–8643. DOI: 10.1021/acs.analchem.0c01662. URL: https://doi.org/10.1021%2Facs.analchem.0c01662.

[9] Shaul Mukamel, Yoshitaka Tanimura, and Peter Hamm. "Coherent Multidimensional Optical Spectroscopy". In: *Accounts of Chemical Research* 42.9 (Sept. 2009), pp. 1207–1209. DOI: 10.1021/ar900227m. URL: https://dx.doi.org/10.1021/ar900227m.

[10] Sarah M. Gallagher, Allison W. Albrecht, John D. Hybl, Brett L. Landin, Bhavani Rajaram, and David M. Jonas. "Heterodyne detection of the complete electric field of femtosecond four-wave mixing signals". In: *Journal of the Optical Society of America B* 15.8 (Aug. 1998), p. 2338. DOI: 10.1364/josab.15.002338. URL: https://doi.org/10.1364/josab.15.002338.

[11] John D. Hybl, Allison W. Albrecht, Sarah M. Gallagher Faeder, and David M. Jonas. "Two-dimensional electronic spectroscopy". In: *Chemical Physics Letters* 297.3-4 (Nov. 1998), pp. 307–313. DOI: 10.1016/s0009-2614(98)01140-3. URL: https://doi.org/10.1016%2Fs0009-2614%2898%2901140-3.

[12] Nicholas M. Kearns, Randy D. Mehlenbacher, Andrew C. Jones, and Martin T. Zanni. "Broadband 2D electronic spectrometer using white light and pulse shaping: noise and signal evaluation at 1 and 100 kHz". In: *Optics Express* 25.7 (Mar. 2017), p. 7869. DOI: 10.1364/oe.25.007869. URL: https://dx.doi.org/10.1364/oe.25.007869.

[13] Blaise Jonathan Thompson. "Development of Frequency Domain Multidimensional Spectroscopy with Applications in Semiconductor Photophysics". PhD thesis. University of Wisconsin–Madison, 2018.

[14] W. M. Tolles, J. W. Nibler, J. R. McDonald, and A. B. Harvey. "Review of Theory and Applications of Coherent Anti-Stokes Raman Spectroscopy". In: *Applied Spectroscopy* 31 (1977), pp. 253–271.

[15] Roger J. Carlson and John C. Wright. "Absorption and Coherent Interference Effects in Multiply Resonant Four-Wave Mixing Spectroscopy". In: *Applied Spectroscopy* 43.7 (Sept. 1989), pp. 1195–1208. DOI: 10.1366/0003702894203408. URL: https://doi.org/10.1366/0003702894203408.

[16] Keith M. Murdoch, David E. Thompson, Kent A. Meyer, and John C. Wright. "Modeling Window Contributions to Four-Wave Mixing Spectra and Measurements of Third-Order Optical Susceptibilities". In: *Appl. Spectrosc.* 54.10 (Oct. 2000), pp. 1495–1505.

[17] Jonathan D. Handali, Kyle F. Sunden, Blaise J. Thompson, Nathan A. Neff-Mallon, Emily M. Kaufman, Thomas C. Brunold, and John C. Wright. "Three Dimensional Triply Resonant Sum Frequency Spectroscopy Revealing Vibronic Coupling in Cobalamins: Toward a Probe of Reaction Coordinates". In: *The Journal of Physical Chemistry A* 122.46 (Oct. 2018), pp. 9031–9042. DOI: 10.1021/acs.jpca.8b07678. URL: https://dx.doi.org/10.1021/acs.jpca.8b07678.

[18] Austin P. Spencer, Hebin Li, Steven T. Cundiff, and David M. Jonas. "Pulse Propagation Effects in Optical 2D Fourier-Transform Spectroscopy: Theory". In: *The Journal of Physical Chemistry A* 119.17 (Apr. 2015), pp. 3936–3960. DOI: 10.1021/acs.jpca.5b00001. URL: https://dx.doi.org/10.1021/acs.jpca.5b00001.

[19] Darien J. Morrow, Daniel D. Kohler, and John C. Wright. "Group- and phase-velocity-mismatch fringes in triple sum-frequency spectroscopy". In: *Physical Review A* 96.6 (Dec. 2017). DOI: 10.1103/physreva.96.063835.

[20] Blaise Thompson, Kyle Sunden, Darien Morrow, Daniel Kohler, and John Wright. "WrightTools: a Python package for multidimensional spectroscopy". In: *Journal of Open Source Software* 4.33 (Jan. 2019), p. 1141. DOI: 10.21105/joss.01141. URL: https://doi.org/10.21105/joss.01141.

[21] Wei Zhao and John C. Wright. "Spectral Simplification in Vibrational Spectroscopy Using Doubly Vibrationally Enhanced Infrared Four Wave Mixing". In: *Journal of the American Chemical Society* 121.47 (Dec. 1999), pp. 10994–10998. DOI: 10.1021/ja9926414.

[22] Andrei V. Pakoulev, Mark A. Rickard, Kathryn M. Kornau, Nathan A. Mathew, Lena A. Yurs, Stephen B. Block, and John C. Wright. "Mixed Frequency-/Time-Domain Coherent Multidimensional Spectroscopy: Research Tool or Potential Analytical Method?" In: *Accounts of Chemical Research* 42.9 (Sept. 2009), pp. 1310–1321. DOI: 10.1021/ar900032g. URL: https://doi.org/10.1021/ar900032g.

[23] John C. Wright. "Multiresonant Coherent Multidimensional Spectroscopy". In: *Annual Review of Physical Chemistry* 62.1 (May 2011), pp. 209–230. DOI: 10.1146/annurev-physchem-032210-103551.

[24] Christopher L. Smallwood and Steven T. Cundiff. "Multidimensional Coherent Spectroscopy of Semiconductors". In: *Laser & Photonics Reviews* (Nov. 2018), p. 1800171. DOI: 10.1002/lpor.201800171. URL: https://doi.org/10.1002/lpor.201800171.

[25] Kyle J. Czech, Blaise J. Thompson, Schuyler Kain, Qi Ding, Melinda J. Shearer, Robert J. Hamers, Song Jin, and John C. Wright. "Measurement of Ultrafast Excitonic Dynamics of Few-Layer MoS2Using State-Selective Coherent Multidimensional Spectroscopy". In: *ACS Nano* 9.12 (Dec. 2015), pp. 12146–12157. DOI: 10.1021/acsnano.5b05198. URL: https://dx.doi.org/10.1021/acsnano.5b05198.

[26] Travis E. Oliphant. "Python for Scientific Computing". In: *Computing in Science & Engineering* 9.3 (2007), pp. 10–20. DOI: 10.1109/MCSE.2007.58.

[27] Travis E Oliphant. *A guide to NumPy*. Trelgol Publishing, 2006.

[28] John D. Hunter. "Matplotlib: A 2D Graphics Environment". In: *Computing in Science Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/mcse.2007.55. URL: https://doi.org/10.1109/mcse.2007.55.

[29] Andrew Collette. *Python and HDF5*. O'Reilly, 2013.

[30] The HDF Group. *Hierarchical Data Format, version 5*. http://www.hdfgroup.org/HDF5/. 1997.

[31] Nathan A. Neff-Mallon and John C. Wright. "Multidimensional Spectral Fingerprints of a New Family of Coherent Analytical Spectroscopies". In: *Analytical Chemistry* 89.24 (Nov. 2017), pp. 13182–13189. DOI: 10.1021/acs.analchem.7b02917.

[32] *WrightTools*. Accessed: 2022-10-02. URL: https://github.com/wright-group/WrightTools.

[33] Blaise J. Thompson, Kyle F. Sunden, Darien J. Morrow, Daniel D. Kohler, Nathan Andrew Neff-Mallon, Kyle J. Czech, Emily M. Kaufman, Thomas Parker, and Rachel Swedin. *Wrighttools*. 2018. DOI: 10.5281/zenodo.1198904. URL: https://zenodo.org/record/1198904.

[34] *WrightTools*. Accessed: 2022-10-02. URL: https://pypi.org/project/wrighttools.

[35] *WrightTools*. Accessed: 2022-10-02. URL: http://anaconda.org/conda-forge/wrighttools.

[36] *WrightTools*. Accessed: 2022-10-02. URL: https://wright.tools.

[37] Daniel D. Kohler, Blaise J. Thompson, and John C. Wright. "Frequency-domain coherent multidimensional spectroscopy when dephasing rivals pulsewidth: Disentangling material and instrument response". In: *The Journal of Chemical Physics* 147.8 (Aug. 2017), p. 084202. DOI: 10.1063/1.4986069. URL: https://doi.org/10.1063/1.4986069.

[38] Jie Chen, Darien J. Morrow, Yongping Fu, Weihao Zheng, Yuzhou Zhao, Lianna Dang, Matthew J. Stolt, Daniel D. Kohler, Xiaoxia Wang, Kyle J. Czech, Matthew P. Hautzinger, Shaohua Shen, Liejin Guo, Anlian Pan, John C. Wright, and Song Jin. "Single-Crystal Thin Films of Cesium Lead Bromide Perovskite Epitaxially Grown on Metal Oxide Perovskite (SrTiO3)". In: *Journal of the American Chemical Society* 139.38 (Sept. 2017), pp. 13525–13532. DOI: 10.1021/jacs.7b07506.

[39] Erik H. Horak, Morgan T. Rea, Kevin D. Heylman, David Gelbwaser-Klimovsky, Semion K. Saikin, Blaise J. Thompson, Daniel D. Kohler, Kassandra A. Knapper, Wei Wei, Feng Pan, Padma Gopalan, John C. Wright, Alán Aspuru-Guzik, and Randall H. Goldsmith. "Exploring Electronic Structure and Order in Polymers via Single-Particle Microresonator Spectroscopy". In: *Nano Letters* 18.3 (Jan. 2018), pp. 1600–1607. DOI: 10.1021/acs.nanolett.7b04211.

[40] Kyle F. Sunden, Blaise J. Thompson, and John C. Wright. "WrightSim: Using PyCUDA to Simulate Multidimensional Spectra". In: *PRoceedings of the 17th PYthon in SCience COnference* (2018). Ed. by Fatih Akici, David Lippa, Dillon Niederhut, and M Pacer, pp. 77–83. DOI: 10.25080/Majora-4af1f417-00c.

[41] Daniel D. Kohler, Blaise J. Thompson, and John C. Wright. "Resonant Third-Order Susceptibility of PbSe Quantum Dots Determined by Standard Dilution and Transient Grating Spectroscopy". In: *The Journal of Physical Chemistry C* 122.31 (July 2018), pp. 18086–18093. DOI: 10.1021/acs.jpcc.8b04462. URL: https://doi.org/10.1021/acs.jpcc.8b04462.

[42] Jonathan D. Handali, Kyle F. Sunden, Emily M. Kaufman, and John C. Wright. "Interference and phase mismatch effects in coherent triple sum frequency spectroscopy". In: *Chemical Physics* (June 2018). DOI: 10.1016/j.chemphys.2018.05.023.

[43] Darien J. Morrow, Daniel D. Kohler, Kyle J. Czech, and John C. Wright. "Communication: Multidimensional triple sum-frequency spectroscopy of MoS2 and comparisons with absorption and second harmonic generation spectroscopies". In: *The Journal of Chemical Physics* 149.9 (Sept. 2018), p. 091101. DOI: 10.1063/1.5047802.

[44] Blaise J. Thompson, Kyle F. Sunden, Darien J. Morrow, and Nathan Andrew Neff-Mallon. *Pycmds*. 2018. DOI: 10.5281/zenodo.1481203. URL: https://zenodo.org/record/1481203.

[45] *conda*. Accessed: 2022-10-02. URL: https://conda.io/en/latest/index.html.

[46] *Anaconda Distribution*. Accessed: 2022-10-02. URL: https://www.anaconda.com/products/distribution.

[47] *Miniconda*. Accessed: 2022-10-02. URL: https://conda.io/en/latest/miniconda.html.

[48] *pip*. Accessed: 2022-10-02. URL: https://pip.pypa.io/.

[49] *attune*. Accessed: 2022-10-02. URL: https://pypi.org/project/attune.

[50] *Getting Started with NI-DAQmx*. Accessed: 2022-10-02. 2022. URL: https://www.ni.com/en-us/support/documentation/supplemental/06/getting-started-with-ni-daqmx--main-page.html.

[51] *Standard Commands for Programmable Instruments (SCPI)*. Version 1999.0. SCPI Consortium. 1999.

[52] *MODBUS APPLICATION PROTOCOL SPECIFICATION*. Version V1.1b3. Modbus. 2012.

[53] *PICam™ 5.x Programmer's Manual*. Teledyne Princeton Instruments. 2021.

[54] *Thorlabs Motion Controllers Host-Controller Communications Protocol*. Thorlabs, GmbH. 2022.

[55] L. R. Dalesio, M. R. Kraimer, and A. J. Kozubal. "EPICS Architecture". In: *ICALEPCS* 91 (1991).

[56] J-M. Chaize, A. G¨otz, J. Meyer, María Pérez, and E. Taurel. "TANGO - AN OBJECT ORIENTED CONTROL SYSTEM BASED ON CORBA". In: *ICALEPCS*. 1999.

[57] S. J. Weber. "PyMoDAQ: An open-source Python-based software for modular data acquisition". In: *Review of Scientific Instruments* 92.4 (Apr. 2021), p. 045104. DOI: 10.1063/5.0032116.

[58] Nate Bogdanowicz, Christopher Rogers, Sébastien Weber, Zakv, Sylvain Pelissier, Jonathan Wheeler, Cxz, Francesco Marazzi, Eedm, and Ivan Galinskiy. *mabuchilab/Instrumental: 0.7*. 2022. DOI: 10.5281/ZENODO.2556398. URL: https://zenodo.org/record/2556398.

[59] Patrick Tapping. *TRSpectrometer Documentation*. Accessed: 2022-10-02. 2021. URL: https://trspectrometer.readthedocs.io/en/latest/.

[60] Lucas Koerner. "Instrbuilder: A Python package for electrical instrument control". In: *Journal of Open Source Software* 4.36 (Apr. 2019), p. 1172. DOI: 10.21105/joss.01172. URL: https://doi.org/10.21105%2Fjoss.01172.

[61] Luke Campagnola, Megan B. Kratz, and Paul B. Manis. "ACQ4: an open-source software platform for data acquisition and analysis in neurophysiology research". In: *Frontiers in Neuroinformatics* 8 (2014). DOI: 10.3389/fninf.2014.00003. URL: https://doi.org/10.3389%2Ffninf.2014.00003.

[62] *PyMeasure*. Accessed: 2022-10-02. 2022. URL: https://pymeasure.readthedocs.io/.

[63] Grant Giesbrecht, Ariel Amsellem, Timo Bauer, Brian Mak, Brian Wynne, Zhihao Qin, and Arun Persaud. "Hardware-Control: Instrument control and automation package". In: *Journal of Open Source Software* 7.72 (Apr. 2022), p. 2688. DOI: 10.21105/joss.02688. URL: https://doi.org/10.21105%2Fjoss.02688.

[64] Alexey Shkarin. *pylablib*. Accessed: 2022-10-02. 2022. URL: https://pypi.org/project/pylablib.

[65] *yaq*. Accessed: 2022-11-06. URL: https://yaq.fyi/.

[66] Eric S. Raymond. *The new hacker's dictionary*. 3rd ed. Cambridge, Mass: MIT Press, 1996.

[67] *Apache Avro Specification*. Accessed: 2022-10-02. 2022. URL: https://avro.apache.org/docs/1.11.1/specification/.

[68] Judith Segal. "When Software Engineers Met Research Scientists: A Case Study". In: *Empirical Software Engineering* 10.4 (Oct. 2005), pp. 517–536. DOI: 10.1007/s10664-005-3865-y.

[69] *yaqc*. Accessed: 2022-10-02. URL: https://pypi.org/project/yaqc.

[70] *yaqc-qtpy*. Accessed: 2022-10-02. URL: https://pypi.org/project/yaqc-qtpy.

[71] Daniel Allan, Thomas Caswell, Stuart Campbell, and Maksim Rakitin. "Bluesky's Ahead: A Multi-Facility Collaboration for an a la Carte Software Project for Data Acquisition and Management". In: *Synchrotron Radiation News* 32.3 (May 2019), pp. 19–22. DOI: 10.1080/08940886.2019.1608121.

[72] *yaqc-bluesky*. Accessed: 2022-10-02. URL: https://pypi.org/project/yaqc-bluesky.

[73] Shaul Mukamel. "Multidimensional Femtosecond Correlation Spectroscopies of Electronic and Vibrational Excitations". In: *Annual Review of Physical Chemistry* 51.1 (Oct. 2000), pp. 691–729. DOI: 10.1146/annurev.physchem.51.1.691. URL: https://doi.org/10.1146/annurev.physchem.51.1.691.

[74] Roger John Carlson. "Quantitative Aspects of High Resolution, Fully Resonant, Four-Wave Mixing Spectroscopy For the Analysis of Vibronic Mode Coupling in Molecules". PhD thesis. University of Wisconsin-Madison, 1988.

[75] Kent Albert Meyer. "Freqncy-Scanned Ultrafast Spectroscopic Techniques Applied to Infrared Four-Wave Mixing Spectroscopy". PhD thesis. University of Wisconsin–Madison, 2004.

[76] Schuyler Kain. "Transition of Frequency-Domain Coherent Multidimensional Spectroscopic Methods to the Femtosecond Time Regime with Applications to Nanoscale Semiconductors". PhD thesis. University of Wisconsin-Madison, 2017.

[77] Kyle Foster Sunden. "yaq: Yet Another Acquisition A modular approach to spectroscopy software and instrumentation". PhD thesis. University of Wisconsin–Madison, 2022.

[78] *yaqd-core*. Accessed: 2022-10-02. URL: https://pypi.org/project/yaqd-core.

[79] *yaq-traits*. Accessed: 2022-10-02. URL: https://pypi.org/project/yaq-traits.

[80] *yaqd-brooks-mfc-gf source code*. Accessed: 2022-10-02. URL: https://github.com/yaq-project/yaqd-brooks/blob/7cc1de/yaqd_brooks/_brooks_mfc_gf.py.

[81] *pyserial*. Accessed: 2022-10-02. URL: https://pypi.org/project/pyserial.

[82] *pyusb*. Accessed: 2022-11-06. URL: https://pypi.org/project/pyusb.

[83] *pyvisa*. Accessed: 2022-11-06. URL: https://pypi.org/project/pyvisa.

[84] *pymodbus*. Accessed: 2022-11-06. URL: https://pypi.org/project/pymodbus.

[85] *yaq/daemons*. Accessed: 2022-11-06. URL: https://yaq.fyi/daemons.

[86] Chase A. Salazar, Blaise J. Thompson, Spring M. M. Knapp, Steven R. Myers, and Shannon S. Stahl. "Multichannel gas-uptake/evolution reactor for monitoring liquid-phase chemical reactions". In: *Review of Scientific Instruments* 92.4 (Apr. 2021), p. 044103. DOI: 10.1063/5.0043007.

[87] Eric Allman. "Managing technical debt". In: *Communications of the ACM* 55.5 (May 2012), pp. 50–55. DOI: 10.1145/2160718.2160733. URL: https://doi.org/10.1145%2F2160718.2160733.

[88] *yaqd-fakes*. Accessed: 2022-10-02. URL: https://pypi.org/project/yaqd-fakes.

[89] Jeremy Cohen, Daniel S. Katz, Michelle Barker, Neil Chue Hong, Robert Haines, and Caroline Jay. "The Four Pillars of Research Software Engineering". In: *IEEE Software* 38.1 (Jan. 2021), pp. 97–105. DOI: 10.1109/ms.2020.2973362.

[90] Anna Nowogrodzki. "How to support open-source software and stay sane". In: *Nature* 571.7763 (July 2019), pp. 133–134. DOI: 10.1038/d41586-019-02046-0.

[91] *yaq-traits*. Accessed: 2022-10-02. URL: https://anaconda.org/conda-forge/yaq-traits.

[92] *Apache Avro Specification: Protocol Declaration*. Accessed: 2022-10-02. URL: https://avro.apache.org/docs/1.11.1/specification/#protocol-declaration.

[93] Tom Preston-Werner and Pradyun Gedam. *TOML Specification 1.0*. 2020. URL: https://toml.io/en/v1.0.0.

[94] *yaq Traits*. Accessed: 2022-10-02. URL: https://yaq.fyi/traits/.

[95] *yaq-traits changelog*. Accessed: 2022-10-02. URL: https://github.com/yaq-project/yaq-traits/blob/main/CHANGELOG.md.

[96] *yaqd-core changelog*. Accessed: 2022-10-02. URL: https://github.com/yaq-project/yaq-python/blob/main/yaqd-core/CHANGELOG.md.

[97] *yaq contact*. Accessed: 2022-10-02. URL: https://yaq.fyi/contact/.

[98] *yaq-traits Issues*. Accessed: 2022-10-02. URL: https://github.com/yaq-project/yaq-traits/issues.

[99] *yaq contact*. Accessed: 2022-10-02. URL: https://yaq.fyi/hardware/.

[100] *Apache Avro Specification: Schema Declaration*. Accessed: 2022-10-02. URL: https://avro.apache.org/docs/1.11.1/specification/#schema-declaration.

[101] *Apache Avro Specification: Messages*. Accessed: 2022-10-02. URL: https://avro.apache.org/docs/1.11.1/specification/#messages.

[102] *yaqd-control*. Accessed: 2022-10-02. URL: https://pypi.org/project/yaqd-control.

[103] *yaqd-control*. Accessed: 2022-10-02. URL: https://anaconda.org/conda-forge/yaqd-control.

[104] *QtPy*. Accessed: 2022-10-02. URL: https://pypi.org/project/qtpy.

[105] *Qt Documentation*. Accessed: 2022-10-02. URL: https://docs.qt.io.

[106] *IPython*. Accessed: 2022-10-02. URL: https://pypi.org/project/ipython.

[107] *let's yaq: new-era-ne1000*. Accessed: 2022-10-02. URL: https://python.yaq.fyi/ne1000/.

[108] *yaqd Cookiecutter for Python*. Accessed: 2022-10-02. URL: https://github.com/yaq-project/yaqd-cookiecutter-python.

[109] *Cookiecutter*. Accessed: 2022-10-02. URL: https://pypi.org/project/cookiecutter.

[110] *pre-commit*. Accessed: 2022-10-02. URL: https://pypi.org/project/pre-commit.

[111] *GitHub Actions*. Accessed: 2022-10-02. URL: https://docs.github.com/en/actions.

[112] *mypy*. Accessed: 2022-10-02. URL: https://pypi.org/project/mypy.

[113] *PyPI - the Python Package Index*. Accessed: 2022-10-02. URL: https://pypi.org/.

[114] *flit*. Accessed: 2022-10-02. URL: https://pypi.org/project/flit.

[115] *PEP 0 – Index of Python Enhancement Proposals (PEPs)*. Accessed: 2022-10-02. URL: https://peps.python.org/.

[116] *Roadmap & NumPy Enhancement Proposals*. Accessed: 2022-10-02. URL: https://numpy.org/neps/.

[117] *I2C-bus specification and user manual*. Version 7.0. NXP. 2022. URL: https://www.nxp.com/docs/en/user-guide/UM10204.pdf.

[118] *System Management Bus (SMBus) Specification*. Version 3.2. System Management Interface Forum, Inc. 2022. URL: http://www.smbus.org/specs/.

[119] *Daemon State Files*. Accessed: 2022-10-02. URL: https://yeps.yaq.fyi/103/.

[120] *logging — Logging facility for Python*. Accessed: 2022-10-02. URL: https://docs.python.org/3/library/logging.html.

[121] *sd-daemon — APIs for new-style daemons*. Accessed: 2022-10-02. URL: https://systemd.network/sd-daemon.html.

[122] R. Gerhards. *The Syslog Protocol*. Tech. rep. Mar. 2009. DOI: 10.17487/rfc5424. URL: https://doi.org/10.17487%2Frfc5424.

[123] *asyncio — Asynchronous I/O*. Accessed: 2022-10-02. URL: https://docs.python.org/3/library/asyncio.html.

[124] *Synchronization Primitives*. Accessed: 2022-10-02. URL: https://docs.python.org/3/library/asyncio-sync.html.

[125] *Queues*. Accessed: 2022-10-02. URL: https://docs.python.org/3/library/asyncio-queue.html.

[126] Blaise J. Thompson, Kyle F. Sunden, and Nathan Andrew Neff-Mallon. *PyCMDS 0.8.0*. 2018. DOI: 10.5281/zenodo.1198911. URL: http://dx.doi.org/10.5281/zenodo.1198911.

[127] Kenneth Lauer. "ophyd Devices: Imposing Hierarchy on the Flat EPICS V3 Namespace". en. In: *Proceedings of the 17th International Conference on Accelerator and Large Experimental Physics Control Systems* ICALEPCS2019 (2020), USA. DOI: 10.18429/JACOW-ICALEPCS2019-WEPHA083. URL: https://jacow.org/icalepcs2019/doi/JACoW-ICALEPCS2019-WEPHA083.html.

[128] *yaqc-bluesky*. Accessed: 2022-10-02. URL: https://pypi.org/project/yaqc-bluesky.

[129] *Bluesky Event Model Documentation*. Accessed: 2022-10-02. URL: https://blueskyproject.io/event-model/.

[130] *Composable cycles*. Accessed: 2022-10-02. URL: https://matplotlib.org/cycler/.

[131] Dirk Merkel. "Docker: lightweight linux containers for consistent development and deployment". In: *Linux journal* 2014.239 (2014), p. 2.

[132] *Docker overview*. Accessed: 2022-10-02. URL: https://docs.docker.com/get-started/overview/.

[133] *Docker Compose Overview*. Accessed: 2022-10-02. URL: https://docs.docker.com/compose/.

[134] *Windows Subsystem for Linux Documentation*. Accessed: 2022-10-02. URL: https://learn.microsoft.com/en-us/windows/wsl/.

[135] *Docker Desktop*. Accessed: 2022-10-02. URL: https://www.docker.com/products/docker-desktop/.

[136] *happi*. Accessed: 2022-10-02. URL: https://pypi.org/project/happi.

[137] Pieter Hintjens. *ZeroMQ*. First edition. Beijing: O'Reilly, 2013.

[138] *FastAPI*. Accessed: 2022-10-02. URL: https://pypi.org/project/FastAPI.

[139] *Kafka 3.2*. Accessed: 2022-10-02. URL: https://kafka.apache.org/documentation/.

[140] Michael Dirolf and Kristina Chodorow. *MongoDB: The Definitive Guide*. eng. 1. ed. Beijing Köln: O'Reilly, 2010.

[141] *Introduction to Redis*. Accessed: 2022-10-02. URL: https://redis.io/docs/about/.

[142] *bluesky-hwproxy*. Accessed: 2022-10-02. URL: https://pypi.org/project/bluesky-hwproxy.

[143] *PyQtGraph: Scientific Graphics and GUI Library for Python*. Accessed: 2022-10-02. URL: http://pyqtgraph.org/.

[144] Callum Forrester, Daniel Allan, Thomas Caswell, and Thomas Cobb. In: *NOBUGS 2022: Book of Abstracts*. Villigen, Switzerland: Paul Scherer Institute, Sept. 2022.

[145] *bluesky-widgets*. Accessed: 2022-10-02. URL: https://pypi.org/project/bluesky-widgets.

[146] Arman Arkilic, Daniel Allan, Daron Chabot, Leo Dalesio, and Wayne Lewis. "Databroker: An Interface for NSLS-II Data Management System". en. In: *Proceedings of the 15th Int. Conf. on Accelerator and Large Experimental Physics Control Systems* ICALEPCS2015 (2015), Australia. DOI: 10.18429/JACOW-ICALEPCS2015-WED3O02. URL: http://accelconf.web.cern.ch/AccelConf/DOI/ICALEPCS2015/JACoW-ICALEPCS2015-WED3O02.html.

[147] Stephan Hoyer and Joe Hamman. "xarray: N-D labeled Arrays and Datasets in Python". In: *Journal of Open Research Software* 5.1 (Apr. 2017), p. 10. DOI: 10.5334/jors.148. URL: https://doi.org/10.5334%2Fjors.148.

[148] *Z812 - 12 mm Motorized Actuator, 1/4"-80 Thread (0.5 m Cable)*. Accessed: 2022-10-02. URL: https://www.thorlabs.us/thorProduct.cfm?partNumber=Z812.

[149] *Piezo Rotation Stage With CONEX-AGP Controller*. Accessed: 2022-10-02. URL: https://www.newport.com/f/agilis-piezo-rotation-stages-with-conex-controller.

[150] *Raspberry Pi 4*. Accessed: 2022-10-02. URL: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/.

[151] *Adafruit DC Stepper Motor HAT for Raspberry Pi - Mini Kit*. Accessed: 2022-10-02. URL: https://www.adafruit.com/product/2348.

[152] *tracespace*. Accessed: 2022-10-02. URL: https://github.com/tracespace/tracespace.

[153] *RS-15 Series: 15W Single Output Switching Power Supply*. Mean Well, Inc.

[154] *PT1B - 1" Translation Stage with 1/4"-170 Adjustment Screw, 1/4"-20 Taps*. Accessed: 2022-10-02. URL: https://www.thorlabs.us/thorProduct.cfm?partNumber=PT1B.

[155] *yaqd-thorlabs*. Accessed: 2022-10-02. URL: https://pypi.org/project/yaqd-thorlabs.

[156] *yaqd-newport*. Accessed: 2022-10-02. URL: https://pypi.org/project/yaqd-newport.

[157] *adafruit-circuitpython-motorkit*. Accessed: 2022-10-02. URL: https://pypi.org/project/adafruit-circuitpython-motorkit.

[158] *adafruit-circuitpython-motor*. Accessed: 2022-10-02. URL: https://pypi.org/project/adafruit-circuitpython-motor.

[159] *gpiozero*. Accessed: 2022-10-02. URL: https://pypi.org/project/gpiozero.

[160] *CARBIDE Unibody-Design Femtosecond Lasers for Industry and Science*. Accessed: 2022-10-03. URL: https://lightcon.com/product/carbide-femtosecond-lasers/.

[161] *MC2000B - Optical Chopper System with MC1F10HP 10/100 Slot (36°) Chopper Blade, 120 VAC Power Cord*. Accessed: 2022-10-03. URL: https://www.thorlabs.us/thorProduct.cfm?partNumber=MC2000B.

[162] *50 mm (1.97") TravelMax Translation Stage with Optical Encoder*. Accessed: 2022-10-03. URL: https://www.thorlabs.us/thorProduct.cfm?partNumber=LNR502E.

[163] *iHR Series*. Accessed: 2022-10-02. URL: https://www.horiba.com/int/scientific/products/detail/action/show/Product/ihr-series-1590/.

[164] *Octave Express 84XX CompuScope*. Accessed: 2022-10-03. URL: https://www.gage-applied.com/digitizers/16-bit/pcie/digitizer-compuscope-octave-express-84xx.htm.

[165] *yaqd-attune*. Accessed: 2022-10-02. URL: https://pypi.org/project/yaqd-attune.

[166] *Spectrometers Qmini*. Accessed: 2022-10-03. URL: https://www.broadcom.com/products/optical-sensors/spectrometers/spectrometers-qmini.

[167] *yaqd-ocean-optics*. Accessed: 2022-10-02. URL: https://pypi.org/project/yaqd-ocean-optics.

[168] Frederic Fournier, Rui Guo, Elizabeth M. Gardner, Paul M. Donaldson, Christian Loeffeld, Ian R. Gould, Keith R. Willison, and David R. Klug. "Biological and Biomedical Applications of Two-Dimensional Vibrational Spectroscopy: Proteomics, Imaging, and Structural Analysis". In: *Accounts of Chemical Research* 42.9 (Sept. 2009), pp. 1322–1331. DOI: 10.1021/ar900074p. URL: https://doi.org/10.1021/ar900074p.

[169] Megan K. Petti, Justin P. Lomont, Michał Maj, and Martin T. Zanni. "Two-Dimensional Spectroscopy Is Being Used to Address Core Scientific Questions in Biology and Materials Science". In: *The Journal of Physical Chemistry B* 122.6 (Feb. 2018), pp. 1771–1780. DOI: 10.1021/acs.jpcb.7b11370. URL: https://doi.org/10.1021/acs.jpcb.7b11370.

[170] J.W. Gibbs. *Elementary Principles in Statistical Mechanics: Developed with Especial Reference to the Rational Foundations of Thermodynamics*. C. Scribner's sons, 1902. URL: https://books.google.com/books?id=IGMSAAAAIAAJ.

[171] Maxim F. Gelin, Dassia Egorova, and Wolfgang Domcke. "Efficient Calculation of Time- and Frequency-Resolved Four-Wave-Mixing Signals". In: *Accounts of Chemical Research* 42.9 (Sept. 2009), pp. 1290–1298. DOI: 10.1021/ar900045d. URL: http://dx.doi.org/10.1021/ar900045d.

[172] Duckhwan Lee and Andreas C. Albrecht. "A Unified View of Raman, Resonance Raman, and Fluorescence Spectroscopy (and their Analogues in Two-Photon Absorption". In: *Advances in infrared and Raman Spectroscopy*. Ed. by R. J. H. Clark and R. E. Hester. 1st ed. London; New York, 1985. Chap. 4, pp. 179–213.

[173] Paul Blanchard, Robert L Devaney, and Glen R Hall. "Numerical Methods". In: *Differential Equations*. Third. Thomson Brooks/Cole, 2006. Chap. 7, pp. 627–667.

[174] Wright Group. *NISE: Numerical Integration of the Shrödinger Equation*. 2016. URL: http://github.com/wright-group/NISE.

[175] jiffyclub. *SnakeViz*. 2017. URL: http://jiffyclub.github.io/snakeviz/.

[176] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. "Scalable parallel programming with CUDA". In: *Queue* 6.2 (Mar. 2008), p. 40. DOI: 10.1145/1365490.1365500. URL: http://dx.doi.org/10.1145/1365490.1365500.

[177] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. "PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation". In: *Parallel Computing* 38.3 (Mar. 2012), pp. 157–174. DOI: 10.1016/j.parco.2011.09.001. URL: http://dx.doi.org/10.1016/j.parco.2011.09.001.