auspex Documentation

Release 2019.2

Auspex Developers

Feb 04, 2023

Contents

1 Installation & Requirements **3**

2 Qubit Experiments **5**

3 Genealogy and Etymology **7**

Python Module Index **87**

Index **89**

Auspex, the *automated system for python-based experiments*, is a framework for performing laboratory measurements. Auspex was developed by a group that primarily performs measurements on superconducting qubits and magnetic memory elements, but its underpinnings are sufficiently general to allow for extension to arbitrary equipment and experiments. Using several layers of abstraction, we attempt to meet the following goals:

1. Instrument drivers should be easy to write.
2. Measurement code should be flexible and reusable.
3. Data acquisition and processing should be asynchronous and reconfigurable.
4. Experiments should be adaptive, not always pre-defined.
5. Experiments should be concerned with information density, and not limited by the convenience of rectilinear sweeps.

A number of inroads towards satisfying points (1) and (2) are made by utilizing metaprogramming to reduce boilerplate code in *Instrument* drivers and to provide a versatile framework for defining an *Experiment*. For (3) we make use of the python *asyncio* library to create a graph-based measurement *Filter* pipeline through which data passes to be processed, plotted, and written to file. For (4), we attempt to mitigate the sharp productivity hits associated with experimentors having to monitor and constantly tweak the parameters of an *Experiment*. This is done by creating a simple interface that allows *Sweeps* to refine themselves based on user-defined criterion functions. Finally, for (5) we build in "unstructured" sweeps that work with parameter tuples rather than "linspace" style ranges for each parameter. The combination of (4) and (5) allows us to take beautiful phase diagrams that require far fewer points than would be required in a rectilinear, non-adaptive scheme.

CHAPTER 1

Installation & Requirements

Auspex can be cloned from GitHub:

```
git clone https://github.com/BBN-Q/auspex.git
```

And subsequently installed using pip:

```
cd auspex
pip install -e .
```

Which will automatically fetch and install all of the requirements. If you are using an anaconda python distribution, some of the requirements should be install with *conda install* (like ruamel_yaml for example). The packages enumerated in *requirements.txt* are required by Auspex.

CHAPTER 2

Qubit Experiments

Auspex is agnostic to the type of experiment being performed, but we include infrastructure for configuring and executing qubit experiments using the gate-level QGL language. In this case, auspex relies on bbndb as a database backend for sharing state and keeping track of configurations. Depending on the experiments being run, one may need to install a number of additional driver libraries.

If you're running on a system with a low file descriptor limit you may see a *ulimit* error when trying to run or simulate experiments. This will look like a *too many files error* in python. This stems from ZMQ asynchronously opening and closing a large number of files. OSX has a default limit per notebook of 256 open files. You can easily change this number at the terminal before launching a notebook: *ulimit -n 4096* or put this line in your *.bash_prifile*.

CHAPTER 3

Genealogy and Etymology

Auspex is a synonym for an *augur*, whose role was to interpret divine will through a variety of omens. While most researchers rightfully place their faiths in the scientific method, it is not uncommon to relate to the roles of the augur. Auspex incorporates concepts from BBN's *QLab* project as well as from the *pymeasure* project from Cornell University.

Contents:

3.1 Instrument Documentation

3.1.1 Instruments

The *Instrument* class is designed to dramatically reduce the amount of boilerplate code required for defining device drivers. The following (from *picosecond.py*) amounts to the majority of the driver definition:

```
class Picosecond10070A(SCPIInstrument):
    """Picosecond 10070A Pulser"""
    amplitude      = FloatCommand(scpi_string="amplitude")
    delay          = FloatCommand(scpi_string="delay")
    duration       = FloatCommand(scpi_string="duration")
    trigger_level  = FloatCommand(scpi_string="level")
    period         = FloatCommand(scpi_string="period")
    frequency      = FloatCommand(scpi_string="frequency",
                      aliases=['freq'])
    offset         = FloatCommand(scpi_string="offset")
    trigger_source = StringCommand(scpi_string="trigger",
                      allowed_values=["INT", "EXT", "GPIB"])

    def trigger(self):
        self.interface.write("*TRG")
```

Each of the Commands is converted into the relevant driver code as detailed below.

Commands

The *Command* class variables are parsed by the *MetaInstrument* metaclass, and automatically expanded into setters and getters (as appropriate) and a *property* that gives convenient access to commands. For example, the following *Command*:

```
frequency = FloatCommand(scpi_string='frequency')
```

will be expanded into the following equivalent set of class methods:

```
def get_frequency(self):
    return float(self.interface.query('frequency?'))
def set_frequency(self, value):
    self.interface.write('frequency {:E}'.format(value))
@property
def frequency(self):
    return self.get_frequency()
@frequency.setter
def frequency(self, value):
    self.set_frequency(value)
```

Instruments with consistent command syntax (which number fewer than one might hope) lend themselves to extremely concise drivers. Using additional keyword arguments such as `allowed_values`, `aliases`, and `value_map` allows for more advanced commands to specified without the usual driver fluff. Full documentation can be found in the API reference.

Property Access

Property access gives us a convenient way of interacting with instrument values. In the following example we construct an instance of the `Picosecond10070A` class and fire off a number of pulses:

```
pspl = Picosecond10070A("GPIB0::24::INSTR")

pspl.amplitude = 0.944                    # Using setter
print("Trigger delay is: ", pspl.delay) # Using getter

for dur in 1e-9*np.arange(1, 11, 0.5):
    pspl.duration = dur
    pspl.trigger()
    time.sleep(0.05)
```

Properties present certain risks alongside their convenience: running `instr.falter_slop = 18.0` will produce no errors (since it's perfectly reasonable Python) despite the user having intended to set the `filter_slope` value. As such, we actually lock the class dictionary after parsing and intilization, and will produce errors informing you of your spelling creativities.

3.2 Experiment Documentation

3.2.1 Scripting

Instantiating a few *Instrument* classes as decribed in the relevant documentation provides us with an environment sufficient to perform any sort of measurement. Let us revisit our simple example with a few added instruments, and also add a few software averages to our measurement.:

```
pspl  = Picosecond10070A("GPIB0::24::INSTR") # Pulse generator
mag   = AMI430("192.168.5.109")              # Magnet controller
keith = Keithley2400("GPIB0::25::INSTR")     # Source meter

pspl.amplitude = 0.944 # V
mag.field      = 0.010 # T
time.sleep(0.1) # Stableize

resistance_values = []
for dur in 1e-9*np.arange(1, 11, 0.05):
    pspl.duration = dur
    pspl.trigger()
    time.sleep(0.05)
    resistance_values.append([keith.resistance for i in range(5)])

avg_res_vals = np.mean(resistance, axis=0)
```

We have omitted a number of configuration commands for brevity. The above script works perfectly well if we always perform the exact same measurement, i.e. we hold the field and pulse amplitude fixed but vary its duration. This is normally an unrealistic restriction, since the experimentor more often than not will want to repeat the same fundamental measurement for any number of sweep conditions.

3.2.2 Defining Experiments

Therefore, we recommend that users package their measurements into *Experiments*, which provide the opportunity for re-use.:

```
class SwitchingExperiment(Experiment):
    # Control parameters
    field          = FloatParameter(default=0.0, unit="T")
    pulse_duration = FloatParameter(default=5.0e-9, unit="s")
    pulse_voltage  = FloatParameter(default=0.1, unit="V")

    # Output data connectors
    resistance = OutputConnector()

    # Constants
    samples = 5

    # Instruments, connections aren't made until run_sweeps called
    pspl  = Picosecond10070A("GPIB0::24::INSTR")
    mag   = AMI430("192.168.5.109")
    keith = Keithley2400("GPIB0::25::INSTR")

    def init_instruments(self):
        # Instrument initialization goes here, and is run
        # automatically by run_sweeps

        # Assign methods to parameters
        self.field.assign_method(self.mag.set_field)
        self.pulse_duration.assign_method(self.pspl.set_duration)
        self.pulse_voltage.assign_method(self.pspl.set_voltage)

        # Create hooks for relevant delays
        self.pulse_duration.add_post_push_hook(lambda: time.sleep(0.05))
        self.pulse_voltage.add_post_push_hook(lambda: time.sleep(0.05))
```

(continues on next page)

(continued from previous page)

```
        self.field.add_post_push_hook(lambda: time.sleep(0.1))

    def init_streams(self):
        # Establish the "intrinsic" data dimensions
        # run by run_sweeps.
        ax = DataAxis("samples", range(self.samples))
        self.resistance.add_axis(ax)

    def run(self):
        # This is the inner loop, which is run for each set of
        # sweep parameters by run_sweeps. Data is pushed out
        # to the world through the output connectors.
        pspl.trigger()
        self.resistance.push(keith.resistance)
```

Here the control parameters, data flow, and the central measurement "kernel" have crystallized into separate entities. To run the same experiment as was performed above, we add a *sweep* to the experiment,:

```
# Define a 1D sweep
exp = SwitchingExperiment()
exp.add_sweep(exp.pulse_duration, 1e-9*np.arange(1, 11, 0.05))
exp.run_sweeps()
```

but we can at this point sweep any *Parameter* or a combination thereof:

```
# Define a 2D sweep
exp = SwitchingExperiment()
exp.add_sweep(exp.field, np.arange(-0.01, 0.015, 0.005))
exp.add_sweep(exp.pulse_voltage, np.linspace(0.1, 1.0, 20))
exp.run_sweeps()
```

These sweeps can be based on *Parameter* tuples in order to accomodate non-rectilinear sweeps, and can be made adaptive by specifying convergence criteria that can modifying the sweeps on the fly. Full documentation is provided here. The time spent writing a full *Experiment* often pays dividends in terms of flexibility.

3.2.3 The Measurement Pipeline

The central `run` method of an *Experiment* should not need to worry about file IO and plotting, nor should we bake common analysis routines (filtering, plotting, etc.) into the code that is only responsible for taking data. Auspex relegates these tasks to the measurement pipeline, which provides dataflow such as that in the image below.

Each block is referred to as a *node* of the experiment graph. Data flow is assumed to be acyclic, though auspex will not save you from yourself if you attempt to circumvent this restriction. Data flow can be one-to-many, but not many-to-one. Certain nodes, such as *correlators* may take multiple inputs, but they are always wired to distinct input connectors. There are a number of advantages to representing processing and analysis as graphs, most of which stem from the ease of reconfiguration. We have even developed a specialized tool, *Quince*, that provides a graphical interfaces for modifying the contents and connectivity of the graph.

Finally, we stress data is streamed asynchronously across the graph. Each node processes data as it is received, though many types of nodes must wait until enough data has accumulated to perform their stated functions.

Connectors, Streams, and Descriptors

OutputConnectors are "ports" on the experiments through which all measurement data flows. As mentioned above, a single *OutputConnector* can send data to any number of subsequent filter nodes. Each such connection consists of a

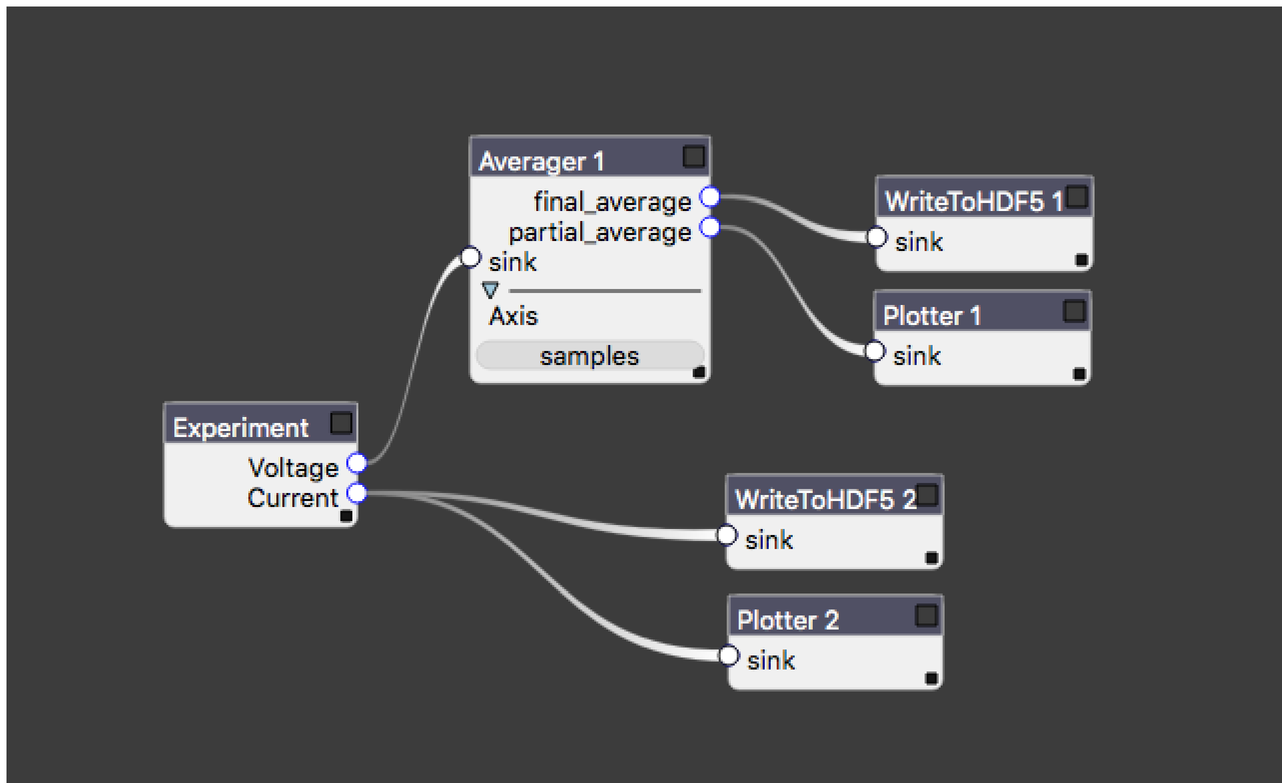


Fig. 1: An example of measurement dataflow starting from the *Experiment* at left.

DataStream, which contains an asyncio-compatible queue for shuttling data. Since data is streamed, rather than passed as tidy arrays, all data streams are described by a *DataStreamDescriptor* that describes the dimensionality of the data.

A *DataStreamDescriptor* contains a list of *Axes*, which contain a list of the points in the axis. These axes may be "intrinisic," as in the case of the `DataAxis("samples", range(self.samples))` axis added in the `init_streams` method above. An axis may also be a *SweepAxis*, which is added to all descriptors automatically when you add a sweep to an experiment. Thus, assuming we're using the 2D sweep from the example above, data emitted by the experiment is described by the following axes,:

```
[DataAxis("samples", range(5)),
SweepAxis("field", np.arange(-0.01, 0.015, 0.005)),
SweepAxis("pulse_voltage", np.linspace(0.1, 1.0, 20))]
```

Importantly, there is no requirement for rectilinear sweeps, which was one of our design goals. Back on the experiment graph, each node can modify this *DataStreamDescriptor* for downstream data: e.g. an averaging node (such as that in the figure above) that is set to average over the "samples" axis will send out data described by the axes

```
[SweepAxis("field", np.arange(-0.01, 0.015, 0.005)),
SweepAxis("pulse_voltage", np.linspace(0.1, 1.0, 20))]
```

Nodes such as data writers are, of course, written such that they store all of the axis information alongside the data. To define our filter pipeline we instantiate the nodes and then we pass a list of "edges" of the graph to the experiment

```
exp   = SwitchingExperiment()
write = WriteToHDF5("filename.h5")
avg   = Averager(axis='samples')
links = [(exp.resistance, avg.sink),
```

(continues on next page)

(continued from previous page)

```
        (avg.final_average, write.sink)]
exp.set_graph(links)
```

Since this is rather tedious to do manually for large sets of nodes, tools like *Quince* and *PyQLab* can be used to lessen the burden.

3.2.4 Running Experiments in Jupyter Notebooks

You should do this.

3.3 Sweep Documentation

Sweeping in measurement software is often a very rigid afair. Typical parameter sweeps take the form

```
for this in range(10):
    for that in np.linspace(30.3, 100.6, 27):
        set_this(this)
        set_that(that)
        measure_everything()
```

A few questions arise at this point:

1. Is there any reason to use a rectangular grid of points?
2. Am I wasting time if features aren't uniformly distributed over this grid?
3. What if our range wasn't sufficient to capture the desired data?
4. What if we didn't get good enough statistics?

To tackle (1), there are some clear reasons to measure on a rectilinear grid. First of all, it is extremely convenient. Also, if you expect that regions of interest (ROI) are distributed evenly across your measurement domain then this seems like a reasonable choice. Finally there are simple aesthetic considerations: image plots look much better when they are fill a rectangular domain rather than leaving swaths of NaNs strewn the periphery of your image.

Point (2) is really an extension of (1): if you are looking at data that follows `sin(x)*sin(y)` then the information density is practically constant across the domain. If you are looking at a crooked phase transition in a binary system, then the vast majority of your points will be wasted on regions with very low information content. Take the following phase diagrams for an MRAM cell's switching probability.

3.3.1 Structured Sweeps

3.3.2 Unstructured Sweeps

3.4 Qubit Experiments

Auspex and QGL comprise BBN's qubit measurement softwre stack. Both packages utilize the underlying database schema provided by bbndb that allows them to easily share state and allows the user to take advantage of versioned measurement configurations.

3.4.1 Important Changes in the BBN-Q Software Ecosystem

There has been a recent change in how we do things:

1. Both Auspex and QGL now utilize the bbndb backend for configuration information. YAML has been completely dropped as it was too freeform and error prone.
2. HDF5 has been dropped as the datafile format for auspex and replaced with a simple numpy-backed binary format with metadata.
3. HDF5 has also been dropped as the sequence file format for our APS1 and APS2 arbitrary pulse sequencers, a simple binary format prevails here as well.
4. The plot server and client have been decoupled from the main auspex code and now are executed independently. They can even be run remotely!
5. Bokeh has been replaced with bqplot where possible, which has much better data throughput.

3.4.2 Tutorials

The best way to gain experience is to follow through with these tutorials:

Example Q1: Configuring a Channel Library from Scratch

This example notebook shows how, using QGL, one can configure a measurement system. All configuration occurs within the notebook, but interfaces with the QGL `ChannelLibrary` object that uses the `bbndb` package database backend.

© Raytheon BBN Technologies 2018

Creating a Channel Library

The `AWGDir` environment variable is used to indicate where QGL will store it's output sequence files. First we load the QGL module. It defaults to a temporary directory as provided by Python's `tempfile` module.

[1]:
```
from QGL import *
```

```
AWG_DIR environment variable not defined. Unless otherwise specified, using temporary␣
↪directory for AWG sequence file outputs.
```

Next we instantiate the channel library. By default `bbndb` will use an sqlite database at the location specified by the `BBN_DB` environment variabe, but we override this behavior below in order to use a temporary in memory database for testing purposes.

[2]:
```
cl = ChannelLibrary(":memory:")
```

```
Creating engine...
```

The channel library has a number of convenience functions defined for create instruments and qubits, as well as functions to define the relationships between them. Let us create a qubit first:

[3]:
```
q1 = cl.new_qubit("q1")
```

Later on we will see how to save and load other versions of the channel library, so remember that *this reference will become stale if other library versions are loaded*. After creation it is safest to refer to channels using keyword syntax on the channel library, i.e. `cl["q1"]`. We'll discuss this more later. Now we create some instrumentation: AWGs, a digitizer, and some microwave sources

```
[4]: # Most calls required label and address
aps2_1 = cl.new_APS2("BBNAPS1", address="192.168.5.101")
aps2_2 = cl.new_APS2("BBNAPS2", address="192.168.5.102")
dig_1  = cl.new_X6("X6_1", address=0)
```

There is more general syntax for arbitrary instruments:

```
[5]: # Label, instrument type, address, and an additional config parameter
h1 = cl.new_source("Holz1", "HolzworthHS9000", "HS9004A-009-1", power=-30)
h2 = cl.new_source("Holz2", "HolzworthHS9000", "HS9004A-009-2", power=-30)
```

Now we want to define which instruments control what.

```
[6]: # Qubit q1 is controlled by AWG aps2_1, and uses microwave source h1
cl.set_control(q1, aps2_1, generator=h1)
# Qubit q1 is measured by AWG aps2_2 and digitizer dig_1, and uses microwave source h2
cl.set_measure(q1, aps2_2, dig_1.ch(1), generator=h2)
# The AWG aps2_1 is the master AWG, and distributes a synchronization trigger on its␣
↪second marker channel
cl.set_master(aps2_1, aps2_1.ch("m2"))
```

These objects are linked to one another, and belong to a relational database. Therefore once can easily drill through the heirarchy using typical "dot" attribute access. i.e. we can configure the sidebanding of q1 using the following:

```
[7]: cl["q1"].measure_chan.frequency = 10e6
```

All of the object above have been added to the current database session, but must be committed in order to be made permanent. That can be done as follows:

```
[8]: cl.commit()
```

At this point the channel database is automatically saved to the "working" copy. All of the current channel libraries can be listed (along with their ID and date stamp) with:

```
[9]: cl.ls()
```

```
<IPython.core.display.HTML object>
```

The channel library will attempt to prevent you from creating redundant objects, e.g.:

```
[10]: q1 = cl.new_qubit("q1")
```

```
A database item with the name q1 already exists. Updating parameters of this existing␣
↪item instead.
```

```
[11]: cl.set_measure(q1, aps2_2, dig_1.ch(1), generator=h2)
```

```
The measurement M-q1 already exists: using this measurement.
The Receiver trigger ReceiverTrig-q1 already exists: using this channel.
```

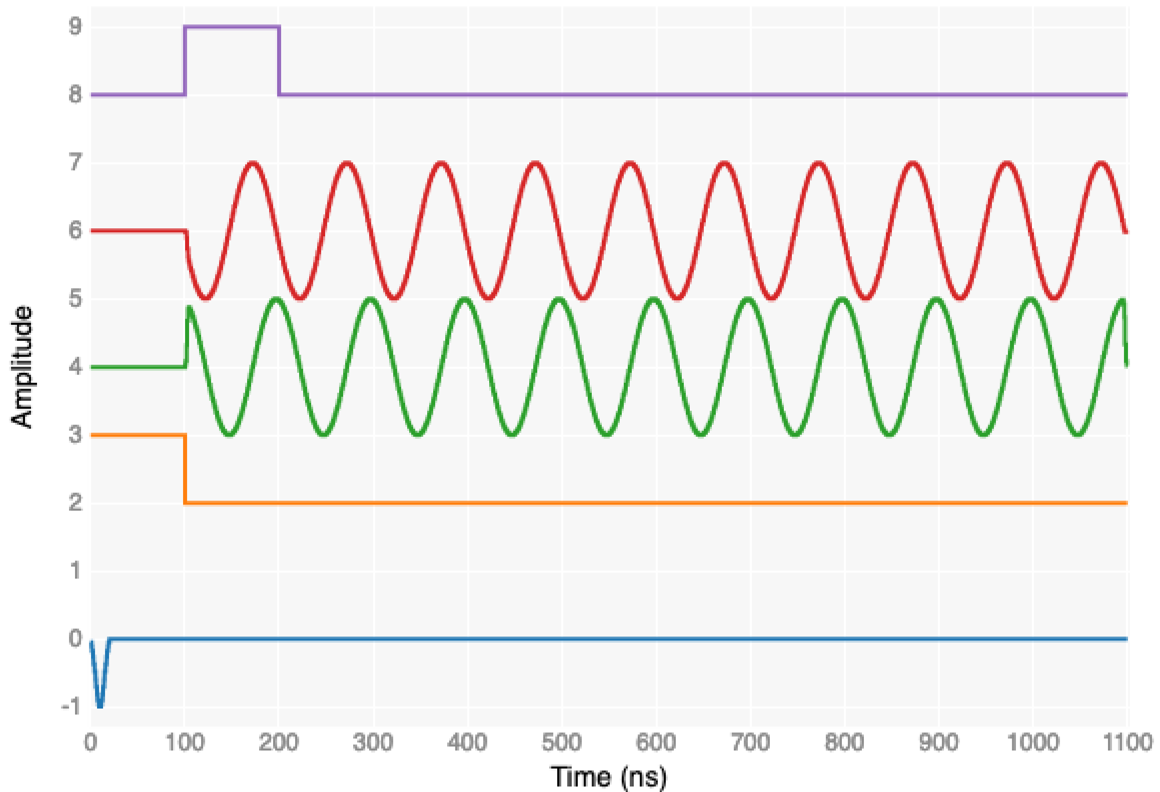
Let's plot the pulse files for a Rabi sequence (giving a directory for storing AWG information).

```
[12]: q1.measure_chan.pulse_params['length'] = 1000e-9
q1.measure_chan.trig_chan.pulse_params['length'] = 100e-9
```

```
[13]: plot_pulse_files(RabiAmp(cl["q1"], np.linspace(-1, 1, 11)), time=True)
```

```
Compiled 11 sequences.
```

```
VBox(children=(IntSlider(value=1, description='Segment', max=11, min=1),␣
↪Figure(animation_duration=50, axes=[A...
```




```
[ ]:
```

Example Q2: Save and Loading Channel Library Versions

This example notebook shows how one may save and load versions of the channel library.

© Raytheon BBN Technologies 2018

Saving Channel Library Versions

We initialize the channel library as shown in tutorial *Q1*:

[2]:
```
from QGL import *

cl = ChannelLibrary(":memory:")
q1 = cl.new_qubit("q1")
aps2_1 = cl.new_APS2("BBNAPS1", address="192.168.5.101")
aps2_2 = cl.new_APS2("BBNAPS2", address="192.168.5.102")
dig_1  = cl.new_X6("X6_1", address=0)
h1 = cl.new_source("Holz1", "HolzworthHS9000", "HS9004A-009-1", power=-30)
h2 = cl.new_source("Holz2", "HolzworthHS9000", "HS9004A-009-2", power=-30)
cl.set_control(q1, aps2_1, generator=h1)
cl.set_measure(q1, aps2_2, dig_1.ch(1), generator=h2)
cl.set_master(aps2_1, aps2_1.ch("m2"))
cl["q1"].measure_chan.frequency = 0e6
cl.commit()
```

```
A database item with the name q1 already exists. Updating parameters of this existing␣
↪item instead.
A database item with the name BBNAPS1 already exists. Updating parameters of this␣
↪existing item instead.
A database item with the name BBNAPS2 already exists. Updating parameters of this␣
↪existing item instead.
A database item with the name X6_1 already exists. Updating parameters of this␣
↪existing item instead.
A database item with the name Holz1 already exists. Updating parameters of this␣
↪existing item instead.
A database item with the name Holz2 already exists. Updating parameters of this␣
↪existing item instead.
```

Let us save this channel library for posterity:

[3]:
```
cl.save_as("NoSidebanding")
```

Now we adjust some parameters and save another version of the channel library

[4]:
```
cl["q1"].measure_chan.frequency = 50e6
cl.commit()
cl.save_as("50MHz-Sidebanding")
```

Maybe we forgot to change something. No worries! We can just update the parameter and create a new copy.

[5]:
```
cl["q1"].pulse_params['length'] = 400e-9
cl.commit()
cl.save_as("50MHz-Sidebanding")
cl.ls()
```

```
<IPython.core.display.HTML object>
```

We see the various versions of the channel library here. Note that the user is *always* modifying the working version of the database: all other versions are archival, but they can be *restored* to the current working version as shown below.

Loading Channel Library Versions

Let us load a previous version of the channel library, noting that the former value of our parameter is restored in the working copy. **CRUCIAL POINT**: do not use the old reference `q1`, which is no longer pointing to the database since the working db has been replaced with the saved version. Instead use dictionary access `cl["q1"]` on the channel library to return the first qubit:

[6]:
```
cl.load("NoSidebanding")
cl["q1"].measure_chan.frequency
```

[6]:
```
0.0
```

Now let's load the *second oldest* version of the 50MHz-sidebanding library:

[7]:
```
cl.load("50MHz-Sidebanding", -1)
cl["q1"].pulse_params['length'], cl["q1"].measure_chan.frequency
```

[7]:
```
(2e-08, 50000000.0)
```

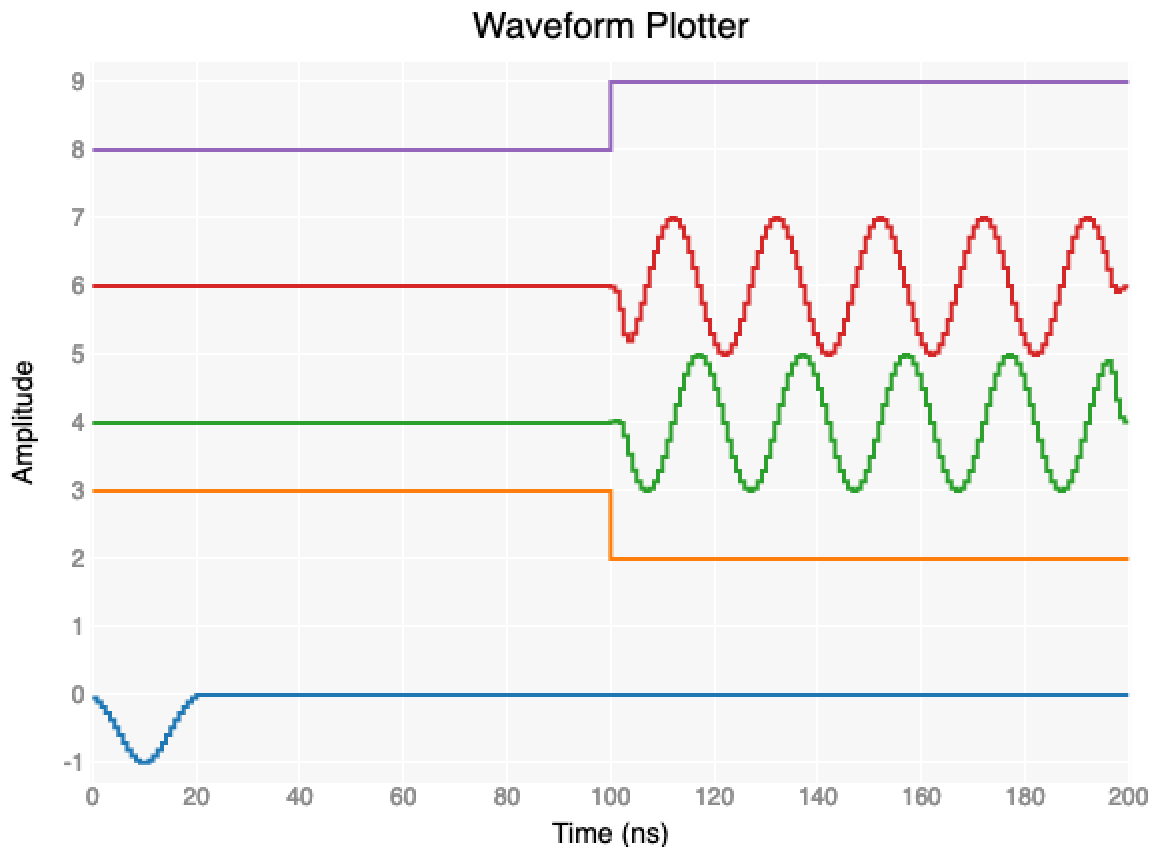
[8]:
```
# q1 = QubitFactory("q1")
plot_pulse_files(RabiAmp(cl["q1"], np.linspace(-1, 1, 11)), time=True)
```

```
Compiled 11 sequences.
```

```
VBox(children=(IntSlider(value=1, description='Segment', max=11, min=1),␣
↪Figure(animation_duration=50, axes=[A...
```

Compiled 11 sequences.

Segment 1



[9]:
```
cl.ls()
```

```
<IPython.core.display.HTML object>
```

[10]:
```
cl.rm("NoSidebanding")
```

[11]:
```
cl.ls()
```

```
<IPython.core.display.HTML object>
```

[12]:
```
cl.rm("50MHz-Sidebanding")
```

[13]:
```
cl.ls()
```

```
<IPython.core.display.HTML object>
```

Example Q3: Managing the Filter Pipeline

This example notebook shows how to use the `PipelineManager` to modify the signal processing on qubit data.

© Raytheon BBN Technologies 2018

We initialize a slightly more advanced channel library:

[1]:
```
from QGL import *

cl = ChannelLibrary(":memory:")

# Create five qubits and supporting hardware
for i in range(5):
    q1 = cl.new_qubit(f"q{i}")
    cl.new_APS2(f"BBNAPS2-{2*i+1}", address=f"192.168.5.{101+2*i}")
    cl.new_APS2(f"BBNAPS2-{2*i+2}", address=f"192.168.5.{102+2*i}")
    cl.new_X6(f"X6_{i}", address=0)
    cl.new_source(f"Holz{2*i+1}", "HolzworthHS9000", f"HS9004A-009-{2*i}", power=-30)
    cl.new_source(f"Holz{2*i+2}", "HolzworthHS9000", f"HS9004A-009-{2*i+1}", power=-
↪30)
    cl.set_control(cl[f"q{i}"], cl[f"BBNAPS2-{2*i+1}"], generator=cl[f"Holz{2*i+1}"])
    cl.set_measure(cl[f"q{i}"], cl[f"BBNAPS2-{2*i+2}"], cl[f"X6_{i}"][1],␣
↪generator=cl[f"Holz{2*i+2}"])

cl.set_master(cl["BBNAPS2-1"], cl["BBNAPS2-1"].ch("m2"))
cl.commit()
```

```
AWG_DIR environment variable not defined. Unless otherwise specified, using temporary␣
↪directory for AWG sequence file outputs.
```

Creating the Default Filter Pipeline

[2]:
```
from auspex.qubit import *
```

```
auspex-WARNING: 2019-04-04 13:38:24,127 ----> You may not have the libusb backend:␣
↪please install it!
auspex-WARNING: 2019-04-04 13:38:24,298 ----> Could not load channelizer library;␣
↪falling back to python methods.
```

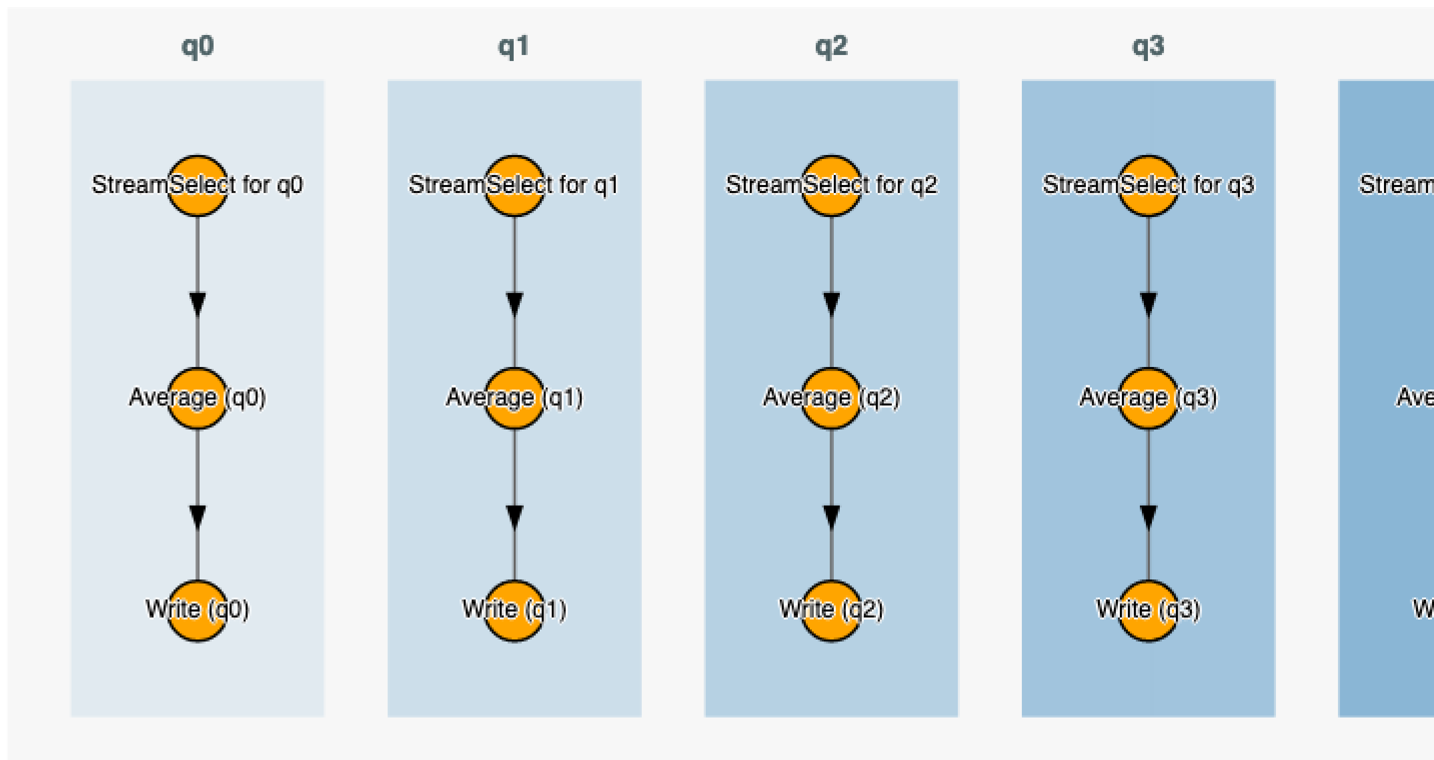
The `PipelineManager` is analogous to the `ChannelLibrary` insomuchas it provides the user with an interface to programmatically modify the filter pipeline, and to save and load different versions of the pipeline.

```
[3]: pl = PipelineManager()
```

```
auspex-INFO: 2019-04-04 13:38:24,641 ----> Could not find an existing pipeline.␣
↪Please create one.
```

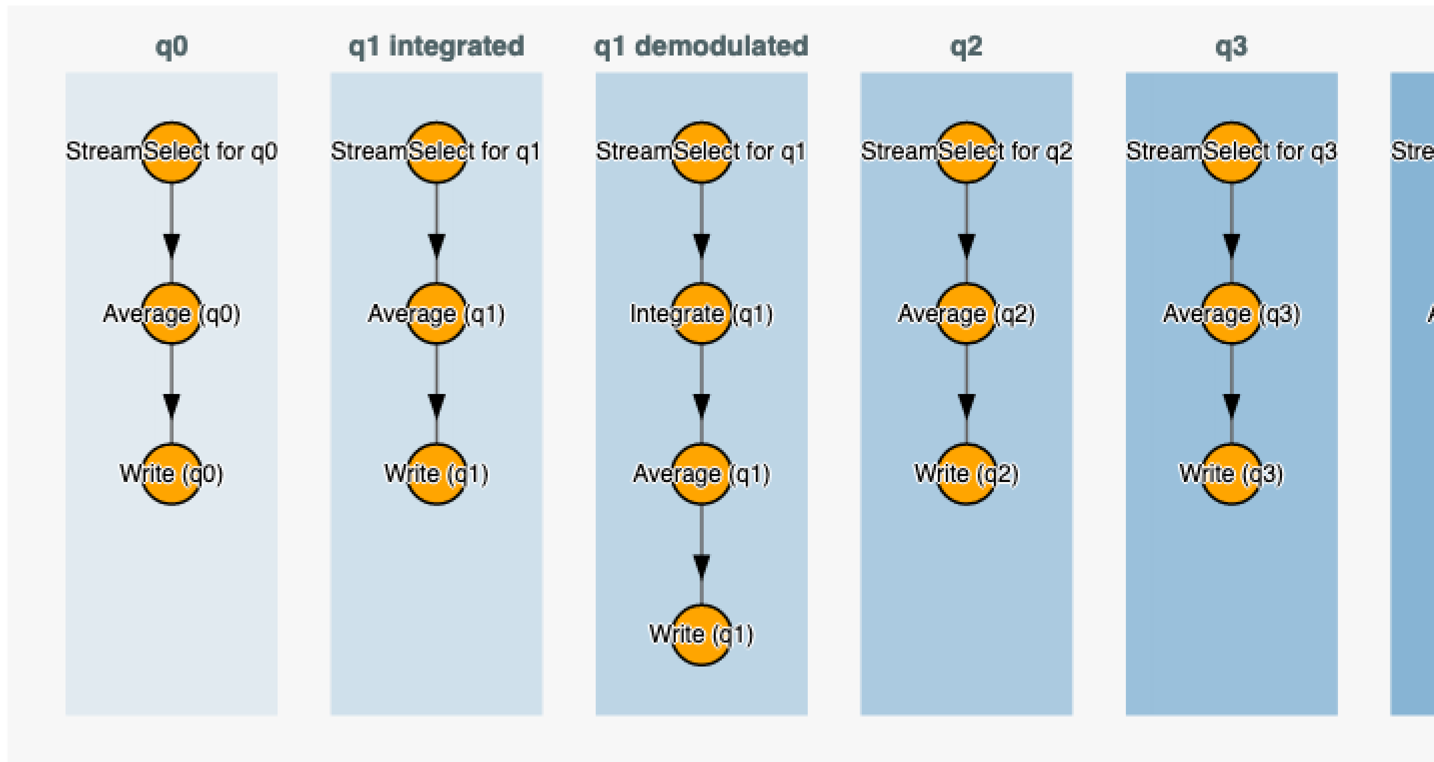
Pipelines are fairly predictable, and will provide some subset of the functionality of demodulating, integrating, average, and writing to file. Some of these can be done on hardware, some in software. The PipelineManager can guess what the user wants for a particular qubit by inspecting which equipment has been assigned to it using the `set_measure` command for the ChannelLibrary. For example, this ChannelLibrary has defined X6-1000M cards for readout, and the description of this instrument indicates that the highest level available stream is integrated. Thus, the PipelineManager automatically inserts the remaining averager and writer.

```
[ ]: pl.create_default_pipeline()
     pl.show_pipeline()
```



Sometimes, for debugging purposes, one may wish to add multiple pipelines per qubit. Additional pipelines can be added explicitly by running:

```
[ ]: pl.add_qubit_pipeline("q1", "demodulated")
     pl.show_pipeline()
```




```
[6]: pl.ls()
```

```
<IPython.core.display.HTML object>
```

We can print the properties of a single node

```
[9]: pl["q1 integrated"].print()
```

```
<IPython.core.display.HTML object>
```

We can print the properties of individual filters or subgraphs:

```
[10]: pl.print("q1 integrated")
```

```
<IPython.core.display.HTML object>
```

Dictionary access is provided to allow drilling down into the pipelines. One can use the specific label of a filter or simple its type in this access mode:

```
[11]: pl["q1 integrated"]["Average"]["Write"].filename = "new.h5"
      pl.print("q1 integrated")
```

```
<IPython.core.display.HTML object>
```

Here uncommitted changes are shown. This can be rectified in the standard way:

```
[12]: cl.commit()
      pl.print("q1 integrated")
```

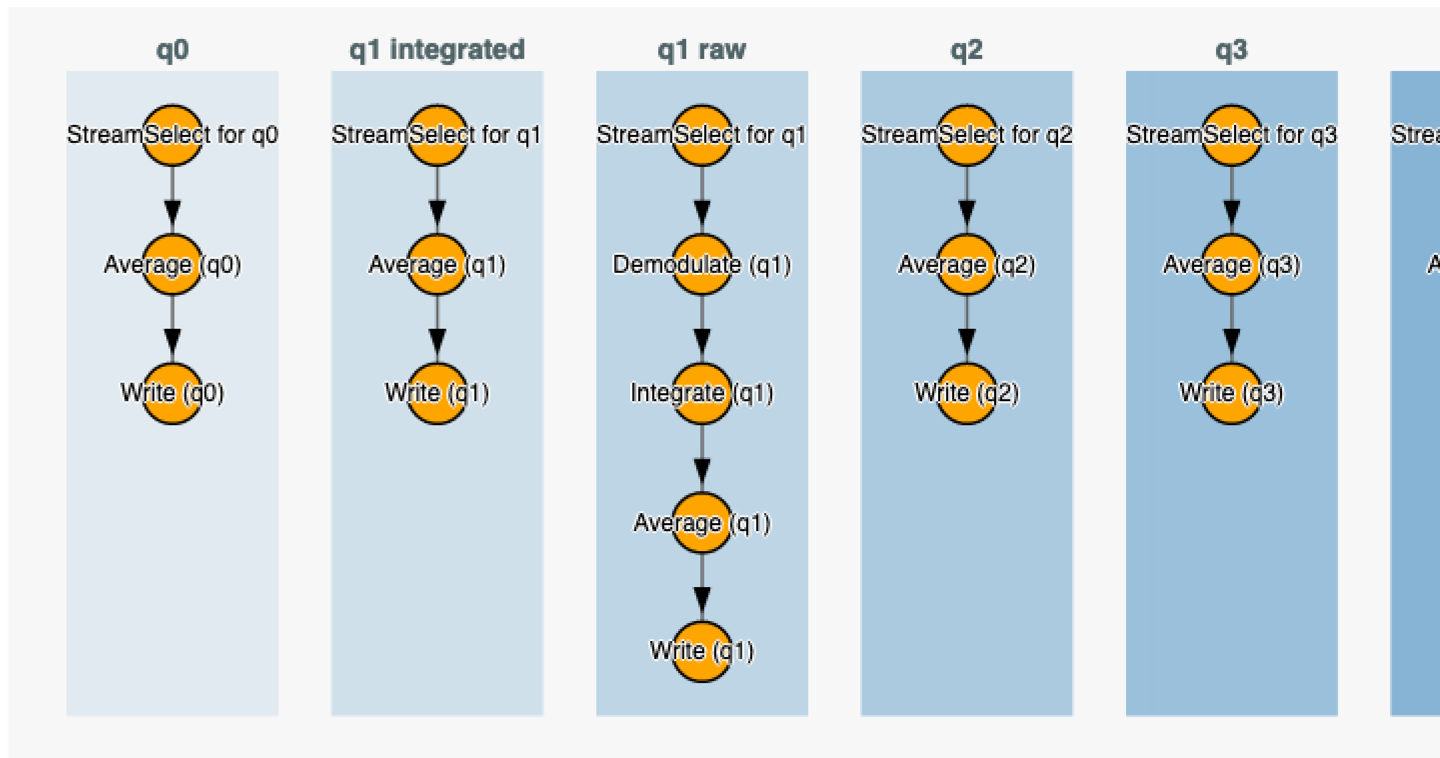
```
<IPython.core.display.HTML object>
```

Programmatic Modification of the Pipeline

Some simple convenience functions allow the use to easily specify complex pipeline structures.

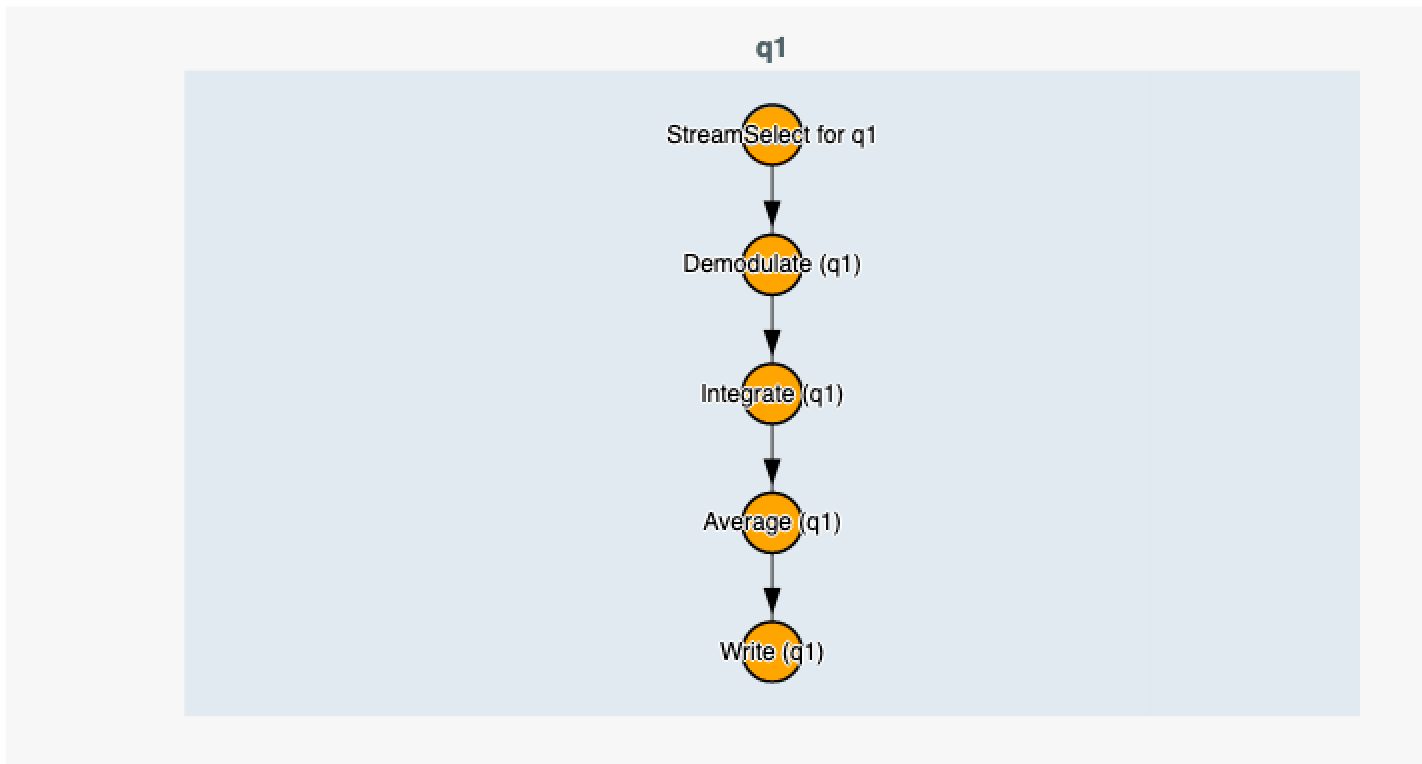
[13]:
```
pl.commit()
pl.save_as("simple")
pl["q1 demodulated"].clear_pipeline()
pl["q1 demodulated"].stream_type = "raw"
pl.recreate_pipeline()
# pl["q1"]["blub"].show_pipeline()
```

[ ]:
```
pl.show_pipeline()
```

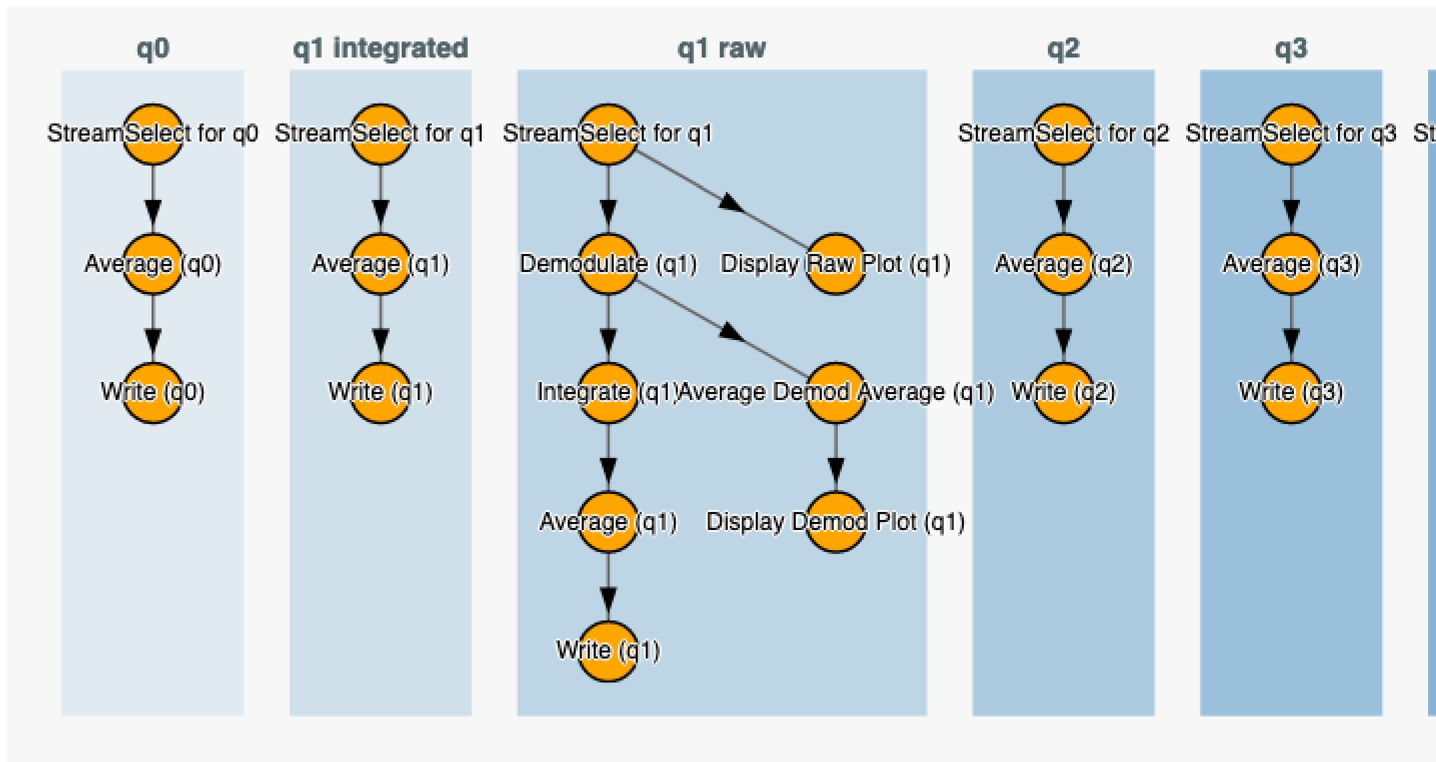


Note the name change. We refer to the pipeline by the stream type of the first element.

[ ]:
```
pl["q1 raw"].show_pipeline()
```



[ ]:
```
pl["q1 raw"].add(Display(label="Raw Plot"))
pl["q1 raw"]["Demodulate"].add(Average(label="Demod Average")).add(Display(label=
↪"Demod Plot"))
pl.show_pipeline()
```

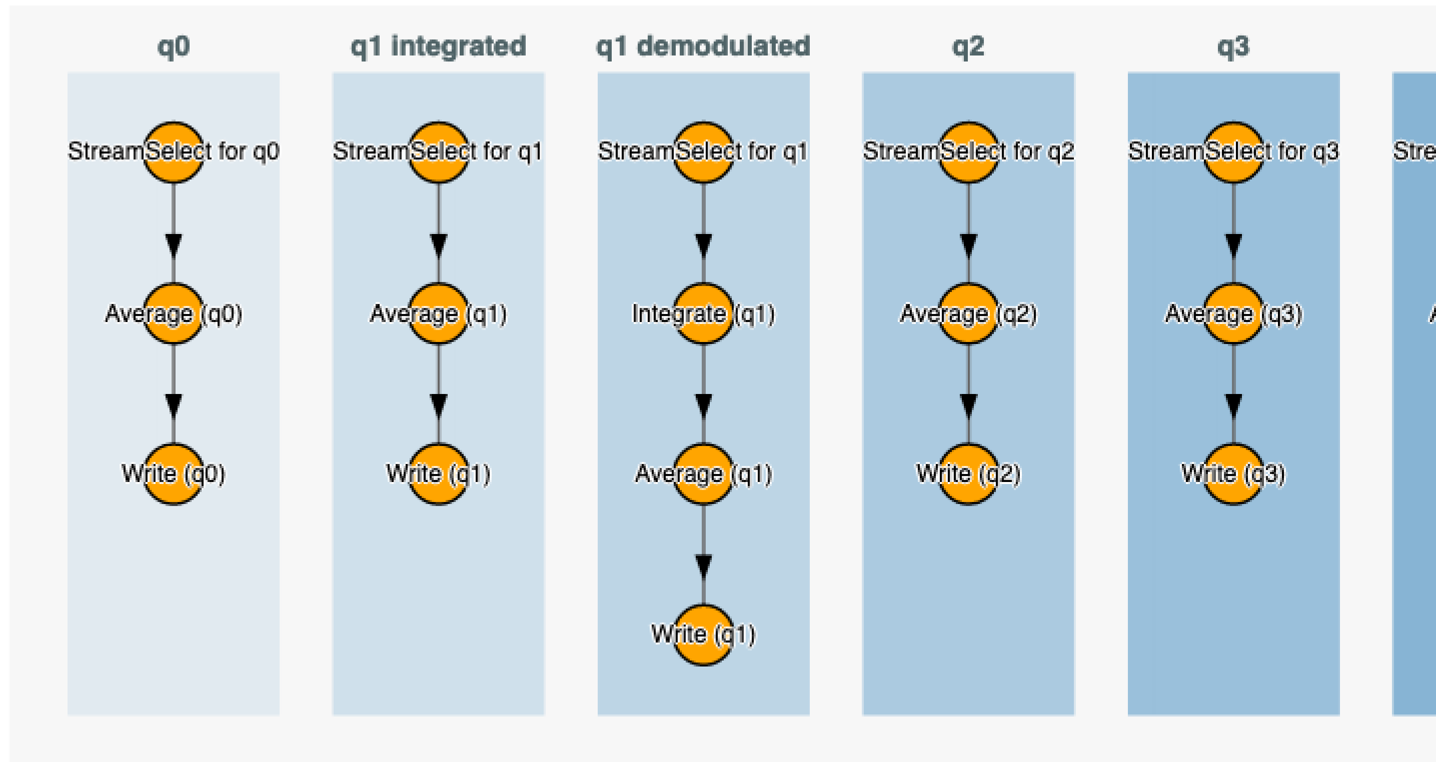


As with the `ChannelLibrary` we can list save, list, and load versions of the filter pipeline.

[17]:
```
pl.session.commit()
pl.save_as("custom")
pl.ls()
```

```
<IPython.core.display.HTML object>
```

[ ]:
```
pl.load("simple")
pl.show_pipeline()
```




```
[19]: pl.ls()
```

```
<IPython.core.display.HTML object>
```

Pipeline examples:

Below are some examples of how more complicated pipelines can be constructed. Defining these as functions allows for quickly changing the structure of the data pipeline depending on the experiment being done. It also improves reproducibility and documents pipeline parameters. For example, to change the pipeline and check its construction,

```
pl = create_tomo_pipeline(save_rr=True)
pl.show_pipeline()
```

Hopefully the examples below will show you some of the more advanced things that can be done with the data pipelines in `Auspex`.

```
[ ]: # a basic pipeline that uses 'raw' data a the beginning of the data processing
def create_standard_pipeline():
    pl = PipelineManager()
    pl.create_default_pipeline(qubits=(cl['q2'],cl['q3']))
    for ql in ['q2', 'q3']:
        qb = cl[ql]
        pl[ql].clear_pipeline()
        pl[ql].stream_type = "raw"
        pl[ql].create_default_pipeline(buffers=False)
        pl[ql].if_freq = qb.measure_chan.autodyne_freq
        pl[ql]["Demodulate"].frequency = qb.measure_chan.autodyne_freq
        pl[ql]["Demodulate"]["Integrate"].simple_kernel = True
        pl[ql]["Demodulate"]["Integrate"].box_car_start = 3e-7
```

(continues on next page)

(continued from previous page)

```
        pl[ql]["Demodulate"]["Integrate"].box_car_stop = 1.3e-6
        #pl[ql]["Demodulate"]["Integrate"].add(Write(label="RR-Writer", groupname=ql+
↪"-int"))
        pl[ql]["Demodulate"]["Integrate"]["Average"].add(Display(label=ql+" - Final␣
↪Average", plot_dims=0))
        pl[ql]["Demodulate"]["Integrate"]["Average"].add(Display(label=ql+" - Partial␣
↪Average", plot_dims=0), connector_out="partial_average")
    return pl

# if you only want to save data integrated with the single-shot filter
def create_integrated_pipeline(save_rr=False, plotting=True):
    pl = PipelineManager()
    pl.create_default_pipeline(qubits=(cl['q2'],cl['q3']))
    for ql in ['q2', 'q3']:
        qb = cl[ql]
        pl[ql].clear_pipeline()
        pl[ql].stream_type = "integrated"
        pl[ql].create_default_pipeline(buffers=False)
        pl[ql].kernel = f"{ql.upper()}_SSF_kernel.txt"
        if save_rr:
            pl[ql].add(Write(label="RR-Writer", groupname=ql+"-rr"))
        if plotting:
            pl[ql]["Average"].add(Display(label=ql+" - Final Average", plot_dims=0))
            pl[ql]["Average"].add(Display(label=ql+" - Partial Average", plot_dims=0),
↪ connector_out="partial_average")

    return pl

# create to single-shot fidelity pipelines for two qubits
def create_fidelity_pipeline():
    pl = PipelineManager()
    pl.create_default_pipeline(qubits=(cl['q2'],cl['q3']))
    for ql in ['q2', 'q3']:
        qb = cl[ql]
        pl[ql].clear_pipeline()
        pl[ql].stream_type = "raw"
        pl[ql].create_default_pipeline(buffers=False)
        pl[ql].if_freq = qb.measure_chan.autodyne_freq
        pl[ql]["Demodulate"].frequency = qb.measure_chan.autodyne_freq
        pl[ql].add(FidelityKernel(save_kernel=True, logistic_regression=False, set_
↪threshold=True, label=f"Q{ql[-1]}_SSF"))
        pl[ql]["Demodulate"]["Integrate"].simple_kernel = True
        pl[ql]["Demodulate"]["Integrate"].box_car_start = 3e-7
        pl[ql]["Demodulate"]["Integrate"].box_car_stop = 1.3e-6
        #pl[ql]["Demodulate"]["Integrate"].add(Write(label="RR-Writer", groupname=ql+
↪"-int"))
        pl[ql]["Demodulate"]["Integrate"]["Average"].add(Display(label=ql+" - Final␣
↪Average", plot_dims=0))
        pl[ql]["Demodulate"]["Integrate"]["Average"].add(Display(label=ql+" - Partial␣
↪Average", plot_dims=0), connector_out="partial_average")
    return pl

# optionally save the demoded data
def create_RR_pipeline(plot=False, write_demods=False):
    pl = PipelineManager()
    pl.create_default_pipeline(qubits=(cl['q2'],cl['q3']))
    for ql in ['q2', 'q3']:
```

(continues on next page)

(continued from previous page)

```
        qb = cl[ql]
        pl[ql].clear_pipeline()
        pl[ql].stream_type = "raw"
        pl[ql].create_default_pipeline(buffers=False)
        pl[ql].if_freq = qb.measure_chan.autodyne_freq
        pl[ql]["Demodulate"].frequency = qb.measure_chan.autodyne_freq
        if write_demods:
            pl[ql]["Demodulate"].add(Write(label="demod-writer", groupname=ql+"-demod
↪"))
        pl[ql]["Demodulate"]["Integrate"].simple_kernel = True
        pl[ql]["Demodulate"]["Integrate"].box_car_start = 3e-7
        pl[ql]["Demodulate"]["Integrate"].box_car_stop = 1.3e-6
        pl[ql]["Demodulate"]["Integrate"].add(Write(label="RR-Writer", groupname=ql+"-
↪int"))
        if plot:
            pl[ql]["Demodulate"]["Integrate"]["Average"].add(Display(label=ql+" -␣
↪Final Average", plot_dims=0))
            pl[ql]["Demodulate"]["Integrate"]["Average"].add(Display(label=ql+" -␣
↪Partial Average", plot_dims=0), connector_out="partial_average")
    return pl

# save everything... using data buffers instead of writing to file
def create_full_pipeline(buffers=True):
    pl = PipelineManager()
    pl.create_default_pipeline(qubits=(cl['q2'],cl['q3']), buffers=True)
    for ql in ['q2', 'q3']:
        qb = cl[ql]
        pl[ql].clear_pipeline()
        pl[ql].stream_type = "raw"
        pl[ql].create_default_pipeline(buffers=buffers)
        if buffers:
            pl[ql].add(Buffer(label="raw_buffer"))
        else:
            pl[ql].add(Write(label="raw-write", groupname=ql+"-raw"))
        pl[ql].if_freq = qb.measure_chan.autodyne_freq
        pl[ql]["Demodulate"].frequency = qb.measure_chan.autodyne_freq
        if buffers:
            pl[ql]["Demodulate"].add(Buffer(label="demod_buffer"))
        else:
            pl[ql]["Demodulate"].add(Write(label="demod_write", groupname=ql+"-demod
↪"))
        pl[ql]["Demodulate"]["Integrate"].simple_kernel = True
        pl[ql]["Demodulate"]["Integrate"].box_car_start = 3e-7
        pl[ql]["Demodulate"]["Integrate"].box_car_stop = 1.6e-6
        if buffers:
            pl[ql]["Demodulate"]["Integrate"].add(Buffer(label="integrator_buffer"))
        else:
            pl[ql]["Demodulate"]["Integrate"].add(Write(label="int_write",␣
↪groupname=ql+"-integrated"))
        pl[ql]["Demodulate"]["Integrate"]["Average"].add(Display(label=ql+" - Final␣
↪Average", plot_dims=0))
        pl[ql]["Demodulate"]["Integrate"]["Average"].add(Display(label=ql+" - Partial␣
↪Average", plot_dims=0), connector_out="partial_average")
    return pl

# A more complicated pipeline with a correlator
# These have to be coded more manually because the correlator needs all the␣
↪correlated channels specified.
```

(continues on next page)

(continued from previous page)

```
# Note that for tomography you're going to want to save the data variance as well,␣
↪though this can be calculated
# after the fact if you save the raw shots (save_rr).
def create_tomo_pipeline(save_rr=False, plotting=True):
    pl = PipelineManager()
    pl.create_default_pipeline(qubits=(cl['q2'],cl['q3']))

    for ql in ['q2', 'q3']:
        qb = cl[ql]
        pl[ql].clear_pipeline()
        pl[ql].stream_type = "integrated"
        pl[ql].create_default_pipeline(buffers=False)
        pl[ql].kernel = f"{ql.upper()}_SSF_kernel.txt"
        pl[ql]["Average"].add(Write(label='var'), connector_out='final_variance')
        pl[ql]["Average"]["var"].groupname = ql + '-main'
        pl[ql]["Average"]["var"].datasetname = 'variance'
        if save_rr:
            pl[ql].add(Write(label="RR-Writer", groupname=ql+"-rr"))
        if plotting:
            pl[ql]["Average"].add(Display(label=ql+" - Final Average", plot_dims=0))
            pl[ql]["Average"].add(Display(label=ql+" - Partial Average", plot_dims=0),
↪ connector_out="partial_average")

    # needed for two-qubit state reconstruction
    pl.add_correlator(pl['q2'], pl['q3'])
    pl['q2']['Correlate'].add(Average(label='corr'))
    pl['q2']['Correlate']['Average'].add(Write(label='corr_write'))
    pl['q2']['Correlate']['Average'].add(Write(label='corr_var'), connector_out=
↪'final_variance')
    pl['q2']['Correlate']['Average']['corr_write'].groupname = 'correlate'
    pl['q2']['Correlate']['Average']['corr_var'].groupname = 'correlate'
    pl['q2']['Correlate']['Average']['corr_var'].datasetname = 'variance'

    return pl
```

Example Q4: Running a Basic Experiment

This example notebook shows how run Auspex qubit experiments using the fake data interface.

© Raytheon BBN Technologies 2018

First we ask auspex to run in dummy mode, which avoids loading instrument drivers.

[1]:
```
import auspex.config as config
config.auspex_dummy_mode = True
```

[ ]:
```
from QGL import *
from auspex.qubit import *
import matplotlib.pyplot as plt
%matplotlib inline
```

Channel library setup

[3]:
```
cl = ChannelLibrary(":memory:")
pl = PipelineManager()
```

(continues on next page)

(continued from previous page)

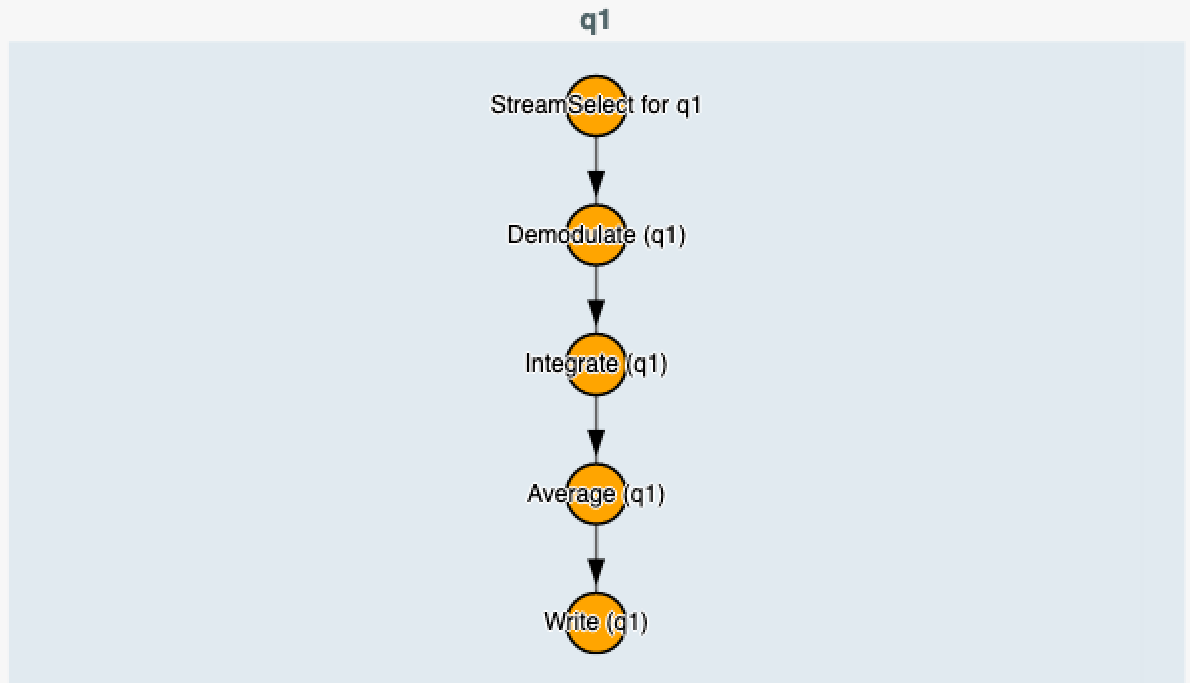
```
q1 = cl.new_qubit("q1")
aps2_1 = cl.new_APS2("BBNAPSa", address="192.168.2.4", trigger_interval=200e-6)
aps2_2 = cl.new_APS2("BBNAPSb", address="192.168.2.2")
dig_1  = cl.new_X6("Dig_1", address="1", sampling_rate=500e6, record_length=1024)
h1 = cl.new_source("Holz_1", "HolzworthHS9000", "HS9004A-009-1", reference='10MHz',␣
↪power=-30)
h2 = cl.new_source("Holz_2", "HolzworthHS9000", "HS9004A-009-2", reference='10MHz',␣
↪power=-30)

cl.set_measure(q1, aps2_1, dig_1.ch(1), trig_channel=aps2_1.ch("m2"), gate=False,␣
↪generator=h1)
cl.set_control(q1, aps2_2, generator=h2)
cl.set_master(aps2_1, aps2_1.ch("m1"))
cl["q1"].measure_chan.frequency = 0e6
cl["q1"].measure_chan.autodyne_freq = 10e6
```

```
auspex-INFO: 2019-04-04 17:36:36,136 ----> Could not find an existing pipeline.␣
↪Please create one.
```

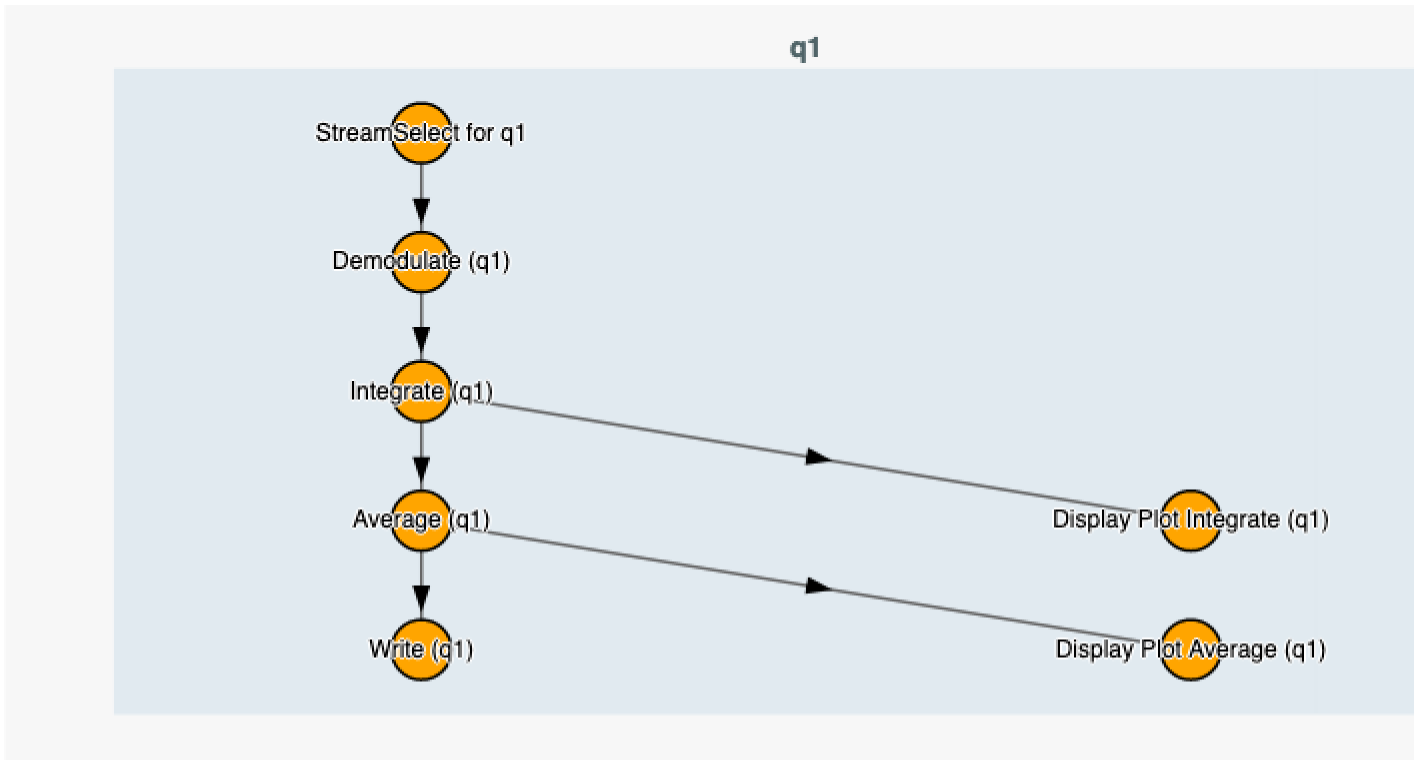
Pipeline setup

```
[4]: pl.create_default_pipeline()
```

```
[ ]: pl["q1"].stream_type = "raw"
     pl.recreate_pipeline(buffers=False)
     pl.show_pipeline()
```



```
[ ]: pl["q1"]["Demodulate"]["Integrate"]["Average"].add(Display(label="Plot Average", plot_
     ↪dims=1), connector_out="partial_average")
     pl["q1"]["Demodulate"]["Integrate"].add(Display(label="Plot Integrate", plot_dims=1))
     pl.show_pipeline()
```



Initialize software demodulation parameters. If these are not properly configured than the `Channelizer` filter will report 'insufficient decimation' or other errors. The integration boxcar parameters are then defined.

```
[19]: demod = pl["q1"]["Demodulate"]
      demod.frequency = cl["q1"].measure_chan.frequency
      demod.decimation_factor = 16
```

```
[20]: integ = pl["q1"]["Demodulate"]["Integrate"]
      integ.box_car_start = 0.2e-6
      integ.box_car_stop= 1.9e-6
```

Taking Fake Data

Now we create a simple experiment, but ask the digitizer to emit fake data of our choosing. This is useful for debugging one's configuration without having access to hardware. The `set_fake_data` method loads the fake dataset into the indicated digitizer's driver. The digitizer driver automatically chooses the nature of it's output depending on whether receiver channels are raw, demodulated, or integrated.

```
[21]: amps = np.linspace(-1,1,51)
      exp = QubitExperiment(RabiAmp(q1,amps),averages=50)
      exp.set_fake_data(dig_1, np.cos(np.linspace(0, 2*np.pi,51)))
      exp.run_sweeps()
```

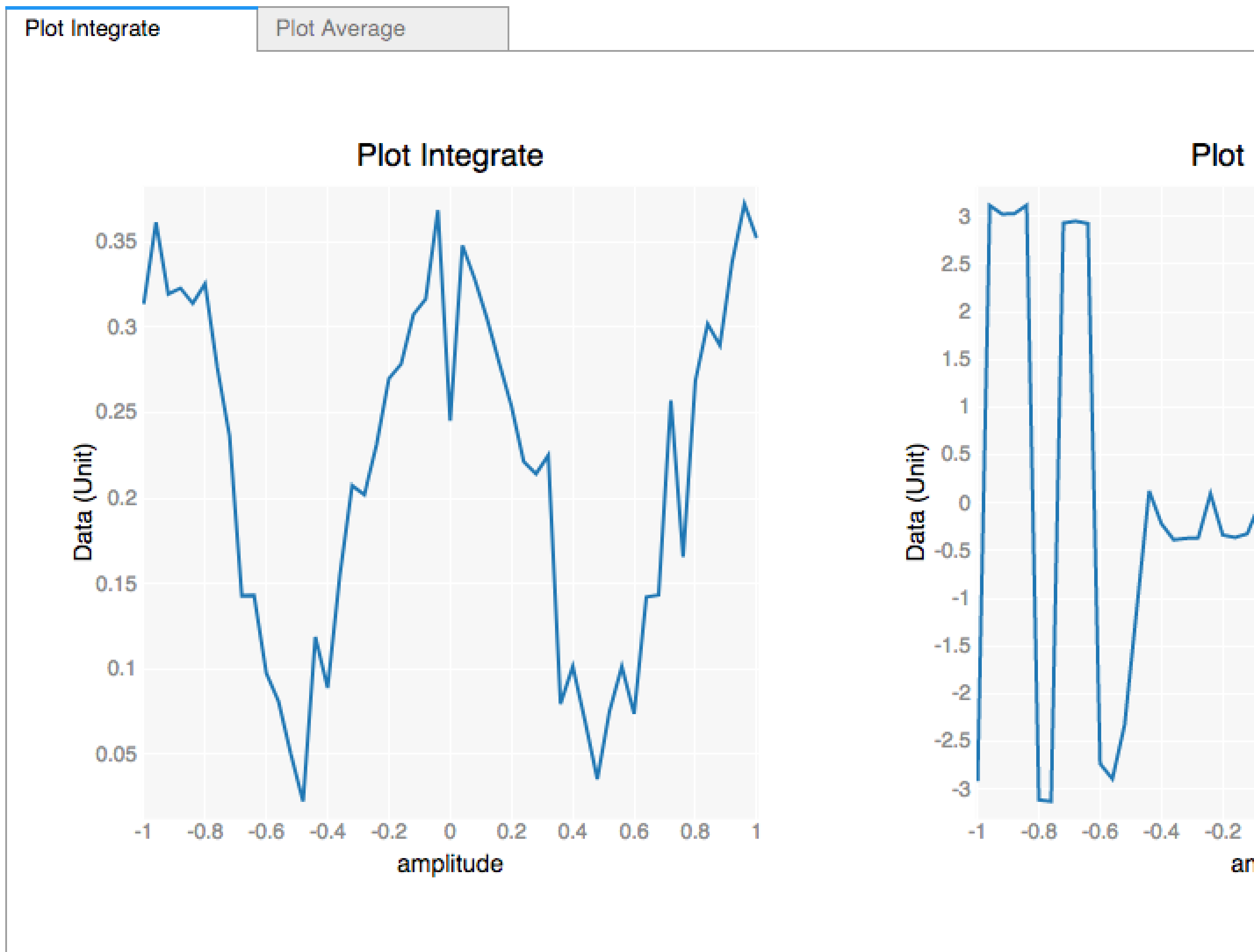
```
Compiled 51 sequences.
Qubit('q1') has 1
```

```
VBox(children=(IntProgress(value=0, bar_style='success', description='Digitizer Data␣
↪q1-raw:', max=652800, sty...
```

```
auspex-INFO: 2019-04-04 17:38:10,603 ----> Connection established to plot server.
```

Plotting the Results in the Notebook

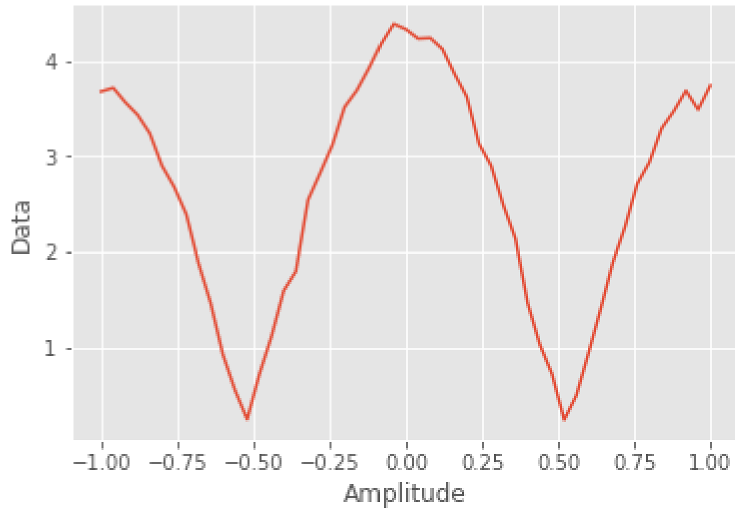
```
[ ]: exp.get_final_plots()
```



Loading the Data Files and Plot Manually

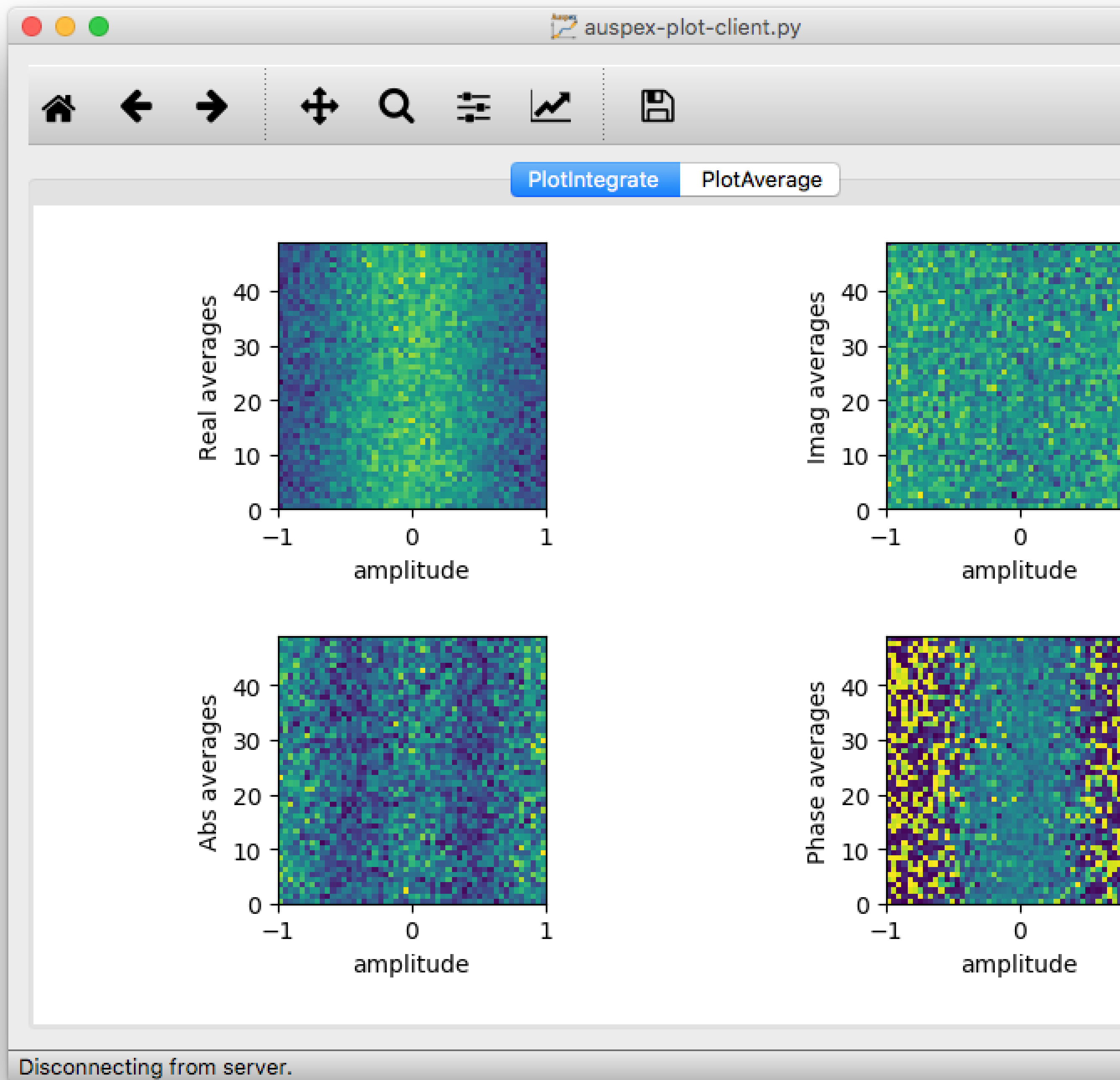
```
[35]: data, desc = exp.outputs_by_qubit["q1"][0].get_data()
```

[36]:
```
plt.plot(desc["amplitude"], np.abs(data))
plt.xlabel("Amplitude"); plt.ylabel("Data");
```



Plotting and the Plot Server & Client

The `Display` nodes in the filter pipeline are turned into live plotters. In the `auspex/utils` directory once can find `auspex-plot-server.py` and `auspex-plot-client.py`. The server, which should be executed first, acts as a data router and can accept multiple clients (and even multiple concurrent Auspex runs). The client polls the server to see whether any plots are available. If so, it grabs their descriptions and constructs a tab for each `Display` filter. The plots are updated as new data becomes available, and will look something like this:



For every execution of `run_sweeps` on an experiment, a new plot will be opened. In the plot client menus, however, the user can close all previous plots as well as choose to 'Auto Close Plots', which closes any previous plots before opening another.

Remote Usage

The plot server and client can be run remotely, as can the Jupyter notebooks one expects to run. By running the following ssh port-forwarding command:

```
ssh -R 7761:localhost:7761 -R 7762:localhost:7762 -L 8889:my.host.com:8888 -l␣
↪username my.host.com
```

You could connect to a remotely running Jupyter notebook on port 8888 locally at port 8889, and then (after starting `auspex-plot-server.py` and `auspex-plot-client.py` on your local machine) watch new plots appear in a live window on your local machine.

This presumes one does not have unfettered network access to the remote host, in which case ssh tunneling is not necessary. Currently the plotter and client assume they are all connecting on localhost, but we will create a convenient interface for their configuration soon.

Monitoring Changes in the Channel Library

The session keeps track of what values have changed without being committed, e.g.:

```
[ ]: cl.session.commit()
cl.session.dirty
```

Everything is in sync with the database. Now we modify some property:

```
[ ]: aps2_1.ch(1).amp_factor = 0.95
cl.session.dirty
```

We see that things have changed that haven't been committed to the database. This can be rectified with another `commit`, or optionally a `rollback`!

```
[ ]: cl.session.rollback()
aps2_1.ch(1).amp_factor
```

Example Q5: Experiment Sweeps

This example notebook shows how to add sweeps to Auspex qubit experiments

© Raytheon BBN Technologies 2018

```
[1]: import auspex.config as config
config.auspex_dummy_mode = True
```

```
[2]: from QGL import *
from auspex.qubit import *
```

```
AWG_DIR environment variable not defined. Unless otherwise specified, using temporary␣
↪directory for AWG sequence file outputs.
```

```
auspex-WARNING: 2019-04-02 14:33:37,114 ----> Could not load channelizer library;␣
↪falling back to python methods.
```

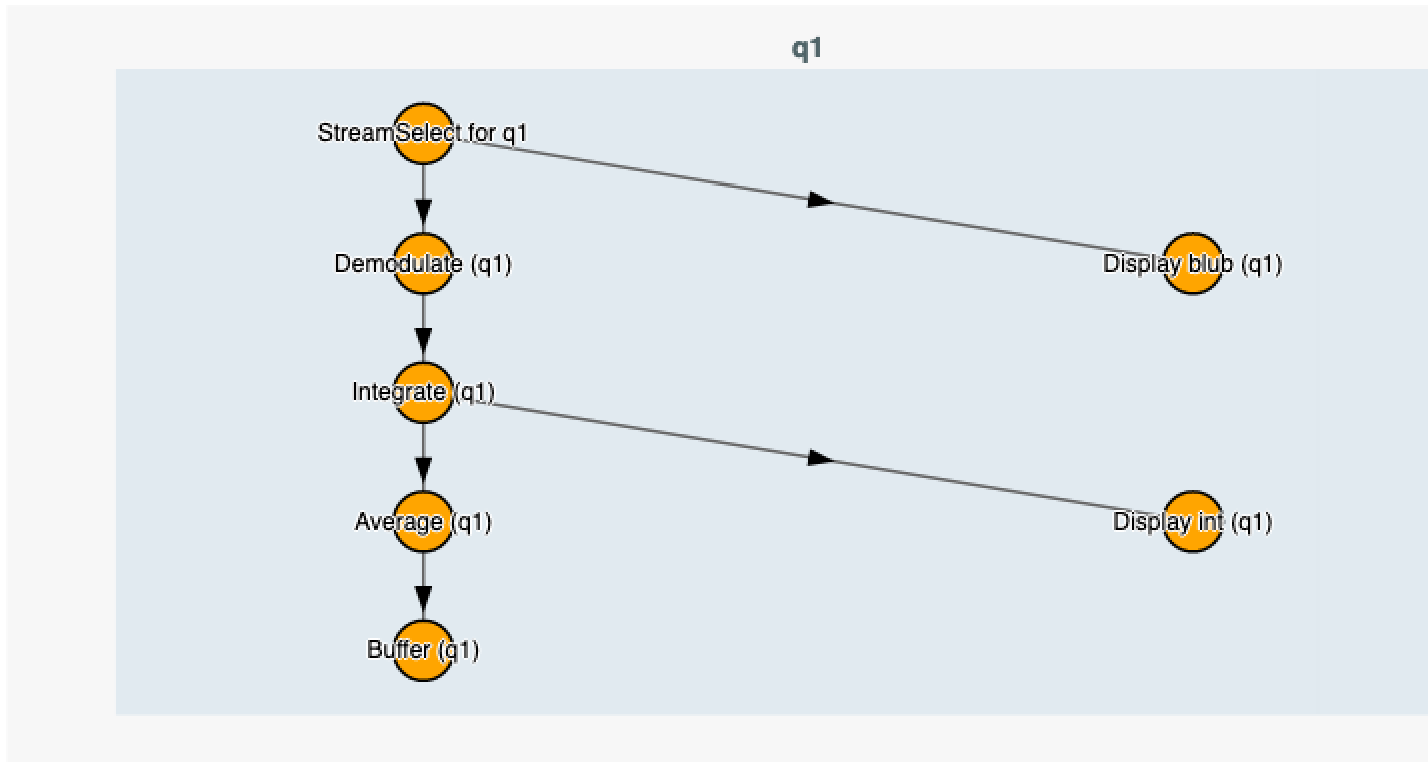
Channel library setup

[3]:
```
cl = ChannelLibrary(":memory:")
pl = PipelineManager()

q1 = cl.new_qubit("q1")
aps2_1 = cl.new_APS2("BBNAPSa", address="192.168.2.4", trigger_interval=200e-6)
aps2_2 = cl.new_APS2("BBNAPSb", address="192.168.2.2")
dig_1  = cl.new_Alazar("Alazar_1", address="1", sampling_rate=500e6, record_
↪length=1024)
h1 = cl.new_source("Holz_1", "HolzworthHS9000", "HS9004A-009-1", reference='10MHz',␣
↪power=-30)
h2 = cl.new_source("Holz_2", "HolzworthHS9000", "HS9004A-009-2", reference='10MHz',␣
↪power=-30)

cl.set_measure(q1, aps2_1, dig_1.ch("1"), trig_channel=aps2_1.ch("m2"), gate=False,␣
↪generator=h1)
cl.set_control(q1, aps2_2, generator=h2)
cl.set_master(aps2_1, aps2_1.ch("m1"))
cl["q1"].measure_chan.frequency = 0e6
cl["q1"].measure_chan.autodyne_freq = 10e6
```

```
auspex-INFO: 2019-04-02 14:33:37,563 ----> Could not find an existing pipeline.␣
↪Please create one.
```

Pipeline setup: **Take Note:** we use the `buffers` keyword argument to automatically generate buffers instead of writers. This is sometimes convenient if you don't require data to be written to file. It becomes immediately available in the notebook after running!

[ ]:
```
pl.create_default_pipeline(buffers=True)
pl["q1"].add(Display(label="blub"))
pl["q1"]["Demodulate"]["Integrate"].add(Display(label="int", plot_dims=1))
pl.show_pipeline()
```



Initialize software demodulation parameters. If these are not properly configured than the `Channelizer` filter will report 'insufficient decimation' or other errors. The integration boxcar parameters are then defined.

[14]:
```
demod = pl["q1"]["Demodulate"]
demod.frequency = cl["q1"].measure_chan.frequency
demod.decimation_factor = 16
```

[15]:
```
integ = pl["q1"]["Demodulate"]["Integrate"]
integ.box_car_start = 0.2e-6
integ.box_car_stop= 1.9e-6
```

Adding experiment sweeps

Once a `QubitExperiment` has been created, we can programmatically add sweeps as shown here.

[ ]:
```
lengths = np.linspace(20e-9, 2020e-9, 31)
exp = QubitExperiment(RabiWidth(q1,lengths),averages=50)
exp.set_fake_data(dig_1, np.exp(-lengths/1e-6)*np.cos(1e7*lengths))
# exp.add_qubit_sweep(q1,"measure", "frequency", np.linspace(6.512e9, 6.522e9, 11))
exp.run_sweeps()
```

```
Compiled 21 sequences.
```

Digitizer Data q1:

Digitizer Data q1:

q1 measure frequency:

We fetch the data and data descriptor directly from the buffer. The data is automatically reshaped to match the experiment axes, and the descriptor enumerates all of the values of these axes for convenience plotting, etc..

[21]:
```
data, descriptor = exp.outputs_by_qubit["q1"][0].get_data()
```

[22]:
```
descriptor.axes
```

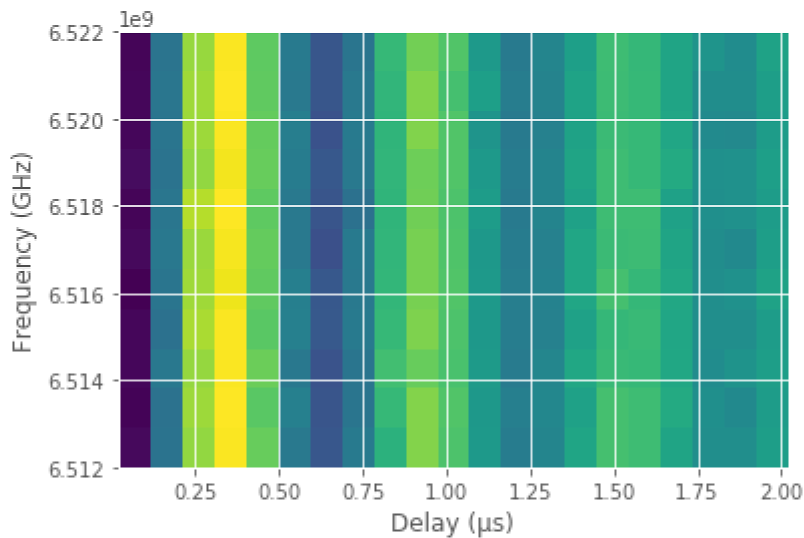
[22]:
```
[<SweepAxis(name=q1 measure frequency,length=11,unit=None,value=6522000000.0,
↪unstructured=False>,
 <DataAxis(name=delay, start=0.02, stop=2.02, num=21, unit=us)>]
```

[23]:
```
data.shape
```

[23]:
```
(11, 21)
```

We even include a convenience `extent` function conforming to the infinitely forgettable matplotlib format.

[24]:
```
import matplotlib.pyplot as plt
%matplotlib inline

plt.imshow(np.real(data), aspect='auto', extent=descriptor.extent())
plt.xlabel("Delay (µs)")
plt.ylabel("Frequency (GHz)");
```



Adding Multiple Sweeps

An arbitrary number of sweeps can be added. For example:

[25]:
```
exp = QubitExperiment(RabiWidth(q1,lengths),averages=50)
exp.add_qubit_sweep(q1,"measure", "frequency", np.linspace(6.512e9, 6.522e9, 5))
exp.add_qubit_sweep(q1,"measure", "amplitude", np.linspace(0.0, 1.0, 21))
```

```
Compiled 21 sequences.
```

If we inspect the internal representation of the "output connector" into which the instrument driver will dump its data, we can see all of the axes it contains.

```
[26]: exp.output_connectors["q1"].descriptor.axes
```

```
[26]: [<SweepAxis(name=q1 measure amplitude,length=21,unit=None,value=0.0,
      ↪unstructured=False>,
       <SweepAxis(name=q1 measure frequency,length=5,unit=None,value=6512000000.0,
      ↪unstructured=False>,
       <DataAxis(name=averages, start=0, stop=49, num=50, unit=None)>,
       <DataAxis(name=delay, start=0.02, stop=2.02, num=21, unit=us)>,
       <DataAxis(name=time, start=0.0, stop=2.046e-06, num=1024, unit=None)>]
```

The `DataAxis` entries are "baked in" using hardware looping, while the `SweepAxis` entries are external software loops facilitated by Auspex.

Example Q6: Calibrations

This example notebook shows how to use the pulse calibration framework.

© Raytheon BBN Technologies 2019

```
[ ]: from QGL import *
     from auspex.qubit import *
```

We use a pre-existing database containing a channel library and pipeline we have established.

```
[3]: cl = ChannelLibrary("my_config")
     pl = PipelineManager()
```

Calibrating Mixers

The APS2 requires mixers to upconvert to qubit and cavity frequencies. We must tune the offset of these mixers and the amplitude factors of the quadrature channels to ensure the best possible results. We repeat the definition of the spectrum analyzer here, assuming that the LO driving this instrument is present in the channel library as `spec_an_LO`.

```
[ ]: spec_an = cl.new_spectrum_analzyer("SpecAn", "ASRL/dev/ttyACM0::INSTR", cl["spec_an_LO
     ↪"])
     cal = MixerCalibration(q2, spec_an, mixer="measure")
     cal.calibrate()
```

If the plot server and client are open, then the data will be plotted along with fits from the calibration procedure. The calibration procedure automatically knows which digitizer and AWG units are needed in the process. The relevant instrument parameters are updated but not commited to the database. Therefore they may be rolled back if undesirable results are found.

Pulse Calibrations

A simple set of calibrations is performed as follows.

```
[ ]: cals = RabiAmpCalibration(q2)
     cal.calibrate()
```

```
[ ]: cal = RamseyCalibration(q2)
     cal.calibrate()
```

Of course this is somewhat repetetive and can be sped up:

```
[ ]: cals = [RabiAmpCalibration, RamseyCalibration, Pi2Calibration, PiCalibration]
     [cal(q2).calibrate() for cal in cals]
```

Automatic Tuneup

While we develop algorithms for fully automated tuneup, some segments of the analysis are (primitively) automated as seen below:

```
[ ]: cal = QubitTuneup(q2, f_start=5.2e9, f_stop=5.8e9, coarse_step=50e6, fine_step=0.5e6,␣
     ↪averages=250, amp=0.1)
     cal.calibrate()
```

Example Q7: Single Shot Fidelity

This example notebook shows how to run single shot fidelity experiments

© Raytheon BBN Technologies 2019

```
[2]: from QGL import *
     from auspex.qubit import *
```

```
Defaulting to temporary directory for AWG sequence file outputs.
```

```
auspex-WARNING: 2019-03-08 10:07:30,785 ----> Could not load channelizer library;␣
↪falling back to python methods.
```

We use a pre-existing database containing a channel library and pipeline we have established.

```
[3]: cl = ChannelLibrary("my_config")
     pl = PipelineManager()
```

Calibrating Mixers

The APS2 requires mixers to upconvert to qubit and cavity frequencies. We must tune the offset of these mixers and the amplitude factors of the quadrature channels to ensure the best possible results. We repeat the definition of the spectrum analyzer here, assuming that the LO driving this instrument is present in the channel library as `spec_an_LO`.

```
[ ]: spec_an = cl.new_spectrum_analzyer("SpecAn", "ASRL/dev/ttyACM0::INSTR", cl["spec_an_LO
     ↪"])
     cal = MixerCalibration(q2, spec_an, mixer="measure")
     cal.calibrate()
```

If the plot server and client are open, then the data will be plotted along with fits from the calibration procedure. The calibration procedure automatically knows which digitizer and AWG units are needed in the process. The relevant instrument parameters are updated but not commited to the database. Therefore they may be rolled back if undesirable results are found.

Example Q8: Realistic Two Qubit Tuneup and Experiments

This example notebook shows how to use APS2/X6 ecosystem to tune up a pair of qubits.

© Raytheon BBN Technologies 2019

```
[ ]: import os
os.environ['AWG_DIR'] = "./AWG"

import matplotlib
import matplotlib.pyplot as plt
import QGL.config
import auspex.config
auspex.config.AWGDir = "/home/qlab/BBN/AWG"
QGL.config.AWGDir = "/home/qlab/BBN/AWG"
auspex.config.KernelDir = "/home/qlab/BBN/Kernels"
QGL.config.KernelDir = "/home/qlab/BBN/Kernels"

%matplotlib inline

from auspex.analysis.qubit_fits import *
from auspex.analysis.helpers import *

from QGL import *
from auspex.qubit import *

#import seaborn as sns
#sns.set_style('whitegrid')
```

Channel Library Setup

```
[ ]: # this will all be system dependent
cl = ChannelLibrary(":memory:")

q1 = cl.new_qubit("q1")
q2 = cl.new_qubit("q2")
ip_addresses = [f"192.168.4.{i}" for i in [21, 22, 23, 24, 25, 26, 28, 29]]
aps2 = cl.new_APS2_rack("Maxwell", ip_addresses, tdm_ip="192.168.4.11")
cl.set_master(aps2.px("TDM"))
dig_1  = cl.new_X6("MyX6", address=0)

dig_1.record_length = 4096

# qubit 1
AM1 = cl.new_source("AutodyneM1", "HolzworthHS9000", "HS9004A-492-1",
                     power=16.0, frequency=6.4762e9, reference="10MHz")

q1src = cl.new_source("q1source", "HolzworthHS9000", "HS9004A-492-2",
                     power=16.0, frequency=4.2e9, reference="10MHz")

cl.set_measure(q1, aps2.tx(4), dig_1.channels[1], gate=False, trig_channel=aps2.tx(6).
↪ch("m3"), generator=AM1)
cl.set_control(q1, aps2.tx(5), generator=q1src)


cl["q1"].measure_chan.autodyne_freq = 11e6
cl["q1"].measure_chan.pulse_params = {"length": 3e-6,
                                      "amp": 1.0,
                                      "sigma": 1.0e-8,
                                      "shape_fun": "tanh"}
```

(continues on next page)

(continued from previous page)

```
cl["q1"].frequency = 67.0e6
cl["q1"].pulse_params = {"length": 100e-9,
                         "pi2Amp": 0.4,
                         "piAmp": 0.8,
                         "shape_fun": "drag",
                         "drag_scaling": 0.0,
                         "sigma": 5.0e-9}

#qubit 2
AM2 = cl.new_source("AutodyneM2", "HolzworthHS9000", "HS9004A-492-3",
                     power=16.0, frequency=6.4762e9, reference="10MHz")

q2src = cl.new_source("q2source", "HolzworthHS9000", "HS9004A-492-4",
                     power=16.0, frequency=4.2e9, reference="10MHz")

cl.set_measure(q2, aps2.tx(7), dig_1.channels[0], gate=False, trig_channel=aps2.tx(6).
↪ch("m3"), generator=AM2)
cl.set_control(q2, aps2.tx(8), generator=q2src)

cl["q2"].measure_chan.autodyne_freq = 11e6
cl["q2"].measure_chan.pulse_params = {"length": 3e-6,
                                      "amp": 1.0,
                                      "sigma": 1.0e-8,
                                      "shape_fun": "tanh"}

cl.commit()
```

[ ]:
```
# initialize all four APS2 to linear regime
for i in range(4,8):
    aps2.tx(i).ch(1).I_channel_amp_factor = 0.5
    aps2.tx(i).ch(1).Q_channel_amp_factor = 0.5
    aps2.tx(i).ch(1).amp_factor = 1
```

[ ]:
```
pl = PipelineManager()
pl.create_default_pipeline(qubits=[q1,q2])

for ql in ['q1','q2']:
    qb = cl[ql]
    pl[ql].clear_pipeline()

    pl[ql].stream_type = "raw"

    pl[ql].create_default_pipeline(buffers=False)

    pl[ql]["Demodulate"].frequency = qb.measure_chan.autodyne_freq

    # only enable this filter when you're running single shot fidelity
    #pl[ql].add(FidelityKernel(save_kernel=True, logistic_regression=True, set_
↪threshold=True, label="Q1_SSF"))

    pl[ql]["Demodulate"]["Integrate"].box_car_start = 3e-7
    pl[ql]["Demodulate"]["Integrate"].box_car_stop = 2.3e-6

    #pl[ql].add(Display(label=ql+" - Raw", plot_dims=0))
    #pl[ql]["Demodulate"].add(Display(label=ql+" - Demod", plot_dims=0))
```

(continues on next page)

(continued from previous page)

```
    pl[ql]["Demodulate"]["Integrate"]["Average"].add(Display(label=ql+" - Final␣
↪Average", plot_dims=0))

    # if you want to see partial averages:
    #pl[ql]["Demodulate"]["Integrate"]["Average"].add(Display(label=ql+" - Partial␣
↪Average", plot_dims=0), connector_out="partial_average")

    #pl[ql]["Demodulate"]["Integrate"]["Average"].add(Display(label=ql+" - Partial␣
↪Average1d", plot_dims=1), connector_out="partial_average")
pl.show_pipeline()
```

Cavity Spectroscopy

```
[ ]: pf = PulsedSpec(q1, specOn=False)
     plot_pulse_files(pf)
```

```
[ ]: exp = QubitExperiment(pf,averages=256)
     #exp.add_qubit_sweep(q2, "measure", "frequency", np.linspace(6.38e9, 6.395e9, 51))
     exp.add_qubit_sweep(q1, "measure", "frequency", np.linspace(6.424e9, 6.432e9, 45))
     #exp.add_qubit_sweep(q1,"measure","amplitude",np.linspace(0.2,0.8,10))
     exp.run_sweeps()
```

```
[ ]: # data, desc = exp.writers[0].get_data()
     # plt.plot(desc.axes[0].points, np.abs(data))
```

```
[ ]: AM1.frequency = 6.42843e9
```

```
[ ]: AM2.frequency = 6.3863e9
```

Qubit Spectroscopy

```
[ ]: qb = q1
```

```
[ ]: qb.frequency = 0.0
     qb.pulse_params['length'] = 5e-6
     qb.pulse_params['shape_fun'] = "tanh"
     pf = PulsedSpec(qb, specOn=True)
     plot_pulse_files(pf)
```

```
[ ]: pf = PulsedSpec(qb, specOn=True)
     exp = QubitExperiment(pf,averages=256)
     exp.add_qubit_sweep(qb, "control", "frequency", np.linspace(5.28e9, 5.34e9, 61))
     #exp.run_sweeps()
```

```
[ ]: # data, desc = exp.writers[0].get_data()
     # plt.plot(desc.axes[0].points, np.abs(data))
```

```
[ ]: q1.frequency = -63e6
     q1.pulse_params['length'] = 200e-9
```

(continues on next page)

(continued from previous page)

```
q1.pulse_params['shape_fun'] = "tanh"
q1.pulse_params['piAmp'] = 0.4
q1.pulse_params['pi2Amp'] = 0.2
pf = PulsedSpec(q1, specOn=True)
#plot_pulse_files(pf)
```

[ ]:
```
fq = 5.26e9 #5.2525e9
q1src.frequency = fq - q1.frequency
q1.phys_chan.I_channel_amp_factor = 0.2
q1.phys_chan.Q_channel_amp_factor = 0.2
```

[ ]:
```
q1src.frequency
```

[ ]:
```
q2.frequency = 81e6
q2.pulse_params['length'] = 200e-9
q2.pulse_params['shape_fun'] = "tanh"
q2.pulse_params['piAmp'] = 0.4
q2.pulse_params['pi2Amp'] = 0.2
pf = PulsedSpec(q2, specOn=True)
#plot_pulse_files(pf)
```

[ ]:
```
fq2 = 5.2113e9
q2src.frequency = fq2 - q2.frequency
q2.phys_chan.I_channel_amp_factor = 0.2
q2.phys_chan.Q_channel_amp_factor = 0.2
```

[ ]:
```
q2src.frequency
```

Mixer calibration

[ ]:
```
salo = cl.new_source("salo", "HolzworthHS9000", "HS9004A-381-4",
                     power=10.0, frequency=6.5e9, reference="10MHz")
specAn = cl.new_spectrum_analzyer('specAn','ASRL/dev/ttyACM0',salo)
```

[ ]:
```
from auspex.instruments import enumerate_visa_instruments, SpectrumAnalyzer
```

[ ]:
```
enumerate_visa_instruments()
```

[ ]:
```
# from here out, uncomment cal.calibrate() if you want to actually run the cal
cal = MixerCalibration(q1,specAn,'control', phase_range = (-0.5, 0.5), amp_range=(0.8,
↪ 1.2))
#cal.calibrate()
```

[ ]:
```
# listed here only if manual adjustment is needed

q1.phys_chan.I_channel_offset = -0.0004
q1.phys_chan.Q_channel_offset = -0.019
q1.phys_chan.amp_factor = 1.004
q1.phys_chan.phase_skew = 0.053
```

```
[ ]: cal = MixerCalibration(q2,specAn,'control', phase_range = (-0.5, 0.5), amp_range=(0.8,
↪ 1.2))
#cal.calibrate()
```

```
[ ]: q2.phys_chan.I_channel_offset = -0.0004
q2.phys_chan.Q_channel_offset = -0.019
q2.phys_chan.amp_factor = 0.985
q2.phys_chan.phase_skew = 0.074
```

```
[ ]: cal = MixerCalibration(q1,specAn,'measure', phase_range = (-0.5, 0.5), amp_range=(0.8,
↪ 1.2))
#cal.calibrate()
```

```
[ ]: cal = MixerCalibration(q2,specAn,'measure', phase_range = (-0.5, 0.5), amp_range=(0.8,
↪ 1.2))
#cal.calibrate()
```

Rabi Width

```
[ ]: pf = RabiWidth(q1, np.arange(20e-9, 0.602e-6, 10e-9))
exp = QubitExperiment(pf, averages=200)
plot_pulse_files(pf)
#exp.add_qubit_sweep(q1, "control", "frequency", q1src.frequency + np.linspace(-6e6,␣
↪6e6, 61))
exp.run_sweeps()
```

```
[ ]: pf = RabiWidth(q2, np.arange(20e-9, 0.602e-6, 10e-9))
exp = QubitExperiment(pf, averages=200)
plot_pulse_files(pf)
#exp.add_qubit_sweep(q2, "control", "frequency", q2src.frequency + np.linspace(-2e6,␣
↪2e6, 21))
#exp.add_qubit_sweep(q2, "measure", "frequency", AM2.frequency + np.linspace(-2e6,␣
↪2e6, 11))
exp.run_sweeps()
```

Rabi Amp

```
[ ]: pf = RabiAmp(q1, np.linspace(-1, 1, 101))
exp = QubitExperiment(pf, averages=128)
exp.run_sweeps()
```

```
[ ]: cal = RabiAmpCalibration(q1,quad='imag')
#cal.calibrate()
```

```
[ ]: q1.pulse_params['piAmp'] = 0.6179
q1.pulse_params['pi2Amp'] = q1.pulse_params['piAmp']/2
```

```
[ ]: pf = RabiAmp(q2, np.linspace(-1, 1, 101))
exp = QubitExperiment(pf, averages=128)
exp.run_sweeps()
```

```
[ ]: cal = RabiAmpCalibration(q2,quad='imag')
     #cal.calibrate()
```

```
[ ]: q2.pulse_params['piAmp'] = 0.743
     q2.pulse_params['pi2Amp'] = q2.pulse_params['piAmp']/2
```

Ramsey Calibration

```
[ ]: # need to be in the neighbourhood for this to work

     cal = RamseyCalibration(q1,quad='imag')
     #cal.calibrate()
```

T1

```
[ ]: qb = q2
     icpts = np.linspace(20e-9, 201.02e-6, 101)
     pf = InversionRecovery(qb, icpts)
     exp = QubitExperiment(pf, averages=400)
     exp.run_sweeps()
```

```
[ ]: data, desc = exp.writers[0].get_data()
```

```
[ ]: from auspex.analysis.qubit_fits import T1Fit, RamseyFit
     from auspex.analysis.helpers import cal_scale
```

```
[ ]: sdata = cal_scale(data)
```

```
[ ]: fit = T1Fit(icpts, abs(data[0:-4]))
```

```
[ ]: fit.make_plots()
```

T2

```
[ ]: rpts = np.linspace(20e-9, 50.02e-6, 101)
     pf = Ramsey(q1, rpts ,TPPIFreq=0e3)
     #exp.add_qubit_sweep(q1, "control", "frequency", q1src.frequency + np.linspace(-2e6,␣
     ↪2e6, 21))
     exp = QubitExperiment(pf, averages=200)
     exp.run_sweeps()
```

```
[ ]: data, desc = exp.writers[0].get_data()
     sdata = cal_scale(data)
```

```
[ ]: fit = RamseyFit(rpts, abs(data[0:-4]),make_plots = True)
```

```
[ ]: fcorrect = fit.fit_params['f']
```

```
[ ]: fcorrect
```

```
[ ]: #q1src.frequency -= fcorrect
```

```
[ ]: pf = Ramsey(q2, rpts,TPPIFreq=0e3)
     exp = QubitExperiment(pf, averages=200)
     exp.run_sweeps()
```

```
[ ]: data, desc = exp.writers[0].get_data()
     sdata = cal_scale(data)
```

```
[ ]: fit = RamseyFit(rpts, abs(data[0:-4]),make_plots = True)
     fcorrect = fit.fit_params['f']
```

```
[ ]: fcorrect*1e-6
```

```
[ ]: #q2src.frequency += fcorrect
```

Echo experiments

```
[ ]: exp = QubitExperiment(HahnEcho(q2, np.linspace(20e-9, 80.02e-6, 81), periods=5),␣
     ↪averages=512)
     exp.run_sweeps()
```

```
[ ]: data, desc = exp.writers[0].get_data()
     cdata = cal_scale(np.real(data))
     fit = RamseyFit(desc.axes[0].points[:-4], cdata, make_plots=True)
     fit.fit_params
```

Single Qubit Cals

```
[ ]: RabiAmpCalibration(q1, quad="imag").calibrate()
```

```
[ ]: PiCalibration(q1, quad="imag", num_pulses=7).calibrate()
```

```
[ ]: Pi2Calibration(q1, quad="imag", num_pulses=7).calibrate()
```

Single-Qubit RB

```
[ ]: from auspex.analysis.qubit_fits import *
     from auspex.analysis.helpers import *
```

```
[ ]: rb_lens = [2, 4, 8, 16, 32, 128, 256, 512]
```

```
[ ]: rb_seqs = create_RB_seqs(1, rb_lens)
     pf = SingleQubitRB(q1, rb_seqs)
```

```
[ ]: exp = QubitExperiment(pf, averages=256)
     exp.run_sweeps()
```

```
[ ]: data, desc = exp.writers[0].get_data()
```

```
[ ]: lens = [int(l) for l in desc.axes[0].points[:-4]]
```

```
[ ]: SingleQubitRBFit(lens, cal_scale(np.imag(data)), make_plots=True)
```

Fancier things

Maybe you want to see how T_1 varies with repeated measurement…

```
[ ]: from auspex.parameter import IntParameter
     import time
```

```
[ ]: N = 1000
     lengths = np.linspace(20e-9, 500.02e-6, 101)
```

```
[ ]: T1seq = [[X(q2), Id(q2, length=d), MEAS(q2)] for d in lengths]
     T1seq += create_cal_seqs((q2,), 2)
```

```
[ ]: wait_param = IntParameter(default=0)
     wait_param.name = "Repeat"
     #wait = lambda x: print(f"{x}")
     def wait(x):
         print(f"{x}")
         time.sleep(2)
     wait_param.assign_method(wait)


     mf = compile_to_hardware(T1seq, "T1/T1")
     exp = QubitExperiment(mf, averages=512)
     exp.wait_param = wait_param
     # with these params each shot is roughly 22 secs apart

     exp.add_sweep(exp.wait_param, list(range(N))) # repeat T1 scan 1000 times
     exp.run_sweeps()
```

```
[ ]: # load data
     # get T1s
     T1s = []
     T1_error = []
     #data, desc = exp.writers[0].get_data()
     for i in range(N):
         cdata = cal_scale(np.imag(data[i,:]))
         fit = T1Fit(lengths, cdata, make_plots=False)
         T1s.append(fit.fit_params["T1"])
         T1_error.append(fit.fit_errors["T1"])
```

```
[ ]: plt.figure(figsize=(8,6))
     #plt.errorbar(range(1000),np.array(T1s)*1e6, yerr=np.array(T1_error)*1e6, fmt='+',␣
     ↪elinewidth=0.5, barsabove=True, capsize=0.7)
     plt.plot(range(1000),np.array(T1s)*1e6, '+')
     plt.title(r'Q2 $T_1$ Variability')
     plt.xlabel('N repeats')
     plt.ylabel(r'$T_1$ (us)')
     #plt.savefig('T1vsTime.png', dpi=300, bbox_inches='tight')
```

2Q RB

```
[ ]: lengths = [2,4,6,8,10] # range(2,10) #[2**n for n in range(1,6)]
     seqs = create_RB_seqs(2, lengths=lengths)
     exp = qef.create(TwoQubitRB(q1, q2, seqs=seqs), expname='Q2Q1RB')
     exp.run_sweeps()
```

2Q Gates

```
[ ]: edge12 = cl.new_edge(q1,q2)
     cl.set_control(edge12, aps2.tx(5), generator=q1src)
```

```
[ ]: q1_10 = q1.phys_chan.generator.frequency + q1.frequency
     q2_10 = q2.phys_chan.generator.frequency + q2.frequency
```

```
[ ]: edge12.frequency = q2_10 - q1.phys_chan.generator.frequency
```

```
[ ]: edge12.pulse_params = {'length': 400e-9, 'amp': 0.8, 'shape_fun': 'tanh', 'sigma':10e-
     ↪9}
```

```
[ ]: q1.measure_chan.pulse_params['amp'] = 1.0
     q2.measure_chan.pulse_params['amp'] = 1.0
```

CR length cal

```
[ ]: crlens = np.arange(100e-9,2.1e-6,100e-9)
     pf = EchoCRLen(q1,q2,lengths=crlens)
     plot_pulse_files(pf)
```

```
[ ]: exp = QubitExperiment(pf, averages=512)
     exp.run_sweeps()
```

Above just runs the experiment used by the calibration routine. Here is the actual calibration:

```
[ ]: crlens = np.arange(100e-9,2.1e-6,100e-9)
     # phase, amp and rise_fall have defaults but you can overwrite them
     pce = CRLenCalibration(cl["q1->q2"], lengths=lengths, phase = 0, amp = 0.8, rise_fall␣
     ↪= 40e-9,
                 do_plotting=True, sample_name="CRLen", averages=512)
     pec.calibrate()
```

CR phase cal

```
[ ]: phases = np.arange(0,np.pi*2,np.pi/20)
     pf = EchoCRPhase(q1,q2,phases,length=1000e-9)
     plot_pulse_files(pf)
```

```
[ ]: exp = QubitExperiment(pf, averages=512)
     exp.run_sweeps()
```

```
[ ]: phases = np.linspace(0, 2*np.pi, 21)
     pce = CRPhaseCalibration(edge12, phases = phases, amp = 0.8, rise_fall = 40e-9,
                 do_plotting=True, sample_name="CRPhase", averages=512)
     pec.calibrate()
```

CR amp cal

```
[ ]: amps = np.arange(0.7,0.9,0.1)
     pf = EchoCRAmp(q1,q2,amps,length=1000e-9)
     plot_pulse_files(pf)
```

```
[ ]: exp = QubitExperiment(pf, averages=512)
     exp.run_sweeps()
```

```
[ ]: pce = CRAmpCalibration(cl["q1->q2"], amp_range = 0.2, rise_fall = 40e-9,
                 do_plotting=True, sample_name="CRAmp", averages=512)
     pec.calibrate()
```

3.4.3 Instrument Drivers

For libaps2, libalazar, and libx6, one should be able to *conda install -c bbn-q xxx* in order to obtain binary distributions of the relevant packages. Otherwise, one must obtain and build those libraries (according to their respective documentation), then make the shared library build products and any python packages available to Auspex by placing them on the path.

3.5 Plot Server

Auspex plotting is facilitated by a plot server and plot clients. A single server can handle multiple running experiments, which publish their data with a unique UUID. Many clients can connect to the server and request data for a particular UUID.

3.5.1 Running the Plot Server

The plot server must currently be started manually with:

```
python plot_server.py
```

3.5.2 Running the Plot Client

The plot client *matplotlib-client.py* should be run automatically whenever plotters are put in an experiment's filter pipeline. The code can be run manually and used to connect to a remote system, simply by running the exectuable with:

```
python matplotlib-client.py
```

3.6 auspex package

3.6.1 Subpackages

auspex.analysis package

Submodules

auspex.analysis.fits module

class auspex.analysis.fits.**Auspex2DFit**(*xpts*, *ypts*, *zpts*, *make_plots=False*)

A generic fit class wrapping scipy.optimize.curve_fit for convenience. Specific fits should inherit this class.

xlabel
Plot x-axis label.

Type str

ylabel
Plot y-axis label.

Type str

title
Plot title.

Type str

annotation()
Annotation for the *make_plot()* method. Should return a string that is passed to *matplotlib.pyplot.annotate*.

make_plots()
Create a plot of the input data and the fitted model. By default will include any annotation defined in the *annotation()* class method.

model(*x=None*)
The fit function evaluated at the parameters found by *curve_fit*.

Parameters **x** – A number or *numpy.array* returned by the model function.

title = 'Auspex Fit'

xlabel = 'X points'

ylabel = 'Y points'

class auspex.analysis.fits.**AuspexFit**(*xpts*, *ypts*, *make_plots=False*, *ax=None*)

A generic fit class wrapping scipy.optimize.curve_fit for convenience. Specific fits should inherit this class.

xlabel
Plot x-axis label.

Type str

ylabel
Plot y-axis label.

Type str

title
Plot title.

Type str

annotation()
Annotation for the *make_plot()* method. Should return a string that is passed to *matplotlib.pyplot.annotate*.

ax = None

bounds = None

make_plots()
Create a plot of the input data and the fitted model. By default will include any annotation defined in the *annotation()* class method.

model(*x=None*)
The fit function evaluated at the parameters found by *curve_fit*.

Parameters **x** – A number or *numpy.array* returned by the model function.

title = 'Auspex Fit'

xlabel = 'X points'

ylabel = 'Y points'

class auspex.analysis.fits.**GaussianFit**(*xpts*, *ypts*, *make_plots=False*, *ax=None*)
A fit to a gaussian function

title = 'Gaussian Fit'

xlabel = 'X Data'

ylabel = 'Y Data'

class auspex.analysis.fits.**LorentzFit**(*xpts*, *ypts*, *make_plots=False*, *ax=None*)
A fit to a simple Lorentzian function *A /((x-b)^2 + (c/2)^2) + d*

title = 'Lorentzian Fit'

xlabel = 'X Data'

ylabel = 'Y Data'

class auspex.analysis.fits.**MultiGaussianFit**(*x*, *y*, *make_plots=False*, *n_gaussians=2*, *n_samples=100000*)
A fit to a sum of gaussian function. Use with care!

title = 'Sum of Gaussians Fit'

xlabel = 'X Data'

ylabel = 'Y Data'

class auspex.analysis.fits.**QuadraticFit**(*xpts*, *ypts*, *make_plots=False*, *ax=None*)
A fit to a simple quadratic function A*(x-x0)**2 + b

title = 'Quadratic Fit'

xlabel = 'X Data'

ylabel = 'Y Data'

auspex.analysis.helpers module

auspex.analysis.helpers.**cal_data**(*data*, *quad=<function real>*, *qubit_name='q1'*, *group_name='main'*, *return_type=<class 'numpy.float32'>*, *key="*)

Rescale data to σ_z. expectation value based on calibration sequences.

Parameters

- **data** (*numpy array*) – The data from the writer or buffer, which is a dictionary whose keys are typically in the format qubit_name-group_name, e.g. ({'q1-main'} : array([(0.0+0.0j, . . .), (. . .), . . .]))
- **quad** (*numpy function*) – This should be the quadrature where most of the data can be found. Options are: np.real, np.imag, np.abs and np.angle
- **qubit_name** (*string*) – Name of the qubit in the data file. Default is 'q1'
- **group_name** (*string*) – Name of the data group to calibrate. Default is 'main'
- **return_type** (*numpy data type*) – Type of the returned data. Default is np.float32.
- **key** (*string*) – In the case where the dictionary keys don't conform to the default format a string can be passed in specifying the data key to scale.

Returns

numpy array (type return_type) Returns the data rescaled to match the calibration results for the σ_z expectation value.

Examples

Loading and calibrating data

```
>>> exp = QubitExperiment(T1(q1),averages=500)
>>> exp.run_sweeps()
>>> data, desc = exp.writers[0].get_data()
```

auspex.analysis.helpers.**cal_ls**()

List of auspex.pulse_calibration results

auspex.analysis.helpers.**cal_scale**(*data*, *bit=0*, *nqubits=1*, *repeats=2*)

Scale data from calibration points. :param data: :type data: Unscaled data with cal points. :param bit: :type bit: Which qubit in the sequence is to be calibrated (0 for 1st, etc. . .). Default 0. :param nqubits: :type nqubits: Number of qubits in the data. Default 1. :param repeats: :type repeats: Number of calibraiton repeats. Default 2.

Returns data

Return type scaled data array

auspex.analysis.helpers.**get_cals**(*qubit*, *params*, *make_plots=True*, *depth=0*)

Return and optionally plot the result of the most recent calibrations/characterizations :param qubit: :type qubit: qubit name (string) :param params: :type params: parameter(s) to plot (string or list of strings) :param

make_plots: :type make_plots: optional bool (default = True) :param depth: :type depth: optional integer, number of most recent calibrations to load (default = 0 for all cals) :param ———–: :param Returns: :param List of: :type List of: dates, values, errors

`auspex.analysis.helpers.`**`get_file_name`**`()`

Helper function to get a filepath from a dialog box

`auspex.analysis.helpers.`**`load_data`**`(`*dirpath=None*`)`

Open data in the .auspex file at dirpath/

Parameters **`dirpath`** (*`string`*) – Path to the .auspex file. If no folder is specified, a dialogue box will ask for a path.

Returns

data_set (Dict{data group}{data name}{data,descriptor}) Data as a dictionary structure with data groups, types of data and the data packed sequentially

Examples

Loading a data container

```
>>> data = load_data('/path/to/my/data.auspex')
>>> data
{'q2-main': {'data': {'data': array([[ 0.16928101-0.05661011j,  0.3225708 +0.
↪08914185j,
    0.2114563 +0.10314941j, ..., -0.32357788+0.16964722j,
>>> data["q3-main"]["variance"]["descriptor"]
<DataStreamDescriptor(num_dims=1, num_points=52)>
>>> data["q3-main"]["variance"]["data"]
array([0.00094496+0.00014886j, 0.00089747+0.00015082j,
0.00090634+0.00015106j, 0.00090128+0.00014451j,...
```

`auspex.analysis.helpers.`**`normalize_buffer_data`**`(`*data*, *desc*, *qubit_index*, *zero_id=0*, *one_id=1*`)`

`auspex.analysis.helpers.`**`normalize_data`**`(`*data*, *zero_id=0*, *one_id=1*`)`

`auspex.analysis.helpers.`**`open_data`**`(`*num=None*, *folder=None*, *groupname='main'*, *datasetname='data'*, *date=None*`)`

Convenience Load data from an *AuspexDataContainer* given a file number and folder. Assumes that files are named with the convention *ExperimentName-NNNNN.auspex*

Parameters

- **`num`** (*`int`*) – File number to be loaded.
- **`folder`** (*`string`*) – Base folder where file is stored. If the *date* parameter is not None, assumes file is a dated folder. If no folder is specified, open a dialogue box. Open the folder with the desired ExperimentName-NNNN.auspex, then press OK
- **`groupname`** (*`string`*) – Group name of data to be loaded.
- **`datasetname`** (*`string, optional`*) – Data set name to be loaded. Default is "data".
- **`date`** (*`string, optional`*) – Date folder from which data is to be loaded. Format is "YYMMDD" Defaults to today's date.

Returns

data (numpy.array) Data loaded from file.

desc (DataSetDescriptor) Dataset descriptor loaded from file.

Examples

Loading a data container

```
>>> data, desc = open_data(42, '/path/to/my/data', "q1-main", date="190301")
```

Module contents

auspex.filters package

Submodules

auspex.filters.average module

auspex.filters.channelizer module

class auspex.filters.channelizer.**Channelizer**(*frequency=None*, *bandwidth=None*, *decimation_factor=None*, *follow_axis=None*, *follow_freq_offset=None*, ***kwargs*)

Digital demodulation and filtering to select a particular frequency multiplexed channel. If an axis name is supplied to *follow_axis* then the filter will demodulate at the freqency *axis_frequency_value - follow_freq_offset* otherwise it will demodulate at *frequency*. Note that the filter coefficients are still calculated with respect to the *frequency* paramter, so it should be chosen accordingly when *follow_axis* is defined.

bandwidth = <FloatParameter(name='bandwidth',value=5000000.0)>

decimation_factor = <IntParameter(name='decimation_factor',value=4)>

final_init()

follow_axis = <Parameter(name='follow_axis',value='')>

follow_freq_offset = <FloatParameter(name='follow_freq_offset',value=0.0)>

frequency = <FloatParameter(name='frequency',value=10000000.0)>

init_filters(*frequency*, *bandwidth*)

process_data(*data*)

sink = <InputConnector(name=)>

source = <OutputConnector(name=)>

update_descriptors()

This method is called whenever the connectivity of the graph changes. This may have implications for the internal functioning of the filter, in which case update_descriptors should be overloaded. Any simple changes to the axes within the StreamDescriptors should take place via the class method descriptor_map.

update_references(*frequency*)

auspex.filters.correlator module

class auspex.filters.correlator.**Correlator**(*filter_name=None*, ***kwargs*)

filter_name = 'Correlator'

operation()
Must be overridden with the desired mathematical function

sink = <InputConnector(name=)>

source = <OutputConnector(name=)>

unit(*base_unit*)
Must be overridden accoriding the desired mathematical function e.g. return base_unit + "^{}".format(len(self.sink.input_streams))

auspex.filters.debug module

class auspex.filters.debug.**Print**(**args*, ***kwargs*)
Debug printer that prints data comming through filter

process_data(*data*)

sink = <InputConnector(name=)>

class auspex.filters.debug.**Passthrough**(**args*, ***kwargs*)

process_data(*data*)

sink = <InputConnector(name=)>

source = <OutputConnector(name=)>

auspex.filters.elementwise module

class auspex.filters.elementwise.**ElementwiseFilter**(*filter_name=None*, ***kwargs*)
Perform elementwise operations on multiple streams: e.g. multiply or add all streams element-by-element

filter_name = 'GenericElementwise'

main()
Generic run method which waits on a single stream and calls *process_data* on any new_data

operation()
Must be overridden with the desired mathematical function

sink = <InputConnector(name=)>

source = <OutputConnector(name=)>

unit(*base_unit*)
Must be overridden accoriding the desired mathematical function e.g. return base_unit + "^{}".format(len(self.sink.input_streams))

update_descriptors()
Must be overridden depending on the desired mathematical function

auspex.filters.filter module

class auspex.filters.filter.**Filter**(*name=None*, ***kwargs*)
Any node on the graph that takes input streams with optional output streams

checkin()
For any filter-specific loop needs

configure_with_proxy(*proxy_obj*)
For use with bbndb, sets this filter's properties using a FilterProxy object taken from the filter database.

descriptor_map(*input_descriptors*)
Return a dict of the output descriptors.

execute_on_run()

main()
Generic run method which waits on a single stream and calls *process_data* on any new_data

on_done()
To be run when the done signal is received, in case additional steps are needed (such as flushing a plot or data).

process_message(*msg*)
To be overridden for interesting default behavior

push_resource_usage()

push_to_all(*message*)

run()
Method to be run in sub-process; can be overridden in sub-class

shutdown()

update_descriptors()
This method is called whenever the connectivity of the graph changes. This may have implications for the internal functioning of the filter, in which case update_descriptors should be overloaded. Any simple changes to the axes within the StreamDescriptors should take place via the class method descriptor_map.

auspex.filters.framer module

class auspex.filters.framer.**Framer**(*axis=None*, ***kwargs*)
Mete out data in increments defined by the specified axis.

axis = <Parameter(name='axis',value=None)>

final_init()

process_data(*data*)

sink = <InputConnector(name=)>

source = <OutputConnector(name=)>

auspex.filters.integrator module

class auspex.filters.integrator.**KernelIntegrator**(***kwargs*)

bias = <FloatParameter(name='bias',value=0.0)>

box_car_start = <FloatParameter(name='box_car_start',value=0.0)>

box_car_stop = <FloatParameter(name='box_car_stop',value=1e-07)>

demod_frequency = <FloatParameter(name='demod_frequency',value=0.0)>
Integrate with a given kernel. Kernel will be padded/truncated to match record length

kernel = <Parameter(name='kernel',value=None)>

process_data(*data*)

simple_kernel = <BoolParameter(name='simple_kernel',value=True)>

sink = <InputConnector(name=)>

source = <OutputConnector(name=)>

update_descriptors()
This method is called whenever the connectivity of the graph changes. This may have implications for the internal functioning of the filter, in which case update_descriptors should be overloaded. Any simple changes to the axes within the StreamDescriptors should take place via the class method descriptor_map.

auspex.filters.io module

class auspex.filters.io.**WriteToFile**(*filename=None*, *groupname=None*, *datasetname=None*, ***kwargs*)
Writes data to file using the Auspex container type, which is a simple directory structure with subdirectories, binary datafiles, and json meta files that store the axis descriptors and other information.

datasetname = <Parameter(name='datasetname',value='data')>

filename = <Parameter(name='filename',value=None)>

final_init()

get_data()

get_data_while_running(*return_queue*)
Return data to the main thread or user as requested. Use a MP queue to transmit.

groupname = <Parameter(name='groupname',value='main')>

process_data(*data*)

sink = <InputConnector(name=)>

class auspex.filters.io.**DataBuffer**(***kwargs*)
Writes data to IO.

checkin()
For any filter-specific loop needs

final_init()

get_data()

main()
Generic run method which waits on a single stream and calls *process_data* on any new_data

process_data(*data*)

sink = <InputConnector(name=)>

auspex.filters.plot module

class auspex.filters.plot.**Plotter**(**args*, *name=''*, *plot_dims=None*, *plot_mode=None*, ***plot_args*)

axis_label(*index*)

desc()

execute_on_run()

final_init()

get_final_plot(*quad_funcs=[<ufunc 'absolute'>, <function angle>]*)

on_done()
To be run when the done signal is received, in case additional steps are needed (such as flushing a plot or data).

plot_dims = <IntParameter(name='plot_dims',value=0)>

plot_mode = <Parameter(name='plot_mode',value='quad')>

process_data(*data*)

send(*message*)

set_done()

set_quit()

sink = <InputConnector(name=)>

update()

update_descriptors()
This method is called whenever the connectivity of the graph changes. This may have implications for the internal functioning of the filter, in which case update_descriptors should be overloaded. Any simple changes to the axes within the StreamDescriptors should take place via the class method descriptor_map.

class auspex.filters.plot.**ManualPlotter**(*name=''*, *x_label=['X']*, *y_label=['y']*, *y_lim=None*, *numplots=1*)
Establish a figure, then give the user complete control over plot creation and data. There isn't any reason to run this as a process, but we provide the same interface for convenience.

add_data_trace(*name*, *custom_mpl_kwargs={}*, *subplot_num=0*)

add_fit_trace(*name*, *custom_mpl_kwargs={}*, *subplot_num=0*)

add_trace(*name*, *matplotlib_kwargs={}*, *subplot_num=0*)

desc()

execute_on_run()

send(*message*)

set_data(*trace_name*, *xdata*, *ydata*)

set_done()

set_quit()

start()

stop()

class auspex.filters.plot.**MeshPlotter**(**args*, *name=''*, *plot_mode=None*, *x_label=''*, *y_label=''*, ***plot_args*)

desc()

execute_on_run()

on_done()
To be run when the done signal is received, in case additional steps are needed (such as flushing a plot or data).

plot_mode = <Parameter(name='plot_mode',value='quad')>

process_direct(*data*)

send(*message*)

sink = <InputConnector(name=)>

update_descriptors()
This method is called whenever the connectivity of the graph changes. This may have implications for the internal functioning of the filter, in which case update_descriptors should be overloaded. Any simple changes to the axes within the StreamDescriptors should take place via the class method descriptor_map.

SingleShotMeasurement auspex.filters.singleshot module ——————————-

class auspex.filters.singleshot.**SingleShotMeasurement**(*save_kernel=False*, *optimal_integration_time=False*, *zero_mean=False*, *set_threshold=False*, *logistic_regression=False*, ***kwargs*)

TOLERANCE = 0.001

compute_filter()
Compute the single shot kernel and obtain single-shot measurement fidelity.

Expects that the data will be in self.ground_data and self.excited_data, which are (T, N)-shaped numpy arrays, with T the time axis and N the number of shots.

final_init()

logistic_fidelity()

logistic_regression = <BoolParameter(name='logistic_regression',value=False)>

optimal_integration_time = <BoolParameter(name='optimal_integration_time',value=False)>

process_data(*data*)
Fill the ground and excited data bins

save_kernel = <BoolParameter(name='save_kernel',value=False)>

set_threshold = <BoolParameter(name='set_threshold',value=False)>

sink = <InputConnector(name=)>

source = <OutputConnector(name=)>

update_descriptors()
This method is called whenever the connectivity of the graph changes. This may have implications for the internal functioning of the filter, in which case update_descriptors should be overloaded. Any simple changes to the axes within the StreamDescriptors should take place via the class method descriptor_map.

zero_mean = <BoolParameter(name='zero_mean',value=False)>

auspex.filters.stream_selectors module

Module contents

auspex.instruments package

Submodules

auspex.instruments.agilent module

class auspex.instruments.agilent.**Agilent33220A**(*resource_name=None*, **args*, ***kwargs*)

Agilent 33220A Function Generator

FUNCTION_MAP = {'DC': 'DC', 'Noise': 'NOIS', 'Pulse': 'PULS', 'Ramp': 'RAMP', 'Sine':

amplitude

auto_range

burst_cycles

burst_mode

burst_state

connect(*resource_name=None*, *interface_type=None*)

Either connect to the resource name specified during initialization, or specify a new resource name here.

dc_offset

frequency

function

get_amplitude(***kwargs*)

get_auto_range(***kwargs*)

get_burst_cycles(***kwargs*)

get_burst_mode(***kwargs*)

get_burst_state(***kwargs*)

get_dc_offset(***kwargs*)

get_frequency(***kwargs*)

get_function(***kwargs*)

get_high_voltage(***kwargs*)

get_load_resistance(***kwargs*)

get_low_voltage(***kwargs*)

get_output(***kwargs*)

get_output_sync(***kwargs*)

get_output_units(***kwargs*)

get_polarity(***kwargs*)

get_pulse_dcyc(***kwargs*)

get_pulse_edge(***kwargs*)

get_pulse_period(***kwargs*)

get_pulse_width(***kwargs*)

get_ramp_symmetry(***kwargs*)

get_trigger_slope(***kwargs*)

get_trigger_source(***kwargs*)

high_voltage

load_resistance

low_voltage

output

output_sync

output_units

polarity

pulse_dcyc

pulse_edge

pulse_period

pulse_width

ramp_symmetry

set_amplitude(*val*, ***kwargs*)

set_auto_range(*val*, ***kwargs*)

set_burst_cycles(*val*, ***kwargs*)

set_burst_mode(*val*, ***kwargs*)

set_burst_state(*val*, ***kwargs*)

set_dc_offset(*val*, ***kwargs*)

set_frequency(*val*, ***kwargs*)

set_function(*val*, ***kwargs*)

set_high_voltage(*val*, ***kwargs*)

set_load_resistance(*val*, ***kwargs*)

set_low_voltage(*val*, ***kwargs*)

set_output(*val*, ***kwargs*)

set_output_sync(*val*, ***kwargs*)

set_output_units(*val*, ***kwargs*)

set_polarity(*val*, ***kwargs*)

set_pulse_dcyc(*val*, ***kwargs*)

set_pulse_edge(*val*, ***kwargs*)

set_pulse_period(*val*, ***kwargs*)

set_pulse_width(*val*, ***kwargs*)

set_ramp_symmetry(*val*, ***kwargs*)

set_trigger_slope(*val*, ***kwargs*)

set_trigger_source(*val*, ***kwargs*)

trigger()

trigger_slope

trigger_source

class auspex.instruments.agilent.**Agilent33500B**(*resource_name=None*, **args*, ***kwargs*)

Agilent/Keysight 33500 series 2-channel Arbitrary Waveform Generator

Replacement model for 33220 series with some changes and additional sequencing functionality

FUNCTION_MAP = {'DC': 'DC', 'Noise': 'NOIS', 'PRBS': 'PRBS', 'Pulse': 'PULS', 'Ramp':

class Segment(*name*, *data=[]*, *dac=True*, *control='once'*, *repeat=0*, *mkr_mode='maintain'*, *mkr_pts=4*)

self_check()

update(***kwargs*)

class Sequence(*name*)

add_segment(*segment*, ***kwargs*)

Create a copy of the segment, update its values, then add to the sequence. The copy and update are to allow reuse of the same segment with different configurations. For safety, avoid reuse, but add different segment objects to the sequence.

get_descriptor()

Return block descriptor to upload to the instrument

abort()

amplitude = <auspex.instruments.instrument.FloatCommand object>

arb_advance = <auspex.instruments.instrument.StringCommand object>

arb_amplitude = <auspex.instruments.instrument.FloatCommand object>

arb_frequency = <auspex.instruments.instrument.FloatCommand object>

arb_sample = <auspex.instruments.instrument.FloatCommand object>

arb_sync()

Restart the sequences and synchronize them

arb_waveform = <auspex.instruments.instrument.StringCommand object>

auto_range = <auspex.instruments.instrument.Command object>

burst_cycles = <auspex.instruments.instrument.FloatCommand object>

burst_mode = <auspex.instruments.instrument.StringCommand object>

burst_state = <auspex.instruments.instrument.Command object>

clear_waveform(*channel=1*)
Clear all waveforms loaded in the memory

connect(*resource_name=None*, *interface_type=None*)
Either connect to the resource name specified during initialization, or specify a new resource name here.

dc_offset = <auspex.instruments.instrument.FloatCommand object>

frequency = <auspex.instruments.instrument.FloatCommand object>

function = <auspex.instruments.instrument.StringCommand object>

get_amplitude(***kwargs*)

get_arb_advance(***kwargs*)
Advance mode to the next point: 'Trigger' or 'Srate' (Sample Rate)

get_arb_amplitude(***kwargs*)

get_arb_frequency(***kwargs*)

get_arb_sample(***kwargs*)
Sample Rate

get_arb_waveform(***kwargs*)

get_auto_range(***kwargs*)

get_burst_cycles(***kwargs*)

get_burst_mode(***kwargs*)

get_burst_state(***kwargs*)

get_dc_offset(***kwargs*)

get_frequency(***kwargs*)

get_function(***kwargs*)

get_high_voltage(***kwargs*)

get_load(***kwargs*)
Expected load resistance, 1-10k

get_low_voltage(***kwargs*)

get_output(***kwargs*)

get_output_gated(***kwargs*)

get_output_sync(***kwargs*)

get_output_trigger(***kwargs*)

get_output_trigger_slope(***kwargs*)

get_output_trigger_source(***kwargs*)

get_output_units(***kwargs*)

get_polarity(***kwargs*)

get_pulse_dcyc(***kwargs*)

get_pulse_edge(***kwargs*)

get_pulse_period(***kwargs*)

get_pulse_width(***kwargs*)

get_ramp_symmetry (***kwargs*)

get_sequence (***kwargs*)

get_sync_mode (***kwargs*)

get_sync_polarity (***kwargs*)

get_sync_source (***kwargs*)

get_trigger_slope (***kwargs*)

get_trigger_source (***kwargs*)

high_voltage = <auspex.instruments.instrument.FloatCommand object>

load = <auspex.instruments.instrument.FloatCommand object>

low_voltage = <auspex.instruments.instrument.FloatCommand object>

output = <auspex.instruments.instrument.Command object>

output_gated = <auspex.instruments.instrument.Command object>

output_sync = <auspex.instruments.instrument.Command object>

output_trigger

output_trigger_slope

output_trigger_source

output_units = <auspex.instruments.instrument.Command object>

polarity = <auspex.instruments.instrument.Command object>

pulse_dcyc = <auspex.instruments.instrument.IntCommand object>

pulse_edge = <auspex.instruments.instrument.FloatCommand object>

pulse_period = <auspex.instruments.instrument.FloatCommand object>

pulse_width = <auspex.instruments.instrument.FloatCommand object>

ramp_symmetry

sequence = <auspex.instruments.instrument.StringCommand object>

set_amplitude (*val*, ***kwargs*)

set_arb_advance (*val*, ***kwargs*)
Advance mode to the next point: 'Trigger' or 'Srate' (Sample Rate)

set_arb_amplitude (*val*, ***kwargs*)

set_arb_frequency (*val*, ***kwargs*)

set_arb_sample (*val*, ***kwargs*)
Sample Rate

set_arb_waveform (*val*, ***kwargs*)

set_auto_range (*val*, ***kwargs*)

set_burst_cycles (*val*, ***kwargs*)

set_burst_mode (*val*, ***kwargs*)

set_burst_state (*val*, ***kwargs*)

set_dc_offset (*val*, ***kwargs*)

set_frequency(*val*, ***kwargs*)

set_function(*val*, ***kwargs*)

set_high_voltage(*val*, ***kwargs*)

set_infinite_load(*channel=1*)

set_load(*val*, ***kwargs*)
Expected load resistance, 1-10k

set_low_voltage(*val*, ***kwargs*)

set_output(*val*, ***kwargs*)

set_output_gated(*val*, ***kwargs*)

set_output_sync(*val*, ***kwargs*)

set_output_trigger(*val*, ***kwargs*)

set_output_trigger_slope(*val*, ***kwargs*)

set_output_trigger_source(*val*, ***kwargs*)

set_output_units(*val*, ***kwargs*)

set_polarity(*val*, ***kwargs*)

set_pulse_dcyc(*val*, ***kwargs*)

set_pulse_edge(*val*, ***kwargs*)

set_pulse_period(*val*, ***kwargs*)

set_pulse_width(*val*, ***kwargs*)

set_ramp_symmetry(*val*, ***kwargs*)

set_sequence(*val*, ***kwargs*)

set_sync_mode(*val*, ***kwargs*)

set_sync_polarity(*val*, ***kwargs*)

set_sync_source(*val*, ***kwargs*)

set_trigger_slope(*val*, ***kwargs*)

set_trigger_source(*val*, ***kwargs*)

sync_mode = <auspex.instruments.instrument.StringCommand object>

sync_polarity = <auspex.instruments.instrument.Command object>

sync_source

trigger(*channel=1*)

trigger_slope

trigger_source

upload_sequence(*sequence*, *channel=1*, *binary=False*)
Upload a sequence to the instrument

upload_waveform(*data*, *channel=1*, *name='mywaveform'*, *dac=True*)
Load string-converted data into a waveform memory

dac: True if values are converted to integer already

upload_waveform_binary(*data*, *channel=1*, *name='mywaveform'*, *dac=True*)
NOT YET WORKING - DO NOT USE Load binary data into a waveform memory

dac: True if values are converted to integer already

class auspex.instruments.agilent.**Agilent34970A**(*resource_name=None*, **args*, ***kwargs*)

Agilent 34970A MUX

ADVSOUR_VALUES = ['EXT', 'BUS', 'IMM']

CONFIG_LIST = []

ONOFF_VALUES = ['ON', 'OFF']

PLC_VALUES = [0.02, 0.2, 1, 10, 20, 100, 200]

RES_VALUES = ['AUTO', 100.0, 1000.0, 10000.0, 100000.0, 1000000.0, 10000000.0, 1000000

TRIGSOUR_VALUES = ['BUS', 'IMM', 'EXT', 'TIM']

advance_source

ch_to_str(*ch_list*)

channel_delay

configlist

connect(*resource_name=None*, *interface_type=None*)
Either connect to the resource name specified during initialization, or specify a new resource name here.

dmm

get_advance_source(***kwargs*)

get_dmm(***kwargs*)

get_trigger_count(***kwargs*)

get_trigger_source(***kwargs*)

get_trigger_timer(***kwargs*)

r_lists()

read()

resistance_range

resistance_resolution

resistance_wire

resistance_zcomp

scan()

scanlist

set_advance_source(*val*, ***kwargs*)

set_dmm(*val*, ***kwargs*)

set_trigger_count(*val*, ***kwargs*)

set_trigger_source(*val*, ***kwargs*)

set_trigger_timer(*val*, ***kwargs*)

trigger_count

trigger_source

trigger_timer

class auspex.instruments.agilent.**AgilentE8363C**(*resource_name=None*, **args*, ***kwargs*)

Agilent E8363C 2-port 40GHz VNA.

data_query_raw = False

ports = (1, 2)

class auspex.instruments.agilent.**AgilentN5183A**(*resource_name=None*, **args*, ***kwargs*)

AgilentN5183A microwave source

alc

connect(*resource_name=None*, *interface_type='VISA'*)

Either connect to the resource name specified during initialization, or specify a new resource name here.

frequency

get_alc(***kwargs*)

get_frequency(***kwargs*)

get_mod(***kwargs*)

get_output(***kwargs*)

get_phase(***kwargs*)

get_power(***kwargs*)

instrument_type = 'Microwave Source'

mod

output

phase

power

reference

set_alc(*val*, ***kwargs*)

set_all(*settings*)

set_frequency(*val*, ***kwargs*)

set_mod(*val*, ***kwargs*)

set_output(*val*, ***kwargs*)

set_phase(*val*, ***kwargs*)

set_power(*val*, ***kwargs*)

class auspex.instruments.agilent.**AgilentN9010A**(*resource_name=None*, **args*, ***kwargs*)

Agilent N9010A SA

averaging_count

clear_averaging()

connect(*resource_name=None*, *interface_type=None*)
Either connect to the resource name specified during initialization, or specify a new resource name here.

frequency_center

frequency_span

frequency_start

frequency_stop

get_averaging_count(***kwargs*)

get_axis()

get_frequency_center(***kwargs*)

get_frequency_span(***kwargs*)

get_frequency_start(***kwargs*)

get_frequency_stop(***kwargs*)

get_marker1_amplitude(***kwargs*)

get_marker1_position(***kwargs*)

get_mode(***kwargs*)

get_num_sweep_points(***kwargs*)

get_pn_carrier_freq(***kwargs*)

get_pn_offset_start(***kwargs*)

get_pn_offset_stop(***kwargs*)

get_pn_trace(*num=3*)

get_resolution_bandwidth(***kwargs*)

get_sweep_time(***kwargs*)

get_trace(*num=1*)

get_video_auto(***kwargs*)

get_video_bandwidth(***kwargs*)

instrument_type = 'Spectrum Analyzer'

marker1_amplitude

marker1_position

marker_X
Queries marker X-value.

Parameters **marker** (*int*) – Marker index (1-12).

Returns X axis value of selected marker.

marker_Y
Queries marker Y-value.

Parameters **marker** (*int*) – Marker index (1-12).

Returns Trace value at selected marker.

marker_to_center(*marker=1*)

mode

noise_marker (*marker=1*, *enable=True*)
Set/unset marker as a noise marker for noise figure measurements.

Parameters

- **marker** (*int*) – Index of marker, [1,12].
- **enable** (*bool*) – Toggles between noise marker (True) and regular marker (False).

Returns None.

num_sweep_points

peak_search (*marker=1*)

pn_carrier_freq

pn_offset_start

pn_offset_stop

resolution_bandwidth

restart_sweep ()
Aborts current sweep and restarts.

set_averaging_count (*val*, ***kwargs*)

set_frequency_center (*val*, ***kwargs*)

set_frequency_span (*val*, ***kwargs*)

set_frequency_start (*val*, ***kwargs*)

set_frequency_stop (*val*, ***kwargs*)

set_marker1_amplitude (*val*, ***kwargs*)

set_marker1_position (*val*, ***kwargs*)

set_mode (*val*, ***kwargs*)

set_num_sweep_points (*val*, ***kwargs*)

set_pn_carrier_freq (*val*, ***kwargs*)

set_pn_offset_start (*val*, ***kwargs*)

set_pn_offset_stop (*val*, ***kwargs*)

set_resolution_bandwidth (*val*, ***kwargs*)

set_sweep_time (*val*, ***kwargs*)

set_video_auto (*val*, ***kwargs*)

set_video_bandwidth (*val*, ***kwargs*)

sweep_time

video_auto

video_bandwidth

class auspex.instruments.agilent.**HP33120A** (*resource_name=None*, **args*, ***kwargs*)
HP33120A Arb Waveform Generator

amplitude

arb_function(*name*)

burst_cycles

burst_source

burst_state

connect(*resource_name=None*, *interface_type=None*)
Either connect to the resource name specified during initialization, or specify a new resource name here.

delete_waveform(*name='all'*)

duty_cycle

frequency

function

get_amplitude(***kwargs*)

get_burst_cycles(***kwargs*)

get_burst_source(***kwargs*)

get_burst_state(***kwargs*)

get_duty_cycle(***kwargs*)

get_frequency(***kwargs*)

get_function(***kwargs*)

get_load(***kwargs*)

get_offset(***kwargs*)

get_voltage_unit(***kwargs*)

load

offset

set_amplitude(*val*, ***kwargs*)

set_burst_cycles(*val*, ***kwargs*)

set_burst_source(*val*, ***kwargs*)

set_burst_state(*val*, ***kwargs*)

set_duty_cycle(*val*, ***kwargs*)

set_frequency(*val*, ***kwargs*)

set_function(*val*, ***kwargs*)

set_load(*val*, ***kwargs*)

set_offset(*val*, ***kwargs*)

set_voltage_unit(*val*, ***kwargs*)

upload_waveform(*data*, *name='volatile'*)

voltage_unit

class auspex.instruments.agilent.**AgilentN5230A**(*resource_name=None*, **args*, ***kwargs*)
Agilent N5230A 4-port 20GHz VNA.

data_query_raw = False

ports = (1, 2, 3, 4)

auspex.instruments.alazar module

class auspex.instruments.alazar.**AlazarATS9870**(*resource_name=None*, *name='Unlabeled Alazar'*, *gen_fake_data=False*)

Alazar ATS9870 digitizer

acquire()

add_channel(*channel*)

configure_with_dict(*settings_dict*)

Accept a sdettings dictionary and attempt to set all of the instrument parameters using the key/value pairs.

connect(*resource_name=None*)

data_available()

disconnect()

done()

get_buffer_for_channel(*channel*)

get_socket(*channel*)

instrument_type = 'Digitizer'

receive_data(*channel*, *oc*, *exit*, *ready*, *run*)

spew_fake_data(*counter*, *ideal_data*, *random_mag=0.1*, *random_seed=12345*)

Generate fake data on the stream. For unittest usage. ideal_data: array or list giving means of the expected signal for each segment

Returns the total number of fake data points, so that we can keep track of how many we expect to receive, when we're doing the test with fake data

stop()

wait_for_acquisition(*dig_run*, *timeout=5*, *ocs=None*, *progressbars=None*)

class auspex.instruments.alazar.**AlazarChannel**(*receiver_channel=None*)

phys_channel = None

set_all(*settings_dict*)

set_by_receiver(*receiver*)

auspex.instruments.ami module

class auspex.instruments.ami.**AMI430**(*resource_name*, **args*, ***kwargs*)

AMI430 Power Supply Programmer

RAMPING_STATES = ['RAMPING to target field/current', 'HOLDING at the target field/curr

SUPPLY_TYPES = ['AMI 12100PS', 'AMI 12200PS', 'AMI 4Q05100PS', 'AMI 4Q06125PS', 'AMI 4

absorber

coil_const

connect (*resource_name=None*, *interface_type=None*)
Either connect to the resource name specified during initialization, or specify a new resource name here.

current_limit

current_magnet

current_max

current_min

current_rating

current_supply

current_target

field

field_target

field_units

get_absorber (***kwargs*)

get_coil_const (***kwargs*)

get_current_limit (***kwargs*)

get_current_magnet (***kwargs*)

get_current_max (***kwargs*)

get_current_min (***kwargs*)

get_current_rating (***kwargs*)

get_current_supply (***kwargs*)

get_current_target (***kwargs*)

get_field (***kwargs*)

get_field_target (***kwargs*)

get_field_units (***kwargs*)

get_inductance (***kwargs*)

get_persistent_switch (***kwargs*)

get_ramp_num_segments (***kwargs*)

get_ramp_rate_units (***kwargs*)

get_ramping_state (***kwargs*)

get_stability (***kwargs*)

get_supply_type (***kwargs*)

get_voltage (***kwargs*)

get_voltage_limit (***kwargs*)

get_voltage_max (***kwargs*)

get_voltage_min (***kwargs*)

inductance

instrument_type = 'Magnet'

pause()
Pauses the Model 430 Programmer at the present operating field/current.

persistent_switch

ramp()
Places the Model 430 Programmer in automatic ramping mode. The Model 430 will continue to ramp at the configured ramp rate(s) until the target field/current is achieved.

ramp_down()
Places the Model 430 Programmer in the MANUAL DOWN ramping mode. Ramping continues at the ramp rate until the Current Limit is achieved (or zero current is achieved for unipolar power supplies).

ramp_num_segments

ramp_rate_units

ramp_up()
Places the Model 430 Programmer in the MANUAL UP ramping mode. Ramping continues at the ramp rate until the Current Limit is achieved.

ramping_state

set_absorber(*val*, ***kwargs*)

set_coil_const(*val*, ***kwargs*)

set_current_limit(*val*, ***kwargs*)

set_current_rating(*val*, ***kwargs*)

set_current_target(*val*, ***kwargs*)

set_field(*val*)
Blocking field setter

set_field_target(*val*, ***kwargs*)

set_field_units(*val*, ***kwargs*)

set_persistent_switch(*val*, ***kwargs*)

set_ramp_num_segments(*val*, ***kwargs*)

set_ramp_rate_units(*val*, ***kwargs*)

set_stability(*val*, ***kwargs*)

set_voltage_limit(*val*, ***kwargs*)

stability

supply_type

voltage

voltage_limit

voltage_max

voltage_min

zero()
Places the Model 430 Programmer in ZEROING CURRENT mode. Ramping automatically initiates and continues at the ramp rate until the power supply output current is less than 0.1% of Imax, at which point the AT ZERO status becomes active.

auspex.instruments.bbn module

class auspex.instruments.bbn.**APS**(*resource_name=None*, *name='Unlabled APS'*)
BBN APSI or DACII

configure_with_proxy(*proxy_obj*)

connect(*resource_name=None*)

disconnect()

get_mixer_correction_matrix()

get_repeat_mode()

get_run_mode()

get_sampling_rate()

get_sequence_file()

get_trigger_interval()

get_trigger_source()

get_waveform_frequency()

instrument_type = 'AWG'

load_waveform(*channel*, *data*)

load_waveform_from_file(*channel*, *filename*)

mixer_correction_matrix

repeat_mode

run_mode

sampling_rate

sequence_file

set_amplitude(*chs*, *value*)

set_mixer_amplitude_imbalance(*chs*, *amp*)

set_mixer_correction_matrix = None

set_mixer_phase_skew(*chs*, *phase*, *SSB=0.0*)

set_offset(*chs*, *value*)

set_repeat_mode(*mode*)

set_run_mode(*mode*)

set_sampling_rate(*freq*)

set_sequence_file(*filename*)

set_trigger_interval(*value*)

set_trigger_source(*source*)

set_waveform_frequency = None

trigger()

trigger_interval

trigger_source

waveform_frequency

class auspex.instruments.bbn.**APS2**(*resource_name=None*, *name='Unlabeled APS2'*)
BBN APS2

amp_factor

configure_with_proxy(*proxy_obj*)

connect(*resource_name=None*)

disconnect()

fpga_temperature

get_amp_factor()

get_fpga_temperature()

get_mixer_correction_matrix()

get_phase_skew()

get_run_mode()

get_sampling_rate()

get_sequence_file()

get_trigger_interval()

get_trigger_source()

get_waveform_frequency()

instrument_type = 'AWG'

load_waveform(*channel*, *data*)

mixer_correction_matrix

phase_skew

run_mode

sampling_rate

sequence_file

set_amp_factor(*amp*)

set_amplitude(*chs*, *value*)

set_fpga_temperature = None

set_mixer_correction_matrix(*matrix*)

set_offset(*chs*, *value*)

set_phase_skew(*skew*)

set_run_mode(*mode*)

set_sampling_rate(*value*)

set_sequence_file(*filename*)

set_trigger_interval(*value*)

set_trigger_source(*source*)

set_waveform_frequency(*freq*)

trigger()

trigger_interval

trigger_source

waveform_frequency

class auspex.instruments.bbn.**TDM**(*resource_name=None*, *name='Unlabeled APS2'*)
BBN TDM

configure_with_proxy(*proxy_obj*)

instrument_type = 'AWG'

class auspex.instruments.bbn.**DigitalAttenuator**(*resource_name=None*, *name='Unlabeled Digital Attenuator'*)
BBN 3 Channel Instrument

NUM_CHANNELS = 3

ch1_attenuation

ch2_attenuation

ch3_attenuation

classmethod channel_check(*chan*)
Assert the channel requested is feasbile

configure_with_proxy(*proxy*)

connect(*resource_name=None*, *interface_type=None*)
Either connect to the resource name specified during initialization, or specify a new resource name here.

get_attenuation(*chan*)

instrument_type = 'Attenuator'

set_attenuation(*chan*, *val*)

class auspex.instruments.bbn.**SpectrumAnalyzer**(*resource_name=None*, **args*, ***kwargs*)
BBN USB Spectrum Analyzer

IF_FREQ = 10700000.0

connect(*resource_name=None*, *interface_type=None*)
Either connect to the resource name specified during initialization, or specify a new resource name here.

get_voltage()

instrument_type = 'Spectrum analyzer'

peak_amplitude()

voltage

auspex.instruments.binutils module

auspex.instruments.bnc module

auspex.instruments.hall_probe module

auspex.instruments.holzworth module

auspex.instruments.instrument module

class auspex.instruments.instrument.**Instrument**

configure_with_dict(*settings_dict*)
Accept a sdettings dictionary and attempt to set all of the instrument parameters using the key/value pairs.

configure_with_proxy(*proxy*)

connect(*resource_name=None*)

disconnect()

auspex.instruments.interface module

class auspex.instruments.interface.**Interface**
Currently just a dummy interface for testing.

close()

query(*value*)

values(*query*)

write(*value*)

class auspex.instruments.interface.**PrologixInterface**(*resource_name*)
Prologix-Ethernet interface for communicating with remote GPIB instruments.

class auspex.instruments.interface.**VisaInterface**(*resource_name*)
PyVISA interface for communicating with instruments.

CLS()

ESE()

ESR()

IDN()

OPC()

RST()

SRE()

STB()

TST()

WAI()

close()

query (*query_string*)

query_ascii_values (*query_string*, ***kwargs*)

query_binary_values (*query_string*, *container=<built-in function array>*, *datatype='h'*, *is_big_endian=False*)

read ()

read_bytes (*count*, *chunk_size=None*, *break_on_termchar=False*)

read_raw (*size=None*)

value (*query_string*)

values (*query_string*)

write (*write_string*)

write_binary_values (*query_string*, *values*, ***kwargs*)

write_raw (*raw_string*)

auspex.instruments.keithley module

auspex.instruments.kepco module

auspex.instruments.keysight module

auspex.instruments.lakeshore module

auspex.instruments.lecroy module

auspex.instruments.magnet module

auspex.instruments.picosecond module

auspex.instruments.prologix module

class auspex.instruments.prologix.**PrologixSocketResource** (*ipaddr=None*, *gpib=None*)

A resource representing a GPIB instrument controlled through a PrologixError GPIB-ETHERNET controller. Mimics the functionality of a pyVISA resource object.

See http://prologix.biz/gpib-ethernet-controller.html for more details and a utility that will discover all prologix instruments on the network.

timeout
Timeout duration for TCP comms. Default 5s.

write_termination
Character added to each outgoing message.

read_termination
Character which denotes the end of a reponse message.

idn_string
GPIB identification command. Defaults to '*IDN?'

bufsize
Maximum amount of data to be received in one call, in bytes.

close()
Close the connection to the Prologix.

connect(*ipaddr=None*, *gpib=None*)
Connect to a GPIB device through a Prologix GPIB-ETHERNET controller. box.

Parameters

- **ipaddr** – The IP address of the Prologix GPIB-ETHERNET.
- **gpib** – The GPIB address of the instrument to be controlled.

Returns None.

query(*command*)
Query instrument with ASCII command then read response.

Parameters command – Message to be sent to instrument.

Returns The instrument data with termination character stripped.

query_ascii_values(*command*, *converter='f'*, *separator=', '*, *container=<class 'list'>*, *bufsize=None*)
Write a string message to device and return values as iterable.

Parameters

- **command** – Message to be sent to device.
- **values** – Data to be written to device (as an interable)
- **converter** – String format code to be used to convert values.
- **separator** – Separator between values – data.split(separator).
- **container** – Iterable type to use for output.
- **bufsize** – Number of bytes to read from instrument. Defaults to resource
- **if None.** (*bufsize*) –

Returns Iterable of values converted from instrument response.

query_binary_values(*command*, *datatype='f'*, *container=<built-in function array>*, *is_big_endian=False*, *bufsize=None*)
Write a string message to device and read binary values, which are returned as iterable. Uses a pyvisa utility function.

Parameters

- **command** – String command sent to instrument.
- **values** – Data to be sent to instrument.
- **datatype** – Format string for single element.
- **container** – Iterable to return number of as.
- **is_big_endian** – Bool indicating endianness.

Returns Iterable of data values to be retuned

read()
Read an ASCII value from the instrument.

Parameters None. –

Returns The instrument data with termination character stripped.

read_raw(*bufsize=None*)
Read bytes from instrument.

Parameters

- **bufsize** – Number of bytes to read from instrument. Defaults to resource
- **if None.** (*bufsize*) –

Returns Instrument data. Nothing is stripped from response.

timeout

write(*command*)
Write a string message to device in ASCII format.

Parameters **command** – The message to be sent.

Returns The number of bytes in the message.

write_ascii_values(*command*, *values*, *converter='f'*, *separator=', '*)
Write a string message to device followed by values in ASCII format.

Parameters

- **command** – Message to be sent to device.
- **values** – Data to be written to device (as an iterable)
- **converter** – String format code to be used to convert values.
- **separator** – Separator between values – separator.join(data).

Returns Total number of bytes sent to instrument.

write_binary_values(*command*, *values*, *datatype='f'*, *is_big_endian=False*)
Write a string message to device followed by values in binary IEEE format using a pyvisa utility function.)

Parameters

- **command** – String command sent to instrument.
- **values** – Data to be sent to instrument.
- **datatype** – Format string for single element.
- **is_big_endian** – Bool indicating endianness.

Returns Number of bytes written to instrument.

write_raw(*command*)
Write a string message to device as raw bytes. No termination character is appended.

Parameters **command** – The message to be sent.

Returns The number of bytes in the message.

auspex.instruments.rfmd module

auspex.instruments.stanford module

auspex.instruments.tektronix module

auspex.instruments.vaunix module

auspex.instruments.X6 module

class auspex.instruments.X6.**X6Channel**(*receiver_channel=None*)

Channel for an X6

set_by_receiver_channel(*receiver*)

class auspex.instruments.X6.**X6**(*resource_name=None*, *name='Unlabeled X6'*, *gen_fake_data=False*)

BBN QDSP running on the II-X6 digitizer

acquire_mode

add_channel(*channel*)

channel_setup(*channel*)

configure_with_dict(*settings_dict*)

Accept a sdettings dictionary and attempt to set all of the instrument parameters using the key/value pairs.

connect(*resource_name=None*)

data_available()

disconnect()

done()

get_buffer_for_channel(*channel*)

get_socket(*channel*)

instrument_type = 'Digitizer'

number_averages

number_segments

number_waveforms

receive_data(*channel*, *oc*, *exit*, *ready*, *run*)

record_length

reference

spew_fake_data(*counter*, *ideal_data*, *random_mag=0.1*, *random_seed=12345*)

Generate fake data on the stream. For unittest usage. ideal_data: array or list giving means of the expected signal for each segment

Returns the total number of fake data points, so that we can keep track of how many we expect to receive, when we're doing the test with fake data

wait_for_acquisition(*dig_run*, *timeout=15*, *ocs=None*, *progressbars=None*)

auspex.instruments.yokogawa module

Module contents

auspex.qubit package

Submodules

auspex.qubit.pipeline module

auspex.qubit.qubit_exp module

auspex.qubit.mixer_calibration module

auspex.qubit.pulse_calibration module

auspex.qubit.single_shot_fidelity module

Module contents

3.6.2 auspex.config module

auspex.config.**isnotebook**()

auspex.config.**load_db**()

3.6.3 auspex.data_format module

class auspex.data_format.**AuspexDataContainer**(*base_path*, *mode='a'*, *open_all=True*)

A container for Auspex data. Data is stored as *datasets* which may be of any dimension. These are in turn organized by *groups* which can be used to store related information. Data is stored as a binary file plus a json metafile which describes the dimension and type of data stored.

Example organization

DataContainer:

- QubitOneGroup
  - | - DemodulatedData
  - | - ThresholdedData
- QubitTwoGroup | - RawData | - DemodulatedData

close()

Close the data container.

new_dataset(*groupname*, *datasetname*, *descriptor*)

Add a dataset to a specific group.

Parameters

- **groupname** – Name of the group to which to add the dataset.
- **datasetname** – Name of the dataset to be added.
- **descriptor** – *DataStreamDescriptor* that describes the dataset that is to be added.

new_group(*groupname*)

Add a group to the data container.

Parameters **groupname** – Name of the data group to be added to the data container.

open_all()

Open all of the datasets contained in this DataContainer. This also populates the list of groups.

Returns A dictionary of all of the datasets, which each item as an (array, descriptor) tuple.

open_dataset (*groupname*, *datasetname*)
Open a particular dataset stored in this DataContainer.

Parameters

- **groupname** – The group name of the data that is to be opened.
- **datasetname** – The name of the dataset that is to be opened.

Returns A numpy array of the data stored. desc: *DataStreamDescriptor* for the data stored.

Return type data

3.6.4 auspex.experiment module

3.6.5 auspex.log module

auspex.log.**in_jupyter**()

3.6.6 auspex.parameter module

class auspex.parameter.**BoolParameter** (*name=None*, *unit=None*, *default=None*, *value_range=None*, *allowed_values=None*, *increment=None*, *snap=None*)

value

class auspex.parameter.**FilenameParameter** (**args*, ***kwargs*)

class auspex.parameter.**FloatParameter** (*name=None*, *unit=None*, *default=None*, *value_range=None*, *allowed_values=None*, *increment=None*, *snap=None*)

value

class auspex.parameter.**IntParameter** (*name=None*, *unit=None*, *default=None*, *value_range=None*, *allowed_values=None*, *increment=None*, *snap=None*)

value

class auspex.parameter.**Parameter** (*name=None*, *unit=None*, *default=None*, *value_range=None*, *allowed_values=None*, *increment=None*, *snap=None*)
Encapsulates the information for an experiment parameter

add_post_push_hook (*hook*)

add_pre_push_hook (*hook*)

assign_method (*method*)

dict_repr ()
Return a dictionary representation. Intended for Quince interop.

push ()

value

class auspex.parameter.**ParameterGroup**(*params*, *name=None*)
An array of Parameters

assign_method(*methods*)

push()

value

3.6.7 auspex.stream module

class auspex.stream.**DataAxis**(*name*, *points=[]*, *unit=None*, *metadata=None*, *dtype=<class 'numpy.float32'>*)
An axis in a data stream

add_points(*points*)

data_type(*with_metadata=False*)

num_points()

points_with_metadata()

reset()

tuple_width()

class auspex.stream.**DataStream**(*name=None*, *unit=None*)
A stream of data

done()

final_init()

num_points()

percent_complete()

pop()

push(*data*)

push_event(*event_type*, *data=None*)

reset()

set_descriptor(*descriptor*)

class auspex.stream.**DataStreamDescriptor**(*dtype=<class 'numpy.float32'>*)
Axes information

add_axis(*axis*, *position=0*)

add_param(*key*, *value*)

axes_done()

axis(*axis_name*)

axis_data_type(*with_metadata=False*, *excluding_axis=None*)

axis_names(*with_metadata=False*)
Returns all axis names included those from unstructured axes

axis_num(*axis_name*)

copy()

data_axis_values()
Returns a list of point lists for each data axis, ignoring sweep axes.

data_dims()

dims()

done()

expected_num_points()

expected_tuples(*with_metadata=False*, *as_structured_array=True*)
Returns a list of tuples representing the cartesian product of the axis values. Should only be used with non-adaptive sweeps.

extent(*flip=False*)
Convenience function for matplotlib.imshow, which expects extent=(left, right, bottom, top).

is_adaptive()

last_data_axis()

num_data_axis_points()

num_dims()

num_new_points_through_axis(*axis_name*)

num_points()

num_points_through_axis(*axis_name*)

pop_axis(*axis_name*)

reset()

tuple_width()

tuples(*as_structured_array=True*)
Returns a list of all tuples visited by the sweeper. Should only be used with adaptive sweeps.

class auspex.stream.**InputConnector**(*name=''*, *parent=None*, *datatype=None*, *max_input_streams=1*)

add_input_stream(*stream*)

done()

num_points()

update_descriptors()

class auspex.stream.**OutputConnector**(*name=''*, *data_name=None*, *unit=None*, *parent=None*, *dtype=<class 'numpy.float32'>*)

add_output_stream(*stream*)

done()

num_points()

push(*data*)

push_event(*event_type*, *data=None*)

set_descriptor(*descriptor*)

update_descriptors()

class auspex.stream.**SweepAxis**(*parameter*, *points=[]*, *metadata=None*, *refine_func=None*, *callback_func=None*)

Structure for sweep axis, separate from DataAxis. Can be an unstructured axis, in which case 'parameter' is actually a list of parameters.

check_for_refinement(*output_connectors_dict*)

Check to see if we need to perform any refinements. If there is a refine_func and it returns a list of points, then we need to extend the axes. Otherwise, if the refine_func returns None or false, then we reset the axis to its original set of points. If there is no refine_func then we don't do anything at all.

push()

Push parameter value(s)

update()

Update value after each run.

auspex.stream.**cartesian**(*arrays*, *out=None*, *dtype='f'*)

http://stackoverflow.com/questions/28684492/numpy-equivalent-of-itertools-product

3.6.8 auspex.sweep module

class auspex.sweep.**Sweeper**

Control center of sweep axes

add_sweep(*axis*)

check_for_refinement(*output_connectors_dict*)

done()

is_adaptive()

swept_parameters()

update()

Update the levels

Python Module Index

a

auspex.analysis, 53
auspex.analysis.fits, 49
auspex.analysis.helpers, 51
auspex.config, 81
auspex.data_format, 81
auspex.filters.channelizer, 53
auspex.filters.correlator, 54
auspex.filters.debug, 54
auspex.filters.elementwise, 54
auspex.filters.filter, 55
auspex.filters.framer, 55
auspex.filters.integrator, 55
auspex.filters.io, 56
auspex.filters.plot, 57
auspex.filters.singleshot, 58
auspex.instruments.agilent, 59
auspex.instruments.alazar, 70
auspex.instruments.ami, 70
auspex.instruments.bbn, 73
auspex.instruments.instrument, 76
auspex.instruments.interface, 76
auspex.instruments.prologix, 77
auspex.instruments.X6, 80
auspex.log, 82
auspex.parameter, 82
auspex.stream, 83
auspex.sweep, 85

Index

A

abort() (*auspex.instruments.agilent.Agilent33500B method*), 61
absorber (*auspex.instruments.ami.AMI430 attribute*), 70
acquire() (*auspex.instruments.alazar.AlazarATS9870 method*), 70
acquire_mode (*auspex.instruments.X6.X6 attribute*), 80
add_axis() (*auspex.stream.DataStreamDescriptor method*), 83
add_channel() (*auspex.instruments.alazar.AlazarATS9870 method*), 70
add_channel() (*auspex.instruments.X6.X6 method*), 80
add_data_trace() (*auspex.filters.plot.ManualPlotter method*), 57
add_fit_trace() (*auspex.filters.plot.ManualPlotter method*), 57
add_input_stream() (*auspex.stream.InputConnector method*), 84
add_output_stream() (*auspex.stream.OutputConnector method*), 84
add_param() (*auspex.stream.DataStreamDescriptor method*), 83
add_points() (*auspex.stream.DataAxis method*), 83
add_post_push_hook() (*auspex.parameter.Parameter method*), 82
add_pre_push_hook() (*auspex.parameter.Parameter method*), 82
add_segment() (*auspex.instruments.agilent.Agilent33500B.Sequence method*), 61
add_sweep() (*auspex.sweep.Sweeper method*), 85
add_trace() (*auspex.filters.plot.ManualPlotter method*), 57
advance_source (*auspex.instruments.agilent.Agilent34970A attribute*), 65
ADVSOUR_VALUES (*auspex.instruments.agilent.Agilent34970A attribute*), 65
Agilent33220A (*class in auspex.instruments.agilent*), 59
Agilent33500B (*class in auspex.instruments.agilent*), 61
Agilent33500B.Segment (*class in auspex.instruments.agilent*), 61
Agilent33500B.Sequence (*class in auspex.instruments.agilent*), 61
Agilent34970A (*class in auspex.instruments.agilent*), 65
AgilentE8363C (*class in auspex.instruments.agilent*), 66
AgilentN5183A (*class in auspex.instruments.agilent*), 66
AgilentN5230A (*class in auspex.instruments.agilent*), 69
AgilentN9010A (*class in auspex.instruments.agilent*), 66
AlazarATS9870 (*class in auspex.instruments.alazar*), 70
AlazarChannel (*class in auspex.instruments.alazar*), 70
alc (*auspex.instruments.agilent.AgilentN5183A attribute*), 66
AMI430 (*class in auspex.instruments.ami*), 70
amp_factor (*auspex.instruments.bbn.APS2 attribute*), 74
amplitude (*auspex.instruments.agilent.Agilent33220A attribute*), 59
amplitude (*auspex.instruments.agilent.Agilent33500B attribute*), 61
amplitude (*auspex.instruments.agilent.HP33120A attribute*), 68
annotation() (*auspex.analysis.fits.Auspex2DFit method*), 49
annotation() (*auspex.analysis.fits.AuspexFit

method), 50
APS (*class in auspex.instruments.bbn*), 73
APS2 (*class in auspex.instruments.bbn*), 74
arb_advance (*auspex.instruments.agilent.Agilent33500B attribute*), 61
arb_amplitude (*auspex.instruments.agilent.Agilent33500B attribute*), 61
arb_frequency (*auspex.instruments.agilent.Agilent33500B attribute*), 61
arb_function() (*auspex.instruments.agilent.HP33120A method*), 68
arb_sample (*auspex.instruments.agilent.Agilent33500B attribute*), 61
arb_sync() (*auspex.instruments.agilent.Agilent33500B method*), 61
arb_waveform (*auspex.instruments.agilent.Agilent33500B attribute*), 61
assign_method() (*auspex.parameter.Parameter method*), 82
assign_method() (*auspex.parameter.ParameterGroup method*), 83
auspex.analysis (*module*), 53
auspex.analysis.fits (*module*), 49
auspex.analysis.helpers (*module*), 51
auspex.config (*module*), 81
auspex.data_format (*module*), 81
auspex.filters.channelizer (*module*), 53
auspex.filters.correlator (*module*), 54
auspex.filters.debug (*module*), 54
auspex.filters.elementwise (*module*), 54
auspex.filters.filter (*module*), 55
auspex.filters.framer (*module*), 55
auspex.filters.integrator (*module*), 55
auspex.filters.io (*module*), 56
auspex.filters.plot (*module*), 57
auspex.filters.singleshot (*module*), 58
auspex.instruments.agilent (*module*), 59
auspex.instruments.alazar (*module*), 70
auspex.instruments.ami (*module*), 70
auspex.instruments.bbn (*module*), 73
auspex.instruments.instrument (*module*), 76
auspex.instruments.interface (*module*), 76
auspex.instruments.prologix (*module*), 77
auspex.instruments.X6 (*module*), 80
auspex.log (*module*), 82
auspex.parameter (*module*), 82
auspex.stream (*module*), 83
auspex.sweep (*module*), 85
Auspex2DFit (*class in auspex.analysis.fits*), 49
AuspexDataContainer (*class in auspex.data_format*), 81
AuspexFit (*class in auspex.analysis.fits*), 49
auto_range (*auspex.instruments.agilent.Agilent33220A attribute*), 59
auto_range (*auspex.instruments.agilent.Agilent33500B attribute*), 61
averaging_count (*auspex.instruments.agilent.AgilentN9010A attribute*), 66
ax (*auspex.analysis.fits.AuspexFit attribute*), 50
axes_done() (*auspex.stream.DataStreamDescriptor method*), 83
axis (*auspex.filters.framer.Framer attribute*), 55
axis() (*auspex.stream.DataStreamDescriptor method*), 83
axis_data_type() (*auspex.stream.DataStreamDescriptor method*), 83
axis_label() (*auspex.filters.plot.Plotter method*), 57
axis_names() (*auspex.stream.DataStreamDescriptor method*), 83
axis_num() (*auspex.stream.DataStreamDescriptor method*), 83

B

bandwidth (*auspex.filters.channelizer.Channelizer attribute*), 53
bias (*auspex.filters.integrator.KernelIntegrator attribute*), 55
BoolParameter (*class in auspex.parameter*), 82
bounds (*auspex.analysis.fits.AuspexFit attribute*), 50
box_car_start (*auspex.filters.integrator.KernelIntegrator attribute*), 56
box_car_stop (*auspex.filters.integrator.KernelIntegrator attribute*), 56
bufsize (*auspex.instruments.prologix.PrologixSocketResource attribute*), 77
burst_cycles (*auspex.instruments.agilent.Agilent33220A attribute*), 59
burst_cycles (*auspex.instruments.agilent.Agilent33500B attribute*), 61
burst_cycles (*auspex.instruments.agilent.HP33120A attribute*), 69
burst_mode (*auspex.instruments.agilent.Agilent33220A attribute*), 59
burst_mode (*auspex.instruments.agilent.Agilent33500B attribute*), 61

burst_source (auspex.instruments.agilent.HP33120A attribute), 69
burst_state (auspex.instruments.agilent.Agilent33220A attribute), 59
burst_state (auspex.instruments.agilent.Agilent33500B attribute), 61
burst_state (auspex.instruments.agilent.HP33120A attribute), 69

C

cal_data() (in module auspex.analysis.helpers), 51
cal_ls() (in module auspex.analysis.helpers), 51
cal_scale() (in module auspex.analysis.helpers), 51
cartesian() (in module auspex.stream), 85
ch1_attenuation (auspex.instruments.bbn.DigitalAttenuator attribute), 75
ch2_attenuation (auspex.instruments.bbn.DigitalAttenuator attribute), 75
ch3_attenuation (auspex.instruments.bbn.DigitalAttenuator attribute), 75
ch_to_str() (auspex.instruments.agilent.Agilent34970A method), 65
channel_check() (auspex.instruments.bbn.DigitalAttenuator class method), 75
channel_delay (auspex.instruments.agilent.Agilent34970A attribute), 65
channel_setup() (auspex.instruments.X6.X6 method), 80
Channelizer (class in auspex.filters.channelizer), 53
check_for_refinement() (auspex.stream.SweepAxis method), 85
check_for_refinement() (auspex.sweep.Sweeper method), 85
checkin() (auspex.filters.filter.Filter method), 55
checkin() (auspex.filters.io.DataBuffer method), 56
clear_averaging() (auspex.instruments.agilent.AgilentN9010A method), 66
clear_waveform() (auspex.instruments.agilent.Agilent33500B method), 61
close() (auspex.data_format.AuspexDataContainer method), 81
close() (auspex.instruments.interface.Interface method), 76
close() (auspex.instruments.interface.VisaInterface method), 76
close() (auspex.instruments.prologix.PrologixSocketResource method), 78
CLS() (auspex.instruments.interface.VisaInterface method), 76
coil_const (auspex.instruments.ami.AMI430 attribute), 70
compute_filter() (auspex.filters.singleshot.SingleShotMeasurement method), 58
CONFIG_LIST (auspex.instruments.agilent.Agilent34970A attribute), 65
configlist (auspex.instruments.agilent.Agilent34970A attribute), 65
configure_with_dict() (auspex.instruments.alazar.AlazarATS9870 method), 70
configure_with_dict() (auspex.instruments.instrument.Instrument method), 76
configure_with_dict() (auspex.instruments.X6.X6 method), 80
configure_with_proxy() (auspex.filters.filter.Filter method), 55
configure_with_proxy() (auspex.instruments.bbn.APS method), 73
configure_with_proxy() (auspex.instruments.bbn.APS2 method), 74
configure_with_proxy() (auspex.instruments.bbn.DigitalAttenuator method), 75
configure_with_proxy() (auspex.instruments.bbn.TDM method), 75
configure_with_proxy() (auspex.instruments.instrument.Instrument method), 76
connect() (auspex.instruments.agilent.Agilent33220A method), 59
connect() (auspex.instruments.agilent.Agilent33500B method), 62
connect() (auspex.instruments.agilent.Agilent34970A method), 65
connect() (auspex.instruments.agilent.AgilentN5183A method), 66
connect() (auspex.instruments.agilent.AgilentN9010A method), 66
connect() (auspex.instruments.agilent.HP33120A method), 69
connect() (auspex.instruments.alazar.AlazarATS9870 method), 70
connect() (auspex.instruments.ami.AMI430 method), 71
connect() (auspex.instruments.bbn.APS method), 73
connect() (auspex.instruments.bbn.APS2 method), 74
connect() (auspex.instruments.bbn.DigitalAttenuator

method), 75
connect() (*auspex.instruments.bbn.SpectrumAnalyzer method*), 75
connect() (*auspex.instruments.instrument.Instrument method*), 76
connect() (*auspex.instruments.prologix.PrologixSocketResource method*), 78
connect() (*auspex.instruments.X6.X6 method*), 80
copy() (*auspex.stream.DataStreamDescriptor method*), 83
Correlator (*class in auspex.filters.correlator*), 54
current_limit (*auspex.instruments.ami.AMI430 attribute*), 71
current_magnet (*auspex.instruments.ami.AMI430 attribute*), 71
current_max (*auspex.instruments.ami.AMI430 attribute*), 71
current_min (*auspex.instruments.ami.AMI430 attribute*), 71
current_rating (*auspex.instruments.ami.AMI430 attribute*), 71
current_supply (*auspex.instruments.ami.AMI430 attribute*), 71
current_target (*auspex.instruments.ami.AMI430 attribute*), 71

D

data_available() (*auspex.instruments.alazar.AlazarATS9870 method*), 70
data_available() (*auspex.instruments.X6.X6 method*), 80
data_axis_values() (*auspex.stream.DataStreamDescriptor method*), 83
data_dims() (*auspex.stream.DataStreamDescriptor method*), 84
data_query_raw (*auspex.instruments.agilent.AgilentE8363C attribute*), 66
data_query_raw (*auspex.instruments.agilent.AgilentN5230A attribute*), 69
data_type() (*auspex.stream.DataAxis method*), 83
DataAxis (*class in auspex.stream*), 83
DataBuffer (*class in auspex.filters.io*), 56
datasetname (*auspex.filters.io.WriteToFile attribute*), 56
DataStream (*class in auspex.stream*), 83
DataStreamDescriptor (*class in auspex.stream*), 83
dc_offset (*auspex.instruments.agilent.Agilent33220A attribute*), 59
dc_offset (*auspex.instruments.agilent.Agilent33500B attribute*), 62
decimation_factor (*auspex.filters.channelizer.Channelizer attribute*), 53
delete_waveform() (*auspex.instruments.agilent.HP33120A method*), 69
demod_frequency (*auspex.filters.integrator.KernelIntegrator attribute*), 56
desc() (*auspex.filters.plot.ManualPlotter method*), 57
desc() (*auspex.filters.plot.MeshPlotter method*), 58
desc() (*auspex.filters.plot.Plotter method*), 57
descriptor_map() (*auspex.filters.filter.Filter method*), 55
dict_repr() (*auspex.parameter.Parameter method*), 82
DigitalAttenuator (*class in auspex.instruments.bbn*), 75
dims() (*auspex.stream.DataStreamDescriptor method*), 84
disconnect() (*auspex.instruments.alazar.AlazarATS9870 method*), 70
disconnect() (*auspex.instruments.bbn.APS method*), 73
disconnect() (*auspex.instruments.bbn.APS2 method*), 74
disconnect() (*auspex.instruments.instrument.Instrument method*), 76
disconnect() (*auspex.instruments.X6.X6 method*), 80
dmm (*auspex.instruments.agilent.Agilent34970A attribute*), 65
done() (*auspex.instruments.alazar.AlazarATS9870 method*), 70
done() (*auspex.instruments.X6.X6 method*), 80
done() (*auspex.stream.DataStream method*), 83
done() (*auspex.stream.DataStreamDescriptor method*), 84
done() (*auspex.stream.InputConnector method*), 84
done() (*auspex.stream.OutputConnector method*), 84
done() (*auspex.sweep.Sweeper method*), 85
duty_cycle (*auspex.instruments.agilent.HP33120A attribute*), 69

E

ElementwiseFilter (*class in auspex.filters.elementwise*), 54
ESE() (*auspex.instruments.interface.VisaInterface method*), 76

ESR() (*auspex.instruments.interface.VisaInterface method*), 76
execute_on_run() (*auspex.filters.filter.Filter method*), 55
execute_on_run() (*auspex.filters.plot.ManualPlotter method*), 57
execute_on_run() (*auspex.filters.plot.MeshPlotter method*), 58
execute_on_run() (*auspex.filters.plot.Plotter method*), 57
expected_num_points() (*auspex.stream.DataStreamDescriptor method*), 84
expected_tuples() (*auspex.stream.DataStreamDescriptor method*), 84
extent() (*auspex.stream.DataStreamDescriptor method*), 84

F

field (*auspex.instruments.ami.AMI430 attribute*), 71
field_target (*auspex.instruments.ami.AMI430 attribute*), 71
field_units (*auspex.instruments.ami.AMI430 attribute*), 71
filename (*auspex.filters.io.WriteToFile attribute*), 56
FilenameParameter (*class in auspex.parameter*), 82
Filter (*class in auspex.filters.filter*), 55
filter_name (*auspex.filters.correlator.Correlator attribute*), 54
filter_name (*auspex.filters.elementwise.ElementwiseFilter attribute*), 54
final_init() (*auspex.filters.channelizer.Channelizer method*), 53
final_init() (*auspex.filters.framer.Framer method*), 55
final_init() (*auspex.filters.io.DataBuffer method*), 56
final_init() (*auspex.filters.io.WriteToFile method*), 56
final_init() (*auspex.filters.plot.Plotter method*), 57
final_init() (*auspex.filters.singleshot.SingleShotMeasurement method*), 58
final_init() (*auspex.stream.DataStream method*), 83
FloatParameter (*class in auspex.parameter*), 82
follow_axis (*auspex.filters.channelizer.Channelizer attribute*), 53
follow_freq_offset (*auspex.filters.channelizer.Channelizer attribute*), 53
fpga_temperature (*auspex.instruments.bbn.APS2 attribute*), 74
Framer (*class in auspex.filters.framer*), 55
frequency (*auspex.filters.channelizer.Channelizer attribute*), 53
frequency (*auspex.instruments.agilent.Agilent33220A attribute*), 59
frequency (*auspex.instruments.agilent.Agilent33500B attribute*), 62
frequency (*auspex.instruments.agilent.AgilentN5183A attribute*), 66
frequency (*auspex.instruments.agilent.HP33120A attribute*), 69
frequency_center (*auspex.instruments.agilent.AgilentN9010A attribute*), 67
frequency_span (*auspex.instruments.agilent.AgilentN9010A attribute*), 67
frequency_start (*auspex.instruments.agilent.AgilentN9010A attribute*), 67
frequency_stop (*auspex.instruments.agilent.AgilentN9010A attribute*), 67
function (*auspex.instruments.agilent.Agilent33220A attribute*), 59
function (*auspex.instruments.agilent.Agilent33500B attribute*), 62
function (*auspex.instruments.agilent.HP33120A attribute*), 69
FUNCTION_MAP (*auspex.instruments.agilent.Agilent33220A attribute*), 59
FUNCTION_MAP (*auspex.instruments.agilent.Agilent33500B attribute*), 61

G

GaussianFit (*class in auspex.analysis.fits*), 50
get_absorber() (*auspex.instruments.ami.AMI430 method*), 71
get_advance_source() (*auspex.instruments.agilent.Agilent34970A method*), 65
get_alc() (*auspex.instruments.agilent.AgilentN5183A method*), 66
get_amp_factor() (*auspex.instruments.bbn.APS2 method*), 74
get_amplitude() (*auspex.instruments.agilent.Agilent33220A method*), 59
get_amplitude() (*auspex.instruments.agilent.Agilent33500B method*), 62

get_amplitude() (auspex.instruments.agilent.HP33120A method), 69
get_arb_advance() (auspex.instruments.agilent.Agilent33500B method), 62
get_arb_amplitude() (auspex.instruments.agilent.Agilent33500B method), 62
get_arb_frequency() (auspex.instruments.agilent.Agilent33500B method), 62
get_arb_sample() (auspex.instruments.agilent.Agilent33500B method), 62
get_arb_waveform() (auspex.instruments.agilent.Agilent33500B method), 62
get_attenuation() (auspex.instruments.bbn.DigitalAttenuator method), 75
get_auto_range() (auspex.instruments.agilent.Agilent33220A method), 59
get_auto_range() (auspex.instruments.agilent.Agilent33500B method), 62
get_averaging_count() (auspex.instruments.agilent.AgilentN9010A method), 67
get_axis() (auspex.instruments.agilent.AgilentN9010A method), 67
get_buffer_for_channel() (auspex.instruments.alazar.AlazarATS9870 method), 70
get_buffer_for_channel() (auspex.instruments.X6.X6 method), 80
get_burst_cycles() (auspex.instruments.agilent.Agilent33220A method), 59
get_burst_cycles() (auspex.instruments.agilent.Agilent33500B method), 62
get_burst_cycles() (auspex.instruments.agilent.HP33120A method), 69
get_burst_mode() (auspex.instruments.agilent.Agilent33220A method), 59
get_burst_mode() (auspex.instruments.agilent.Agilent33500B method), 62
get_burst_source() (auspex.instruments.agilent.HP33120A method), 69
get_burst_state() (auspex.instruments.agilent.Agilent33220A method), 59
get_burst_state() (auspex.instruments.agilent.Agilent33500B method), 62
get_burst_state() (auspex.instruments.agilent.HP33120A method), 69
get_cals() (in module auspex.analysis.helpers), 51
get_coil_const() (auspex.instruments.ami.AMI430 method), 71
get_current_limit() (auspex.instruments.ami.AMI430 method), 71
get_current_magnet() (auspex.instruments.ami.AMI430 method), 71
get_current_max() (auspex.instruments.ami.AMI430 method), 71
get_current_min() (auspex.instruments.ami.AMI430 method), 71
get_current_rating() (auspex.instruments.ami.AMI430 method), 71
get_current_supply() (auspex.instruments.ami.AMI430 method), 71
get_current_target() (auspex.instruments.ami.AMI430 method), 71
get_data() (auspex.filters.io.DataBuffer method), 56
get_data() (auspex.filters.io.WriteToFile method), 56
get_data_while_running() (auspex.filters.io.WriteToFile method), 56
get_dc_offset() (auspex.instruments.agilent.Agilent33220A method), 59
get_dc_offset() (auspex.instruments.agilent.Agilent33500B method), 62
get_descriptor() (auspex.instruments.agilent.Agilent33500B.Sequence method), 61
get_dmm() (auspex.instruments.agilent.Agilent34970A method), 65
get_duty_cycle() (auspex.instruments.agilent.HP33120A method), 69
get_field() (auspex.instruments.ami.AMI430 method), 71
get_field_target() (auspex.instruments.ami.AMI430 method), 71
get_field_units() (auspex.instruments.ami.AMI430 method), 71
get_file_name() (in module auspex.analysis.helpers), 52
get_final_plot() (auspex.filters.plot.Plotter

method), 57
get_fpga_temperature() (auspex.instruments.bbn.APS2 method), 74
get_frequency() (auspex.instruments.agilent.Agilent33220A method), 59
get_frequency() (auspex.instruments.agilent.Agilent33500B method), 62
get_frequency() (auspex.instruments.agilent.AgilentN5183A method), 66
get_frequency() (auspex.instruments.agilent.HP33120A method), 69
get_frequency_center() (auspex.instruments.agilent.AgilentN9010A method), 67
get_frequency_span() (auspex.instruments.agilent.AgilentN9010A method), 67
get_frequency_start() (auspex.instruments.agilent.AgilentN9010A method), 67
get_frequency_stop() (auspex.instruments.agilent.AgilentN9010A method), 67
get_function() (auspex.instruments.agilent.Agilent33220A method), 59
get_function() (auspex.instruments.agilent.Agilent33500B method), 62
get_function() (auspex.instruments.agilent.HP33120A method), 69
get_high_voltage() (auspex.instruments.agilent.Agilent33220A method), 59
get_high_voltage() (auspex.instruments.agilent.Agilent33500B method), 62
get_inductance() (auspex.instruments.ami.AMI430 method), 71
get_load() (auspex.instruments.agilent.Agilent33500B method), 62
get_load() (auspex.instruments.agilent.HP33120A method), 69
get_load_resistance() (auspex.instruments.agilent.Agilent33220A method), 59
get_low_voltage() (auspex.instruments.agilent.Agilent33220A method), 59
get_low_voltage() (auspex.instruments.agilent.Agilent33500B method), 62
get_marker1_amplitude() (auspex.instruments.agilent.AgilentN9010A method), 67
get_marker1_position() (auspex.instruments.agilent.AgilentN9010A method), 67
get_mixer_correction_matrix() (auspex.instruments.bbn.APS method), 73
get_mixer_correction_matrix() (auspex.instruments.bbn.APS2 method), 74
get_mod() (auspex.instruments.agilent.AgilentN5183A method), 66
get_mode() (auspex.instruments.agilent.AgilentN9010A method), 67
get_num_sweep_points() (auspex.instruments.agilent.AgilentN9010A method), 67
get_offset() (auspex.instruments.agilent.HP33120A method), 69
get_output() (auspex.instruments.agilent.Agilent33220A method), 59
get_output() (auspex.instruments.agilent.Agilent33500B method), 62
get_output() (auspex.instruments.agilent.AgilentN5183A method), 66
get_output_gated() (auspex.instruments.agilent.Agilent33500B method), 62
get_output_sync() (auspex.instruments.agilent.Agilent33220A method), 59
get_output_sync() (auspex.instruments.agilent.Agilent33500B method), 62
get_output_trigger() (auspex.instruments.agilent.Agilent33500B method), 62
get_output_trigger_slope() (auspex.instruments.agilent.Agilent33500B method), 62
get_output_trigger_source() (auspex.instruments.agilent.Agilent33500B method), 62
get_output_units() (auspex.instruments.agilent.Agilent33220A method), 59
get_output_units() (aus-

pex.instruments.agilent.Agilent33500B method), 62
get_persistent_switch() (*auspex.instruments.ami.AMI430 method*), 71
get_phase() (*auspex.instruments.agilent.AgilentN5183A method*), 66
get_phase_skew() (*auspex.instruments.bbn.APS2 method*), 74
get_pn_carrier_freq() (*auspex.instruments.agilent.AgilentN9010A method*), 67
get_pn_offset_start() (*auspex.instruments.agilent.AgilentN9010A method*), 67
get_pn_offset_stop() (*auspex.instruments.agilent.AgilentN9010A method*), 67
get_pn_trace() (*auspex.instruments.agilent.AgilentN9010A method*), 67
get_polarity() (*auspex.instruments.agilent.Agilent33220A method*), 59
get_polarity() (*auspex.instruments.agilent.Agilent33500B method*), 62
get_power() (*auspex.instruments.agilent.AgilentN5183A method*), 66
get_pulse_dcyc() (*auspex.instruments.agilent.Agilent33220A method*), 60
get_pulse_dcyc() (*auspex.instruments.agilent.Agilent33500B method*), 62
get_pulse_edge() (*auspex.instruments.agilent.Agilent33220A method*), 60
get_pulse_edge() (*auspex.instruments.agilent.Agilent33500B method*), 62
get_pulse_period() (*auspex.instruments.agilent.Agilent33220A method*), 60
get_pulse_period() (*auspex.instruments.agilent.Agilent33500B method*), 62
get_pulse_width() (*auspex.instruments.agilent.Agilent33220A method*), 60
get_pulse_width() (*auspex.instruments.agilent.Agilent33500B method*), 62
get_ramp_num_segments() (*auspex.instruments.ami.AMI430 method*), 71
get_ramp_rate_units() (*auspex.instruments.ami.AMI430 method*), 71
get_ramp_symmetry() (*auspex.instruments.agilent.Agilent33220A method*), 60
get_ramp_symmetry() (*auspex.instruments.agilent.Agilent33500B method*), 62
get_ramping_state() (*auspex.instruments.ami.AMI430 method*), 71
get_repeat_mode() (*auspex.instruments.bbn.APS method*), 73
get_resolution_bandwidth() (*auspex.instruments.agilent.AgilentN9010A method*), 67
get_run_mode() (*auspex.instruments.bbn.APS method*), 73
get_run_mode() (*auspex.instruments.bbn.APS2 method*), 74
get_sampling_rate() (*auspex.instruments.bbn.APS method*), 73
get_sampling_rate() (*auspex.instruments.bbn.APS2 method*), 74
get_sequence() (*auspex.instruments.agilent.Agilent33500B method*), 63
get_sequence_file() (*auspex.instruments.bbn.APS method*), 73
get_sequence_file() (*auspex.instruments.bbn.APS2 method*), 74
get_socket() (*auspex.instruments.alazar.AlazarATS9870 method*), 70
get_socket() (*auspex.instruments.X6.X6 method*), 80
get_stability() (*auspex.instruments.ami.AMI430 method*), 71
get_supply_type() (*auspex.instruments.ami.AMI430 method*), 71
get_sweep_time() (*auspex.instruments.agilent.AgilentN9010A method*), 67
get_sync_mode() (*auspex.instruments.agilent.Agilent33500B method*), 63
get_sync_polarity() (*auspex.instruments.agilent.Agilent33500B method*), 63
get_sync_source() (*auspex.instruments.agilent.Agilent33500B method*), 63
get_trace() (*auspex.instruments.agilent.AgilentN9010A method*), 67
get_trigger_count() (*aus-

pex.instruments.agilent.Agilent34970A method), 65
get_trigger_interval() (auspex.instruments.bbn.APS method), 73
get_trigger_interval() (auspex.instruments.bbn.APS2 method), 74
get_trigger_slope() (auspex.instruments.agilent.Agilent33220A method), 60
get_trigger_slope() (auspex.instruments.agilent.Agilent33500B method), 63
get_trigger_source() (auspex.instruments.agilent.Agilent33220A method), 60
get_trigger_source() (auspex.instruments.agilent.Agilent33500B method), 63
get_trigger_source() (auspex.instruments.agilent.Agilent34970A method), 65
get_trigger_source() (auspex.instruments.bbn.APS method), 73
get_trigger_source() (auspex.instruments.bbn.APS2 method), 74
get_trigger_timer() (auspex.instruments.agilent.Agilent34970A method), 65
get_video_auto() (auspex.instruments.agilent.AgilentN9010A method), 67
get_video_bandwidth() (auspex.instruments.agilent.AgilentN9010A method), 67
get_voltage() (auspex.instruments.ami.AMI430 method), 71
get_voltage() (auspex.instruments.bbn.SpectrumAnalyzer method), 75
get_voltage_limit() (auspex.instruments.ami.AMI430 method), 71
get_voltage_max() (auspex.instruments.ami.AMI430 method), 71
get_voltage_min() (auspex.instruments.ami.AMI430 method), 71
get_voltage_unit() (auspex.instruments.agilent.HP33120A method), 69
get_waveform_frequency() (auspex.instruments.bbn.APS method), 73
get_waveform_frequency() (auspex.instruments.bbn.APS2 method), 74
groupname (auspex.filters.io.WriteToFile attribute), 56

H

high_voltage (auspex.instruments.agilent.Agilent33220A attribute), 60
high_voltage (auspex.instruments.agilent.Agilent33500B attribute), 63
HP33120A (class in auspex.instruments.agilent), 68

I

IDN() (auspex.instruments.interface.VisaInterface method), 76
idn_string (auspex.instruments.prologix.PrologixSocketResource attribute), 77
IF_FREQ (auspex.instruments.bbn.SpectrumAnalyzer attribute), 75
in_jupyter() (in module auspex.log), 82
inductance (auspex.instruments.ami.AMI430 attribute), 71
init_filters() (auspex.filters.channelizer.Channelizer method), 53
InputConnector (class in auspex.stream), 84
Instrument (class in auspex.instruments.instrument), 76
instrument_type (auspex.instruments.agilent.AgilentN5183A attribute), 66
instrument_type (auspex.instruments.agilent.AgilentN9010A attribute), 67
instrument_type (auspex.instruments.alazar.AlazarATS9870 attribute), 70
instrument_type (auspex.instruments.ami.AMI430 attribute), 72
instrument_type (auspex.instruments.bbn.APS attribute), 73
instrument_type (auspex.instruments.bbn.APS2 attribute), 74
instrument_type (auspex.instruments.bbn.DigitalAttenuator attribute), 75
instrument_type (auspex.instruments.bbn.SpectrumAnalyzer attribute), 75
instrument_type (auspex.instruments.bbn.TDM attribute), 75
instrument_type (auspex.instruments.X6.X6 attribute), 80
Interface (class in auspex.instruments.interface), 76
IntParameter (class in auspex.parameter), 82
is_adaptive() (auspex.stream.DataStreamDescriptor method),

84
is_adaptive() (auspex.sweep.Sweeper method), 85
isnotebook() (in module auspex.config), 81

K

kernel (auspex.filters.integrator.KernelIntegrator attribute), 56
KernelIntegrator (class in auspex.filters.integrator), 55

L

last_data_axis() (auspex.stream.DataStreamDescriptor method), 84
load (auspex.instruments.agilent.Agilent33500B attribute), 63
load (auspex.instruments.agilent.HP33120A attribute), 69
load_data() (in module auspex.analysis.helpers), 52
load_db() (in module auspex.config), 81
load_resistance (auspex.instruments.agilent.Agilent33220A attribute), 60
load_waveform() (auspex.instruments.bbn.APS method), 73
load_waveform() (auspex.instruments.bbn.APS2 method), 74
load_waveform_from_file() (auspex.instruments.bbn.APS method), 73
logistic_fidelity() (auspex.filters.singleshot.SingleShotMeasurement method), 58
logistic_regression (auspex.filters.singleshot.SingleShotMeasurement attribute), 58
LorentzFit (class in auspex.analysis.fits), 50
low_voltage (auspex.instruments.agilent.Agilent33220A attribute), 60
low_voltage (auspex.instruments.agilent.Agilent33500B attribute), 63

M

main() (auspex.filters.elementwise.ElementwiseFilter method), 54
main() (auspex.filters.filter.Filter method), 55
main() (auspex.filters.io.DataBuffer method), 56
make_plots() (auspex.analysis.fits.Auspex2DFit method), 49
make_plots() (auspex.analysis.fits.AuspexFit method), 50
ManualPlotter (class in auspex.filters.plot), 57
marker1_amplitude (auspex.instruments.agilent.AgilentN9010A attribute), 67
marker1_position (auspex.instruments.agilent.AgilentN9010A attribute), 67
marker_to_center() (auspex.instruments.agilent.AgilentN9010A method), 67
marker_X (auspex.instruments.agilent.AgilentN9010A attribute), 67
marker_Y (auspex.instruments.agilent.AgilentN9010A attribute), 67
MeshPlotter (class in auspex.filters.plot), 57
mixer_correction_matrix (auspex.instruments.bbn.APS attribute), 73
mixer_correction_matrix (auspex.instruments.bbn.APS2 attribute), 74
mod (auspex.instruments.agilent.AgilentN5183A attribute), 66
mode (auspex.instruments.agilent.AgilentN9010A attribute), 67
model() (auspex.analysis.fits.Auspex2DFit method), 49
model() (auspex.analysis.fits.AuspexFit method), 50
MultiGaussianFit (class in auspex.analysis.fits), 50

N

new_dataset() (auspex.data_format.AuspexDataContainer method), 81
new_group() (auspex.data_format.AuspexDataContainer method), 81
noise_marker() (auspex.instruments.agilent.AgilentN9010A method), 68
normalize_buffer_data() (in module auspex.analysis.helpers), 52
normalize_data() (in module auspex.analysis.helpers), 52
NUM_CHANNELS (auspex.instruments.bbn.DigitalAttenuator attribute), 75
num_data_axis_points() (auspex.stream.DataStreamDescriptor method), 84
num_dims() (auspex.stream.DataStreamDescriptor method), 84
num_new_points_through_axis() (auspex.stream.DataStreamDescriptor method), 84
num_points() (auspex.stream.DataAxis method), 83
num_points() (auspex.stream.DataStream method), 83
num_points() (auspex.stream.DataStreamDescriptor method), 84
num_points() (auspex.stream.InputConnector method), 84

num_points() (*auspex.stream.OutputConnector method*), 84
num_points_through_axis() (*auspex.stream.DataStreamDescriptor method*), 84
num_sweep_points (*auspex.instruments.agilent.AgilentN9010A attribute*), 68
number_averages (*auspex.instruments.X6.X6 attribute*), 80
number_segments (*auspex.instruments.X6.X6 attribute*), 80
number_waveforms (*auspex.instruments.X6.X6 attribute*), 80

O

offset (*auspex.instruments.agilent.HP33120A attribute*), 69
on_done() (*auspex.filters.filter.Filter method*), 55
on_done() (*auspex.filters.plot.MeshPlotter method*), 58
on_done() (*auspex.filters.plot.Plotter method*), 57
ONOFF_VALUES (*auspex.instruments.agilent.Agilent34970A attribute*), 65
OPC() (*auspex.instruments.interface.VisaInterface method*), 76
open_all() (*auspex.data_format.AuspexDataContainer method*), 81
open_data() (*in module auspex.analysis.helpers*), 52
open_dataset() (*auspex.data_format.AuspexDataContainer method*), 82
operation() (*auspex.filters.correlator.Correlator method*), 54
operation() (*auspex.filters.elementwise.ElementwiseFilter method*), 54
optimal_integration_time (*auspex.filters.singleshot.SingleShotMeasurement attribute*), 58
output (*auspex.instruments.agilent.Agilent33220A attribute*), 60
output (*auspex.instruments.agilent.Agilent33500B attribute*), 63
output (*auspex.instruments.agilent.AgilentN5183A attribute*), 66
output_gated (*auspex.instruments.agilent.Agilent33500B attribute*), 63
output_sync (*auspex.instruments.agilent.Agilent33220A attribute*), 60
output_sync (*auspex.instruments.agilent.Agilent33500B attribute*), 63
output_trigger (*auspex.instruments.agilent.Agilent33500B attribute*), 63
output_trigger_slope (*auspex.instruments.agilent.Agilent33500B attribute*), 63
output_trigger_source (*auspex.instruments.agilent.Agilent33500B attribute*), 63
output_units (*auspex.instruments.agilent.Agilent33220A attribute*), 60
output_units (*auspex.instruments.agilent.Agilent33500B attribute*), 63
OutputConnector (*class in auspex.stream*), 84

P

Parameter (*class in auspex.parameter*), 82
ParameterGroup (*class in auspex.parameter*), 82
Passthrough (*class in auspex.filters.debug*), 54
pause() (*auspex.instruments.ami.AMI430 method*), 72
peak_amplitude() (*auspex.instruments.bbn.SpectrumAnalyzer method*), 75
peak_search() (*auspex.instruments.agilent.AgilentN9010A method*), 68
percent_complete() (*auspex.stream.DataStream method*), 83
persistent_switch (*auspex.instruments.ami.AMI430 attribute*), 72
phase (*auspex.instruments.agilent.AgilentN5183A attribute*), 66
phase_skew (*auspex.instruments.bbn.APS2 attribute*), 74
phys_channel (*auspex.instruments.alazar.AlazarChannel attribute*), 70
PLC_VALUES (*auspex.instruments.agilent.Agilent34970A attribute*), 65
plot_dims (*auspex.filters.plot.Plotter attribute*), 57
plot_mode (*auspex.filters.plot.MeshPlotter attribute*), 58
plot_mode (*auspex.filters.plot.Plotter attribute*), 57
Plotter (*class in auspex.filters.plot*), 57
pn_carrier_freq (*auspex.instruments.agilent.AgilentN9010A attribute*), 68
pn_offset_start (*auspex.instruments.agilent.AgilentN9010A attribute*), 68
pn_offset_stop (*auspex.instruments.agilent.AgilentN9010A at-*

tribute), 68
points_with_metadata() (auspex.stream.DataAxis method), 83
polarity (auspex.instruments.agilent.Agilent33220A attribute), 60
polarity (auspex.instruments.agilent.Agilent33500B attribute), 63
pop() (auspex.stream.DataStream method), 83
pop_axis() (auspex.stream.DataStreamDescriptor method), 84
ports (auspex.instruments.agilent.AgilentE8363C attribute), 66
ports (auspex.instruments.agilent.AgilentN5230A attribute), 70
power (auspex.instruments.agilent.AgilentN5183A attribute), 66
Print (class in auspex.filters.debug), 54
process_data() (auspex.filters.channelizer.Channelizer method), 53
process_data() (auspex.filters.debug.Passthrough method), 54
process_data() (auspex.filters.debug.Print method), 54
process_data() (auspex.filters.framer.Framer method), 55
process_data() (auspex.filters.integrator.KernelIntegrator method), 56
process_data() (auspex.filters.io.DataBuffer method), 56
process_data() (auspex.filters.io.WriteToFile method), 56
process_data() (auspex.filters.plot.Plotter method), 57
process_data() (auspex.filters.singleshot.SingleShotMeasurement method), 58
process_direct() (auspex.filters.plot.MeshPlotter method), 58
process_message() (auspex.filters.filter.Filter method), 55
PrologixInterface (class in auspex.instruments.interface), 76
PrologixSocketResource (class in auspex.instruments.prologix), 77
pulse_dcyc (auspex.instruments.agilent.Agilent33220A attribute), 60
pulse_dcyc (auspex.instruments.agilent.Agilent33500B attribute), 63
pulse_edge (auspex.instruments.agilent.Agilent33220A attribute), 60
pulse_edge (auspex.instruments.agilent.Agilent33500B attribute), 63
pulse_period (auspex.instruments.agilent.Agilent33220A attribute), 60
pulse_period (auspex.instruments.agilent.Agilent33500B attribute), 63
pulse_width (auspex.instruments.agilent.Agilent33220A attribute), 60
pulse_width (auspex.instruments.agilent.Agilent33500B attribute), 63
push() (auspex.parameter.Parameter method), 82
push() (auspex.parameter.ParameterGroup method), 83
push() (auspex.stream.DataStream method), 83
push() (auspex.stream.OutputConnector method), 84
push() (auspex.stream.SweepAxis method), 85
push_event() (auspex.stream.DataStream method), 83
push_event() (auspex.stream.OutputConnector method), 84
push_resource_usage() (auspex.filters.filter.Filter method), 55
push_to_all() (auspex.filters.filter.Filter method), 55

Q

QuadraticFit (class in auspex.analysis.fits), 50
query() (auspex.instruments.interface.Interface method), 76
query() (auspex.instruments.interface.VisaInterface method), 76
query() (auspex.instruments.prologix.PrologixSocketResource method), 78
query_ascii_values() (auspex.instruments.interface.VisaInterface method), 77
query_ascii_values() (auspex.instruments.prologix.PrologixSocketResource method), 78
query_binary_values() (auspex.instruments.interface.VisaInterface method), 77
query_binary_values() (auspex.instruments.prologix.PrologixSocketResource method), 78

R

r_lists() (auspex.instruments.agilent.Agilent34970A method), 65
ramp() (auspex.instruments.ami.AMI430 method), 72
ramp_down() (auspex.instruments.ami.AMI430 method), 72
ramp_num_segments (auspex.instruments.ami.AMI430 attribute), 72

ramp_rate_units (*auspex.instruments.ami.AMI430 attribute*), 72
ramp_symmetry (*auspex.instruments.agilent.Agilent33220A attribute*), 60
ramp_symmetry (*auspex.instruments.agilent.Agilent33500B attribute*), 63
ramp_up() (*auspex.instruments.ami.AMI430 method*), 72
ramping_state (*auspex.instruments.ami.AMI430 attribute*), 72
RAMPING_STATES (*auspex.instruments.ami.AMI430 attribute*), 70
read() (*auspex.instruments.agilent.Agilent34970A method*), 65
read() (*auspex.instruments.interface.VisaInterface method*), 77
read() (*auspex.instruments.prologix.PrologixSocketResource method*), 78
read_bytes() (*auspex.instruments.interface.VisaInterface method*), 77
read_raw() (*auspex.instruments.interface.VisaInterface method*), 77
read_raw() (*auspex.instruments.prologix.PrologixSocketResource method*), 79
read_termination (*auspex.instruments.prologix.PrologixSocketResource attribute*), 77
receive_data() (*auspex.instruments.alazar.AlazarATS9870 method*), 70
receive_data() (*auspex.instruments.X6.X6 method*), 80
record_length (*auspex.instruments.X6.X6 attribute*), 80
reference (*auspex.instruments.agilent.AgilentN5183A attribute*), 66
reference (*auspex.instruments.X6.X6 attribute*), 80
repeat_mode (*auspex.instruments.bbn.APS attribute*), 73
RES_VALUES (*auspex.instruments.agilent.Agilent34970A attribute*), 65
reset() (*auspex.stream.DataAxis method*), 83
reset() (*auspex.stream.DataStream method*), 83
reset() (*auspex.stream.DataStreamDescriptor method*), 84
resistance_range (*auspex.instruments.agilent.Agilent34970A attribute*), 65
resistance_resolution (*auspex.instruments.agilent.Agilent34970A attribute*), 65
resistance_wire (*auspex.instruments.agilent.Agilent34970A attribute*), 65
resistance_zcomp (*auspex.instruments.agilent.Agilent34970A attribute*), 65
resolution_bandwidth (*auspex.instruments.agilent.AgilentN9010A attribute*), 68
restart_sweep() (*auspex.instruments.agilent.AgilentN9010A method*), 68
RST() (*auspex.instruments.interface.VisaInterface method*), 76
run() (*auspex.filters.filter.Filter method*), 55
run_mode (*auspex.instruments.bbn.APS attribute*), 73
run_mode (*auspex.instruments.bbn.APS2 attribute*), 74

S

sampling_rate (*auspex.instruments.bbn.APS attribute*), 73
sampling_rate (*auspex.instruments.bbn.APS2 attribute*), 74
save_kernel (*auspex.filters.singleshot.SingleShotMeasurement attribute*), 58
scan() (*auspex.instruments.agilent.Agilent34970A method*), 65
scanlist (*auspex.instruments.agilent.Agilent34970A attribute*), 65
self_check() (*auspex.instruments.agilent.Agilent33500B.Segment method*), 61
send() (*auspex.filters.plot.ManualPlotter method*), 57
send() (*auspex.filters.plot.MeshPlotter method*), 58
send() (*auspex.filters.plot.Plotter method*), 57
sequence (*auspex.instruments.agilent.Agilent33500B attribute*), 63
sequence_file (*auspex.instruments.bbn.APS attribute*), 73
sequence_file (*auspex.instruments.bbn.APS2 attribute*), 74
set_absorber() (*auspex.instruments.ami.AMI430 method*), 72
set_advance_source() (*auspex.instruments.agilent.Agilent34970A method*), 65
set_alc() (*auspex.instruments.agilent.AgilentN5183A method*), 66
set_all() (*auspex.instruments.agilent.AgilentN5183A method*), 66
set_all() (*auspex.instruments.alazar.AlazarChannel method*), 70
set_amp_factor() (*auspex.instruments.bbn.APS2 method*), 74

set_amplitude() (auspex.instruments.agilent.Agilent33220A method), 60
set_amplitude() (auspex.instruments.agilent.Agilent33500B method), 63
set_amplitude() (auspex.instruments.agilent.HP33120A method), 69
set_amplitude() (auspex.instruments.bbn.APS method), 73
set_amplitude() (auspex.instruments.bbn.APS2 method), 74
set_arb_advance() (auspex.instruments.agilent.Agilent33500B method), 63
set_arb_amplitude() (auspex.instruments.agilent.Agilent33500B method), 63
set_arb_frequency() (auspex.instruments.agilent.Agilent33500B method), 63
set_arb_sample() (auspex.instruments.agilent.Agilent33500B method), 63
set_arb_waveform() (auspex.instruments.agilent.Agilent33500B method), 63
set_attenuation() (auspex.instruments.bbn.DigitalAttenuator method), 75
set_auto_range() (auspex.instruments.agilent.Agilent33220A method), 60
set_auto_range() (auspex.instruments.agilent.Agilent33500B method), 63
set_averaging_count() (auspex.instruments.agilent.AgilentN9010A method), 68
set_burst_cycles() (auspex.instruments.agilent.Agilent33220A method), 60
set_burst_cycles() (auspex.instruments.agilent.Agilent33500B method), 63
set_burst_cycles() (auspex.instruments.agilent.HP33120A method), 69
set_burst_mode() (auspex.instruments.agilent.Agilent33220A method), 60
set_burst_mode() (auspex.instruments.agilent.Agilent33500B method), 63
set_burst_source() (auspex.instruments.agilent.HP33120A method), 69
set_burst_state() (auspex.instruments.agilent.Agilent33220A method), 60
set_burst_state() (auspex.instruments.agilent.Agilent33500B method), 63
set_burst_state() (auspex.instruments.agilent.HP33120A method), 69
set_by_receiver() (auspex.instruments.alazar.AlazarChannel method), 70
set_by_receiver_channel() (auspex.instruments.X6.X6Channel method), 80
set_coil_const() (auspex.instruments.ami.AMI430 method), 72
set_current_limit() (auspex.instruments.ami.AMI430 method), 72
set_current_rating() (auspex.instruments.ami.AMI430 method), 72
set_current_target() (auspex.instruments.ami.AMI430 method), 72
set_data() (auspex.filters.plot.ManualPlotter method), 57
set_dc_offset() (auspex.instruments.agilent.Agilent33220A method), 60
set_dc_offset() (auspex.instruments.agilent.Agilent33500B method), 63
set_descriptor() (auspex.stream.DataStream method), 83
set_descriptor() (auspex.stream.OutputConnector method), 84
set_dmm() (auspex.instruments.agilent.Agilent34970A method), 65
set_done() (auspex.filters.plot.ManualPlotter method), 57
set_done() (auspex.filters.plot.Plotter method), 57
set_duty_cycle() (auspex.instruments.agilent.HP33120A method), 69
set_field() (auspex.instruments.ami.AMI430 method), 72
set_field_target() (auspex.instruments.ami.AMI430 method), 72
set_field_units() (auspex.instruments.ami.AMI430 method), 72
set_fpga_temperature() (aus-

pex.instruments.bbn.APS2 attribute), 74
set_frequency() (*auspex.instruments.agilent.Agilent33220A method*), 60
set_frequency() (*auspex.instruments.agilent.Agilent33500B method*), 63
set_frequency() (*auspex.instruments.agilent.AgilentN5183A method*), 66
set_frequency() (*auspex.instruments.agilent.HP33120A method*), 69
set_frequency_center() (*auspex.instruments.agilent.AgilentN9010A method*), 68
set_frequency_span() (*auspex.instruments.agilent.AgilentN9010A method*), 68
set_frequency_start() (*auspex.instruments.agilent.AgilentN9010A method*), 68
set_frequency_stop() (*auspex.instruments.agilent.AgilentN9010A method*), 68
set_function() (*auspex.instruments.agilent.Agilent33220A method*), 60
set_function() (*auspex.instruments.agilent.Agilent33500B method*), 64
set_function() (*auspex.instruments.agilent.HP33120A method*), 69
set_high_voltage() (*auspex.instruments.agilent.Agilent33220A method*), 60
set_high_voltage() (*auspex.instruments.agilent.Agilent33500B method*), 64
set_infinite_load() (*auspex.instruments.agilent.Agilent33500B method*), 64
set_load() (*auspex.instruments.agilent.Agilent33500B method*), 64
set_load() (*auspex.instruments.agilent.HP33120A method*), 69
set_load_resistance() (*auspex.instruments.agilent.Agilent33220A method*), 60
set_low_voltage() (*auspex.instruments.agilent.Agilent33220A method*), 60
set_low_voltage() (*auspex.instruments.agilent.Agilent33500B method*), 64
set_marker1_amplitude() (*auspex.instruments.agilent.AgilentN9010A method*), 68
set_marker1_position() (*auspex.instruments.agilent.AgilentN9010A method*), 68
set_mixer_amplitude_imbalance() (*auspex.instruments.bbn.APS method*), 73
set_mixer_correction_matrix (*auspex.instruments.bbn.APS attribute*), 73
set_mixer_correction_matrix() (*auspex.instruments.bbn.APS2 method*), 74
set_mixer_phase_skew() (*auspex.instruments.bbn.APS method*), 73
set_mod() (*auspex.instruments.agilent.AgilentN5183A method*), 66
set_mode() (*auspex.instruments.agilent.AgilentN9010A method*), 68
set_num_sweep_points() (*auspex.instruments.agilent.AgilentN9010A method*), 68
set_offset() (*auspex.instruments.agilent.HP33120A method*), 69
set_offset() (*auspex.instruments.bbn.APS method*), 73
set_offset() (*auspex.instruments.bbn.APS2 method*), 74
set_output() (*auspex.instruments.agilent.Agilent33220A method*), 60
set_output() (*auspex.instruments.agilent.Agilent33500B method*), 64
set_output() (*auspex.instruments.agilent.AgilentN5183A method*), 66
set_output_gated() (*auspex.instruments.agilent.Agilent33500B method*), 64
set_output_sync() (*auspex.instruments.agilent.Agilent33220A method*), 60
set_output_sync() (*auspex.instruments.agilent.Agilent33500B method*), 64
set_output_trigger() (*auspex.instruments.agilent.Agilent33500B method*), 64
set_output_trigger_slope() (*auspex.instruments.agilent.Agilent33500B method*), 64

set_output_trigger_source() (*auspex.instruments.agilent.Agilent33500B method*), 64
set_output_units() (*auspex.instruments.agilent.Agilent33220A method*), 60
set_output_units() (*auspex.instruments.agilent.Agilent33500B method*), 64
set_persistent_switch() (*auspex.instruments.ami.AMI430 method*), 72
set_phase() (*auspex.instruments.agilent.AgilentN5183A method*), 66
set_phase_skew() (*auspex.instruments.bbn.APS2 method*), 74
set_pn_carrier_freq() (*auspex.instruments.agilent.AgilentN9010A method*), 68
set_pn_offset_start() (*auspex.instruments.agilent.AgilentN9010A method*), 68
set_pn_offset_stop() (*auspex.instruments.agilent.AgilentN9010A method*), 68
set_polarity() (*auspex.instruments.agilent.Agilent33220A method*), 60
set_polarity() (*auspex.instruments.agilent.Agilent33500B method*), 64
set_power() (*auspex.instruments.agilent.AgilentN5183A method*), 66
set_pulse_dcyc() (*auspex.instruments.agilent.Agilent33220A method*), 60
set_pulse_dcyc() (*auspex.instruments.agilent.Agilent33500B method*), 64
set_pulse_edge() (*auspex.instruments.agilent.Agilent33220A method*), 60
set_pulse_edge() (*auspex.instruments.agilent.Agilent33500B method*), 64
set_pulse_period() (*auspex.instruments.agilent.Agilent33220A method*), 61
set_pulse_period() (*auspex.instruments.agilent.Agilent33500B method*), 64
set_pulse_width() (*auspex.instruments.agilent.Agilent33220A method*), 61
set_pulse_width() (*auspex.instruments.agilent.Agilent33500B method*), 64
set_quit() (*auspex.filters.plot.ManualPlotter method*), 57
set_quit() (*auspex.filters.plot.Plotter method*), 57
set_ramp_num_segments() (*auspex.instruments.ami.AMI430 method*), 72
set_ramp_rate_units() (*auspex.instruments.ami.AMI430 method*), 72
set_ramp_symmetry() (*auspex.instruments.agilent.Agilent33220A method*), 61
set_ramp_symmetry() (*auspex.instruments.agilent.Agilent33500B method*), 64
set_repeat_mode() (*auspex.instruments.bbn.APS method*), 73
set_resolution_bandwidth() (*auspex.instruments.agilent.AgilentN9010A method*), 68
set_run_mode() (*auspex.instruments.bbn.APS method*), 73
set_run_mode() (*auspex.instruments.bbn.APS2 method*), 74
set_sampling_rate() (*auspex.instruments.bbn.APS method*), 73
set_sampling_rate() (*auspex.instruments.bbn.APS2 method*), 75
set_sequence() (*auspex.instruments.agilent.Agilent33500B method*), 64
set_sequence_file() (*auspex.instruments.bbn.APS method*), 73
set_sequence_file() (*auspex.instruments.bbn.APS2 method*), 75
set_stability() (*auspex.instruments.ami.AMI430 method*), 72
set_sweep_time() (*auspex.instruments.agilent.AgilentN9010A method*), 68
set_sync_mode() (*auspex.instruments.agilent.Agilent33500B method*), 64
set_sync_polarity() (*auspex.instruments.agilent.Agilent33500B method*), 64
set_sync_source() (*auspex.instruments.agilent.Agilent33500B method*), 64
set_threshold (*auspex.filters.singleshot.SingleShotMeasurement attribute*), 58
set_trigger_count() (*auspex.instruments.agilent.Agilent34970A*

method), 65
set_trigger_interval() (*auspex.instruments.bbn.APS method*), 73
set_trigger_interval() (*auspex.instruments.bbn.APS2 method*), 75
set_trigger_slope() (*auspex.instruments.agilent.Agilent33220A method*), 61
set_trigger_slope() (*auspex.instruments.agilent.Agilent33500B method*), 64
set_trigger_source() (*auspex.instruments.agilent.Agilent33220A method*), 61
set_trigger_source() (*auspex.instruments.agilent.Agilent33500B method*), 64
set_trigger_source() (*auspex.instruments.agilent.Agilent34970A method*), 65
set_trigger_source() (*auspex.instruments.bbn.APS method*), 73
set_trigger_source() (*auspex.instruments.bbn.APS2 method*), 75
set_trigger_timer() (*auspex.instruments.agilent.Agilent34970A method*), 65
set_video_auto() (*auspex.instruments.agilent.AgilentN9010A method*), 68
set_video_bandwidth() (*auspex.instruments.agilent.AgilentN9010A method*), 68
set_voltage_limit() (*auspex.instruments.ami.AMI430 method*), 72
set_voltage_unit() (*auspex.instruments.agilent.HP33120A method*), 69
set_waveform_frequency (*auspex.instruments.bbn.APS attribute*), 74
set_waveform_frequency() (*auspex.instruments.bbn.APS2 method*), 75
shutdown() (*auspex.filters.filter.Filter method*), 55
simple_kernel (*auspex.filters.integrator.KernelIntegrator attribute*), 56
SingleShotMeasurement (*class in auspex.filters.singleshot*), 58
sink (*auspex.filters.channelizer.Channelizer attribute*), 53
sink (*auspex.filters.correlator.Correlator attribute*), 54
sink (*auspex.filters.debug.Passthrough attribute*), 54
sink (*auspex.filters.debug.Print attribute*), 54
sink (*auspex.filters.elementwise.ElementwiseFilter attribute*), 54
sink (*auspex.filters.framer.Framer attribute*), 55
sink (*auspex.filters.integrator.KernelIntegrator attribute*), 56
sink (*auspex.filters.io.DataBuffer attribute*), 56
sink (*auspex.filters.io.WriteToFile attribute*), 56
sink (*auspex.filters.plot.MeshPlotter attribute*), 58
sink (*auspex.filters.plot.Plotter attribute*), 57
sink (*auspex.filters.singleshot.SingleShotMeasurement attribute*), 58
source (*auspex.filters.channelizer.Channelizer attribute*), 53
source (*auspex.filters.correlator.Correlator attribute*), 54
source (*auspex.filters.debug.Passthrough attribute*), 54
source (*auspex.filters.elementwise.ElementwiseFilter attribute*), 54
source (*auspex.filters.framer.Framer attribute*), 55
source (*auspex.filters.integrator.KernelIntegrator attribute*), 56
source (*auspex.filters.singleshot.SingleShotMeasurement attribute*), 58
SpectrumAnalyzer (*class in auspex.instruments.bbn*), 75
spew_fake_data() (*auspex.instruments.alazar.AlazarATS9870 method*), 70
spew_fake_data() (*auspex.instruments.X6.X6 method*), 80
SRE() (*auspex.instruments.interface.VisaInterface method*), 76
stability (*auspex.instruments.ami.AMI430 attribute*), 72
start() (*auspex.filters.plot.ManualPlotter method*), 57
STB() (*auspex.instruments.interface.VisaInterface method*), 76
stop() (*auspex.filters.plot.ManualPlotter method*), 57
stop() (*auspex.instruments.alazar.AlazarATS9870 method*), 70
supply_type (*auspex.instruments.ami.AMI430 attribute*), 72
SUPPLY_TYPES (*auspex.instruments.ami.AMI430 attribute*), 70
sweep_time (*auspex.instruments.agilent.AgilentN9010A attribute*), 68
SweepAxis (*class in auspex.stream*), 84
Sweeper (*class in auspex.sweep*), 85
swept_parameters() (*auspex.sweep.Sweeper method*), 85
sync_mode (*auspex.instruments.agilent.Agilent33500B attribute*), 64
sync_polarity (*auspex.instruments.agilent.Agilent33500B attribute*), 64

sync_source (*auspex.instruments.agilent.Agilent33500B attribute*), 64

T

TDM (*class in auspex.instruments.bbn*), 75
timeout (*auspex.instruments.prologix.PrologixSocketResource attribute*), 77, 79
title (*auspex.analysis.fits.Auspex2DFit attribute*), 49
title (*auspex.analysis.fits.AuspexFit attribute*), 50
title (*auspex.analysis.fits.GaussianFit attribute*), 50
title (*auspex.analysis.fits.LorentzFit attribute*), 50
title (*auspex.analysis.fits.MultiGaussianFit attribute*), 50
title (*auspex.analysis.fits.QuadraticFit attribute*), 50
TOLERANCE (*auspex.filters.singleshot.SingleShotMeasurement attribute*), 58
trigger() (*auspex.instruments.agilent.Agilent33220A method*), 61
trigger() (*auspex.instruments.agilent.Agilent33500B method*), 64
trigger() (*auspex.instruments.bbn.APS method*), 74
trigger() (*auspex.instruments.bbn.APS2 method*), 75
trigger_count (*auspex.instruments.agilent.Agilent34970A attribute*), 65
trigger_interval (*auspex.instruments.bbn.APS attribute*), 74
trigger_interval (*auspex.instruments.bbn.APS2 attribute*), 75
trigger_slope (*auspex.instruments.agilent.Agilent33220A attribute*), 61
trigger_slope (*auspex.instruments.agilent.Agilent33500B attribute*), 64
trigger_source (*auspex.instruments.agilent.Agilent33220A attribute*), 61
trigger_source (*auspex.instruments.agilent.Agilent33500B attribute*), 64
trigger_source (*auspex.instruments.agilent.Agilent34970A attribute*), 66
trigger_source (*auspex.instruments.bbn.APS attribute*), 74
trigger_source (*auspex.instruments.bbn.APS2 attribute*), 75
trigger_timer (*auspex.instruments.agilent.Agilent34970A attribute*), 66
TRIGSOUR_VALUES (*auspex.instruments.agilent.Agilent34970A attribute*), 65
TST() (*auspex.instruments.interface.VisaInterface method*), 76
tuple_width() (*auspex.stream.DataAxis method*), 83
tuple_width() (*auspex.stream.DataStreamDescriptor method*), 84
tuples() (*auspex.stream.DataStreamDescriptor method*), 84

U

unit() (*auspex.filters.correlator.Correlator method*), 54
unit() (*auspex.filters.elementwise.ElementwiseFilter method*), 54
update() (*auspex.filters.plot.Plotter method*), 57
update() (*auspex.instruments.agilent.Agilent33500B.Segment method*), 61
update() (*auspex.stream.SweepAxis method*), 85
update() (*auspex.sweep.Sweeper method*), 85
update_descriptors() (*auspex.filters.channelizer.Channelizer method*), 53
update_descriptors() (*auspex.filters.elementwise.ElementwiseFilter method*), 54
update_descriptors() (*auspex.filters.filter.Filter method*), 55
update_descriptors() (*auspex.filters.integrator.KernelIntegrator method*), 56
update_descriptors() (*auspex.filters.plot.MeshPlotter method*), 58
update_descriptors() (*auspex.filters.plot.Plotter method*), 57
update_descriptors() (*auspex.filters.singleshot.SingleShotMeasurement method*), 58
update_descriptors() (*auspex.stream.InputConnector method*), 84
update_descriptors() (*auspex.stream.OutputConnector method*), 84
update_references() (*auspex.filters.channelizer.Channelizer method*), 53
upload_sequence() (*auspex.instruments.agilent.Agilent33500B method*), 64
upload_waveform() (*auspex.instruments.agilent.Agilent33500B method*), 64
upload_waveform() (*auspex.instruments.agilent.HP33120A method*), 69

upload_waveform_binary() (auspex.instruments.agilent.Agilent33500B method), 64

V

value (auspex.parameter.BoolParameter attribute), 82
value (auspex.parameter.FloatParameter attribute), 82
value (auspex.parameter.IntParameter attribute), 82
value (auspex.parameter.Parameter attribute), 82
value (auspex.parameter.ParameterGroup attribute), 83
value() (auspex.instruments.interface.VisaInterface method), 77
values() (auspex.instruments.interface.Interface method), 76
values() (auspex.instruments.interface.VisaInterface method), 77
video_auto (auspex.instruments.agilent.AgilentN9010A attribute), 68
video_bandwidth (auspex.instruments.agilent.AgilentN9010A attribute), 68
VisaInterface (class in auspex.instruments.interface), 76
voltage (auspex.instruments.ami.AMI430 attribute), 72
voltage (auspex.instruments.bbn.SpectrumAnalyzer attribute), 75
voltage_limit (auspex.instruments.ami.AMI430 attribute), 72
voltage_max (auspex.instruments.ami.AMI430 attribute), 72
voltage_min (auspex.instruments.ami.AMI430 attribute), 72
voltage_unit (auspex.instruments.agilent.HP33120A attribute), 69

W

WAI() (auspex.instruments.interface.VisaInterface method), 76
wait_for_acquisition() (auspex.instruments.alazar.AlazarATS9870 method), 70
wait_for_acquisition() (auspex.instruments.X6.X6 method), 80
waveform_frequency (auspex.instruments.bbn.APS attribute), 74
waveform_frequency (auspex.instruments.bbn.APS2 attribute), 75
write() (auspex.instruments.interface.Interface method), 76
write() (auspex.instruments.interface.VisaInterface method), 77
write() (auspex.instruments.prologix.PrologixSocketResource method), 79
write_ascii_values() (auspex.instruments.prologix.PrologixSocketResource method), 79
write_binary_values() (auspex.instruments.interface.VisaInterface method), 77
write_binary_values() (auspex.instruments.prologix.PrologixSocketResource method), 79
write_raw() (auspex.instruments.interface.VisaInterface method), 77
write_raw() (auspex.instruments.prologix.PrologixSocketResource method), 79
write_termination (auspex.instruments.prologix.PrologixSocketResource attribute), 77
WriteToFile (class in auspex.filters.io), 56

X

X6 (class in auspex.instruments.X6), 80
X6Channel (class in auspex.instruments.X6), 80
xlabel (auspex.analysis.fits.Auspex2DFit attribute), 49
xlabel (auspex.analysis.fits.AuspexFit attribute), 49, 50
xlabel (auspex.analysis.fits.GaussianFit attribute), 50
xlabel (auspex.analysis.fits.LorentzFit attribute), 50
xlabel (auspex.analysis.fits.MultiGaussianFit attribute), 50
xlabel (auspex.analysis.fits.QuadraticFit attribute), 51

Y

ylabel (auspex.analysis.fits.Auspex2DFit attribute), 49
ylabel (auspex.analysis.fits.AuspexFit attribute), 50
ylabel (auspex.analysis.fits.GaussianFit attribute), 50
ylabel (auspex.analysis.fits.LorentzFit attribute), 50
ylabel (auspex.analysis.fits.MultiGaussianFit attribute), 50
ylabel (auspex.analysis.fits.QuadraticFit attribute), 51

Z

zero() (auspex.instruments.ami.AMI430 method), 72
zero_mean (auspex.filters.singleshot.SingleShotMeasurement attribute), 58