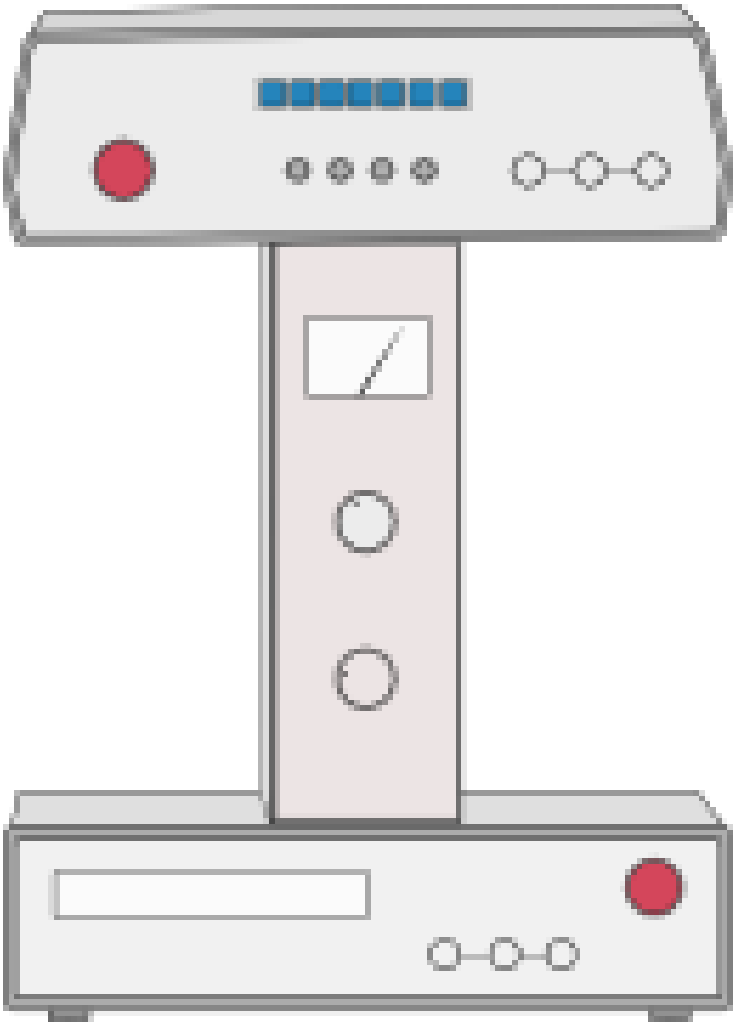
Instrumental Documentation

Release 0.2

Mabuchi Lab

December 16, 2015

Contents



Instrumental is a Python-based library for controlling lab hardware like cameras, DAQs, oscilloscopes, spectrometers, and more. It has high-level drivers for instruments from NI, Tektronix, Thorlabs, PCO, Photometrics, Burleigh, and others.

Instrumental's goal is to make common tasks simple to perform, while still providing the flexibility to perform complex tasks with relative ease. It also makes it easy to mess around with instruments in the shell. For example, to list the available instruments and open one of them:

```
>>> from instrumental import instrument, list_instruments
>>> insts = list_instruments()
>>> insts
[<TEKTRONIX 'DPO4034'>, <TEKTRONIX 'MSO4034'>, <NIDAQ 'Dev1'>]
>>> daq = instrument(insts[2])
>>> daq
<instrumental.drivers.daq.ni.NIDAQ at 0xb61...>
```

If you're going to be using an instrument repeatedly, save it for later:

```
>>> daq.save_instrument('myDAQ')
```

Then you can simply open it by name:

```
>>> daq = instrument('myDAQ')
```

Check out Working with Instruments for more detailed info.

Instrumental also bundles in some additional support code, including:

- Plotting and curve fitting utilities
- Tools for working with optics, including Gaussian beams and ABCD matrices
- Utilities for acquiring and organizing data

Instrumental makes use of NumPy, SciPy, Matplotlib, and Pint, a Python units library. It optionally uses PyVISA/VISA and other drivers for interfacing with lab equipment.

To download Instrumental or browse its source, see our GitHub page.

Note: Instrumental is currently still under heavy development, so its interfaces are subject to change. Contributions are greatly appreciated, see the Developer's Guide for more info.

CHAPTER 1

User Guide

1.1 Installation

1.1.1 Brief Install Instructions

If you already have NumPy/SciPy/Matplotlib/pip installed, installing Instrumental is simple. First install Pint:

```
$ pip install pint
```

Now download and extract a zip of Instrumental from the Github page or clone it using git. Now install:

```
$ cd /path/to/Instrumental
$ python setup.py install
$ python post_install.py
```

`post_install.py` installs a config file, so you only have to run it the first time you install Instrumental.

1.1.2 Detailed Install Instructions

Python Sci-Comp Stack

To install the standard scientific computing stack, I recommend using Anaconda. Download the appropriate installer from the download page and run it to install Anaconda. The default installation will include NumPy, SciPy, and Matplotlib as well as lots of other useful stuff.

Pint

Next, install Pint for units support:

```
$ pip install pint
```

For more information, or to get a more recent version, check out the Pint install page.

Instrumental

If you're using git, you can clone the Instrumental repository to get the source code. If you don't know git or don't want to set up a local repo yet, you can just download a zip file by clicking the 'Download ZIP' button on the right hand side of the Instrumental Github page. Unzip the code wherever you'd like, then open a command prompt to that directory and run:

```
$ python setup.py install
$ python post_install.py
```

to install Instrumental to your Python site-packages directory and add a default configuration to your config directory. You're all set! Now go check out some of the examples in the `examples` directory contained in the files you downloaded!

Optional Driver Libraries

VISA

To operate devices that communicate using VISA (e.g. Tektronix scopes) you will need:

1. an implementation of VISA, and
2. a Python interface layer called PyVISA

There are various implementations of VISA available, but two I know of are TekVISA (from Tektronix) and NI-VISA (from National Instruments). I would recommend NI-VISA, though either one should work fine. Installers for each can be downloaded from the NI or Tektronix websites, though you'll have to create a free account.

Once you've installed VISA, install PyVISA by running:

```
$ pip install pyvisa
```

on the command line. As a quick test PyVISA has installed correctly, open a python interpreter and run:

```
>>> import visa
>>> rm = visa.ResourceManager()
>>> rm.list_resources()
```

More info about PyVISA, including more detailed install-related information can be found here.

Thorlabs DCx Cameras

To operate Thorlabs DCx cameras, you'll need the drivers from Thorlabs under the "Software and Support" tab. Run the .exe installer which, among other things, will install the .dll shared libraries somewhere in your PATH (hopefully). Currently the code only looks for the 64-bit driver, so if you're on a 32-bit system I may need to work with you to fix this.

NI DAQs

Currently, NI-DAQmx support requires PyDAQmx. It can be installed via pip:

```
$ pip install PyDAQmx
```

You will also need to have NI-DAQmx installed. You can find the installer on the National Instruments website.

1.2 Overview

1.2.1 Drivers

The `drivers` subpackage's purpose is to provide relatively high-level 'drivers' for interfacing with lab equipment. Currently it (fully or partially) supports:

- Tektronix TDS300 and MSO/DPO4000 series oscilloscopes
- Tektronix AFG3000 series arbitrary function generators
- Thorlabs DCx class USB cameras
- NI DAQmx compatible DAQ devices
- Attocube ECC100 controller and associated translation stages and goniometers

Drivers are planned for:

- Thorlabs PM100x series optical power meters
- Newport 1830-C optical power meter
- Thorlabs APT motion control systems (e.g. T-Cube motor controllers)

It should be pretty easy to write drivers for other VISA-compatible devices via PyVISA. Driver submissions are greatly appreciated!

1.2.2 Plotting

The `plotting` module provides or aims to provide

- Unit-aware plotting functions as a drop-in replacement for pyplot
- Easy slider-plots

1.2.3 Fitting

The `fitting` module is a good place for curating 'standard' fitting tools for common cases like

- Triple-lorentzian cavity scans
- Ringdown traces (exponential decay)

It should also provide optional unit-awareness.

1.2.4 Optics

The `optics` module is a repository for useful optics code. Currently it focuses on using numerical ABCD matrices to manipulate and visualize paraxial gaussian beams. For example, it can be used to easily specify the elements of an optical cavity, solve for the supported modes, and plot the tangential and sagittal spot size throughout the beam path.

1.2.5 Tools

The `tools` module is used for full-fledged scripts and programs that may make use of all of the other modules above. A good example would be a script that pulls a trace from the scope, auto-fits a ringdown curve, and saves both the raw data and fit parameters to files in a well-organized directory structure.

1.3 Quickstart

1.3.1 Using Instruments

Much of Instrumental's utility is in its ability to communicate with lab equipment. Our goal is to make interfacing with equipment as simple and powerful as it should be.

Connecting to a VISA instrument is easy:

```
>>> from instrumental import instrument
>>> scope = instrument(visa_address='TCPIP::0.0.0.1::INSTR')
>>> scope
<instrumental.drivers.scopes.tektronix.TDS_3000 object at 0x7f...>
```

It can be even easier if you've already set up an alias in your `instrumental.conf` file:

```
>>> scope = instrument('myScopeAlias')
<instrumental.drivers.scopes.tektronix.TDS_3000 object at 0x7f...>
```

For more detailed info, see Working with Instruments. Now we can use our new scope object to grab some data:

```
>>> x, y = scope.get_data()
>>> x
<Quantity([-9.99800000e-07, ..., 1.00000000e-06], 'second')>
>>> y
<Quantity([1.13007813, ..., -0.04835938], 'volt')>
```

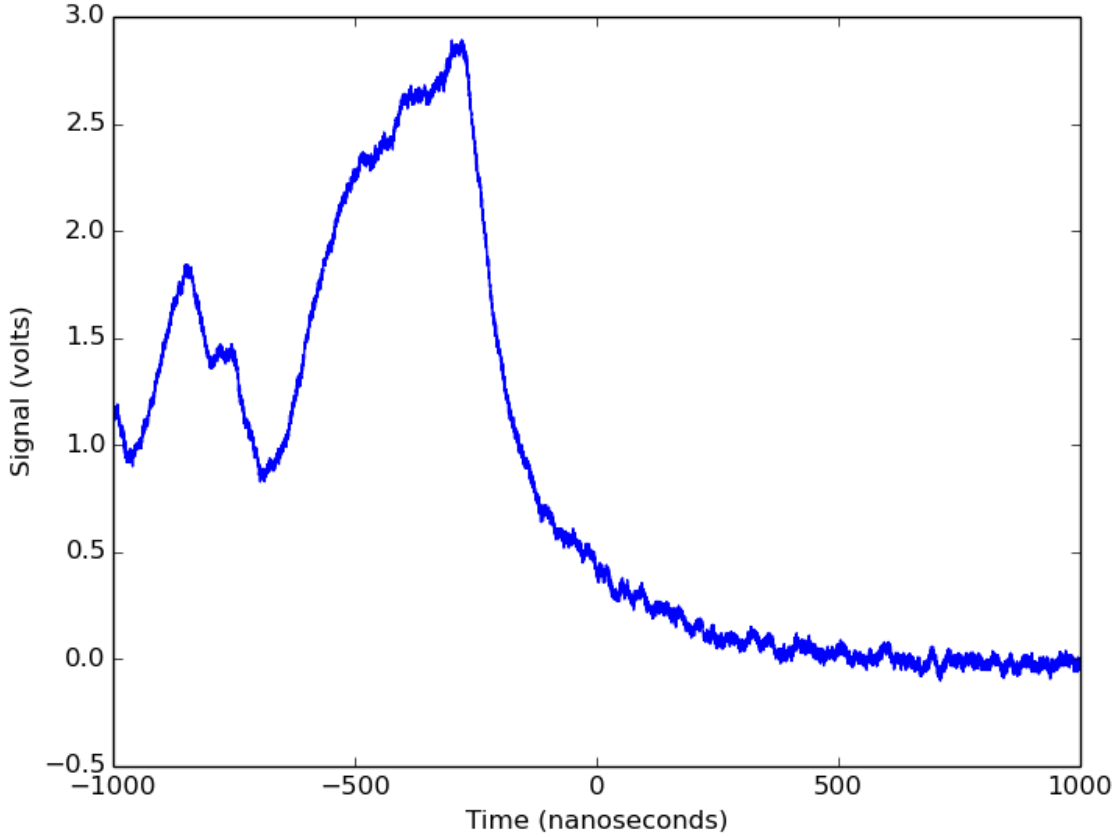
Notice that our data already has units! By default, the scope grabs data from its first channel. We can grab data from the other channel by using:

```
>>> x, y = scope.get_data(channel=2)
```

Now let's plot our data:

```
>>> import instrumental.plotting as plt
>>> plt.plot(x.to('ns'), y)
>>> plt.ylabel('Signal')
>>> plt.xlabel('Time')
>>> plt.show()
```

This gives us



But... where did those unit labels come from? Instrumental's wrapped versions of `xlabel` and `ylabel` add them automatically so you don't have to.

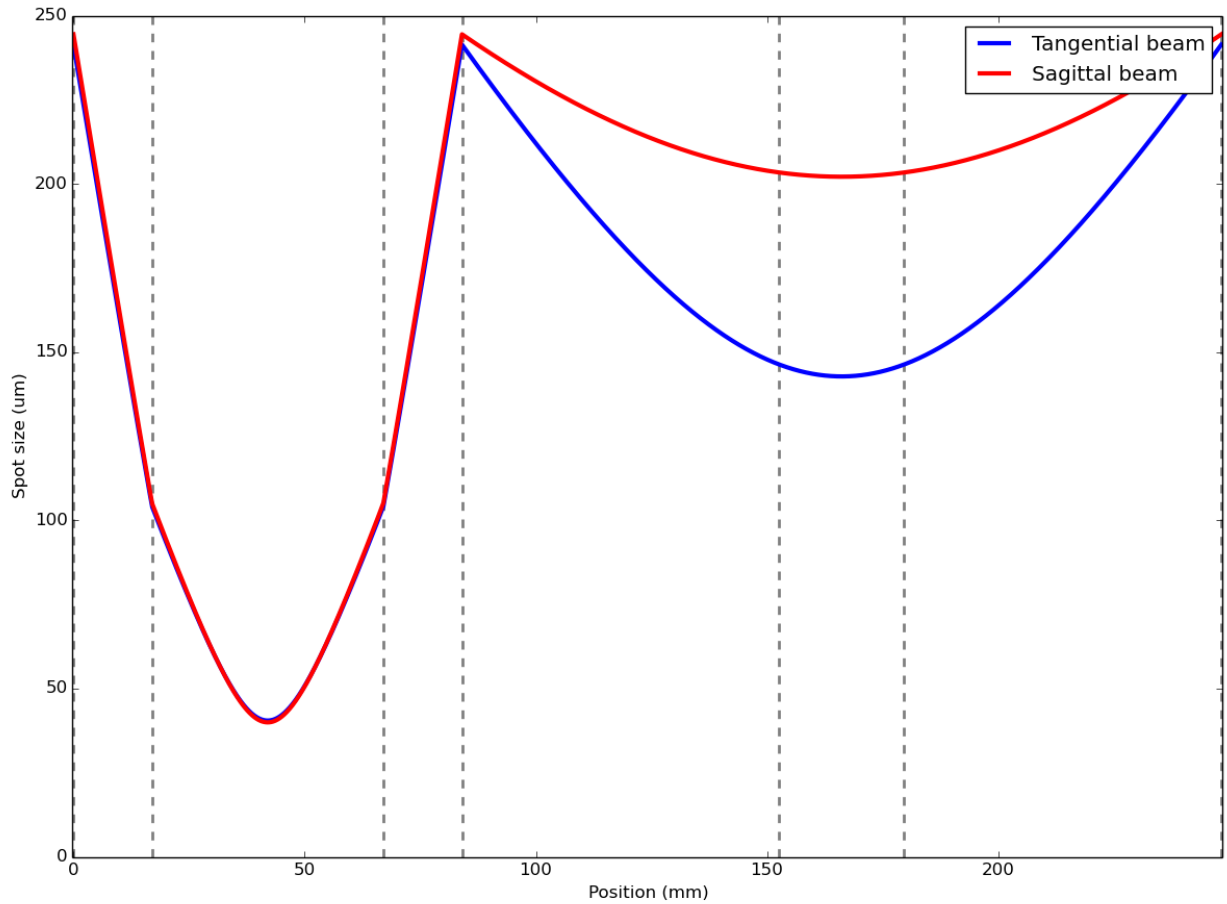
1.3.2 Solving for and Plotting a Cavity Mode

A common use case for working with ray transfer matrices is solving for a cavity mode and looking at the mode's spatial profile. Instrumental makes this easy. Here's a short script that constructs a bowtie cavity with a crystal inside, solves for its tangential and sagittal modes, and plots them:

```
from instrumental import (plotting as plt, Space, Mirror, Interface,
                          find_cavity_modes, plot_profile)
# Indices of refraction
n0, nc = 1, 2.18

# Create cavity elements
cavity_elems = [Mirror(R='50mm', aoi='18deg'), Space('1.7cm'),
                Interface(n0, nc), Space('5cm', nc),
                Interface(nc, n0), Space('1.7cm'),
                Mirror(R='50mm', aoi='18deg'), Space('6.86cm'),
                Mirror(), Space('2.7cm'),
                Mirror(), Space('6.86cm')]

# Find tangential and sagittal cavity modes
```

```
qt_r, qs_r = find_cavity_modes(cavity_elems)

# Beam profile inside the cavity
plot_profile(qt_r, qs_r, '1064nm', cavity_elems, cyclical=True)
plt.legend()
plt.show()
```

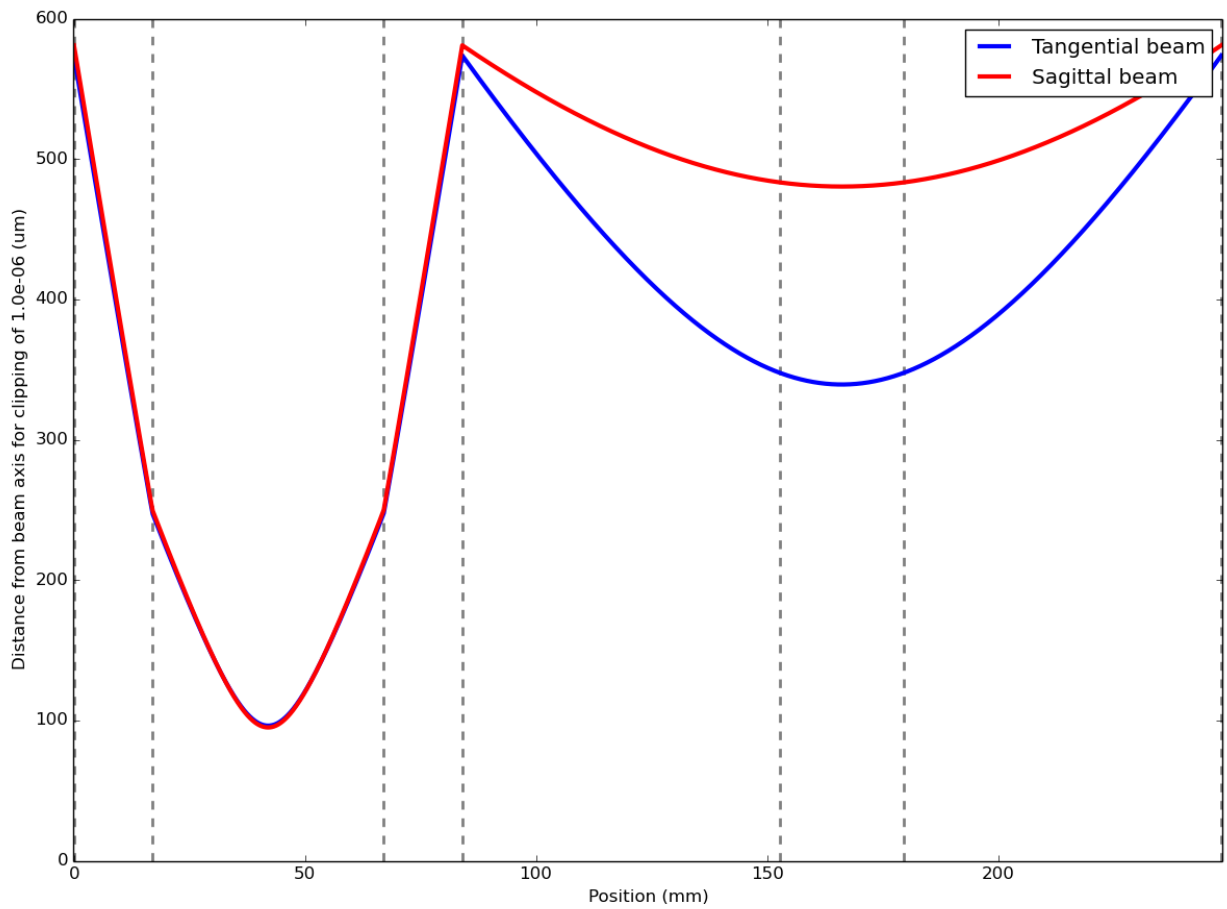
This will produce a plot that looks something like



We might also be interested in any losses from clipping, or aperture effects. To look at this, we can simply change the plotting line to:

```
plot_profile(qt_r, qs_r, '1064nm', cavity_elems, cyclical=True,
             clipping=1e-6)
```

This will now plot the radial distance at which power losses from clipping become 1 part per million, i.e. 1e-6.



This example makes good use of Instrumental's unit-friendliness. We're using all sorts of length scales here, from nanometers to centimeters, all handled simply and explicitly. Are some of your lengths in inches? No problem! No more wondering "...is this variable the wavelength in nanometers, or in meters?"

1.4 Working with Instruments

1.4.1 Getting Started

Instrumental tries to make it easy to find and open all the instruments available to your computer. This is primarily accomplished using `list_instruments()` and `instrument()`:

```
>>> from instrumental import instrument, list_instruments
>>> insts = list_instruments()
>>> insts
[<TEKTRONIX 'DPO4034'>, <TEKTRONIX 'MSO4034'>, <NIDAQ 'Dev1'>]
```

You can then use the output of `list_instruments()` to open the instrument you want:

```
>>> daq = instrument(insts[2])
>>> daq
<instrumental.drivers.daq.ni.NIDAQ at 0xb61...>
```

If you're going to be using an instrument repeatedly, save it for later:

```
>>> daq.save_instrument('myDAQ')
```

Then you can simply open it by name:

```
>>> daq = instrument('myDAQ')
```

An Even Quicker Way

Here's a shortcut for opening an instrument that means you don't have to assign the instrument list to a variable, or even know how to count–just use part of the instrument's string:

```
>>> list_instruments()
[<TEKTRONIX 'DPO4034'>, <TEKTRONIX 'MSO4034'>, <NIDAQ 'Dev1'>]
>>> instrument('DPO')  # Opens the <TEKTRONIX 'DPO4034'>
>>> instrument('NIDAQ')  # Opens the <NIDAQ 'Dev1'>
```

This will work as long as the string you use isn't saved as an instrument alias. If you use a string that matches multiple instruments, it just picks the first in the list.

Remote Instruments

You can even control instruments that are attached to a remote computer:

```
>>> list_instruments(server='192.168.1.10')
```

This lists only the instruments located on the remote machine, not any local ones.

The remote PC must be running as an Instrumental server (and its firewall configured to allow inbound connections on this port). To do this, run the script `tools/server.py` that comes packaged with Instrumental. The client needs to specify the server's IP address (or hostname), and port number (if differs from the default of 28265). Alternatively, you may save an alias for this server in the `[servers]` section of you `instrumental.conf` file. See *Saved Instruments* for more information about `instrumental.conf`. Then you can list the remote instruments like this:

```
>>> list_instruments(server='myServer')
```

You can then open your instrument using `instrument()` as usual, but now you'll get a `RemoteInstrument`, which you can control just like a regular `Instrument`.

1.4.2 How Does it All Work?

Listing Instruments

What exactly is `list_instruments()` doing? Basically it walks through all the driver modules, trying to import them one by one. If import fails (perhaps the DLL isn't available because the user doesn't have this instrument), that module is skipped. Each module is responsible for returning a list of its available instruments, e.g. the `drivers.daqs.ni` module returns a list of all the NI DAQs that are accessible. `list_instruments()` combines all these instruments into one big list and returns it.

There's an unfortunate side-effect of this: if a module fails to import due to a bug, the exception is caught and ignored, so you don't get a helpful traceback. To diagnose issues with a driver module, you can import the module directly:

```
>>> import instrumental.drivers.daq.ni
```

or enable logging before calling `list_instruments()`:

```
>>> import logging
>>> logging.basicConfig(level=logging.INFO)
```

`list_instruments()` doesn't open instruments directly, but instead returns a list of dictionary-like elements that contain info about how to open the instrument. For example, for our DAQ:

```
>>> dict(insts[2])
{u'nidaq_devname': u'Dev1'}
```

This tells us that the daq is uniquely identified by the parameter `nidaq_devname`. So, we could also open it with keyword arguments:

```
>>> instrument(nidaq_devname='Dev1')
<instrumental.drivers.daq.ni.NIDAQ at 0xb69...>
```

or a dictionary:

```
>>> instrument({'nidaq_devname': 'Dev1'})
<instrumental.drivers.daq.ni.NIDAQ at 0xb62...>
```

Behind the scenes, `instrument()` uses the keywords to figure out what type of instrument you're talking about, and what class should be instantiated.

Saved Instruments

Opening instruments using `list_instruments()` is really helpful when you're messing around in the shell and don't quite know what info you need yet, or you're checking what devices are available to you. But if you've found your device and want to write a script that reuses it constantly, it's nice to have it saved under an alias, which you can do easily with `save_instrument()` as we showed above.

When you do this, the instrument's info gets saved in your `instrumental.conf` config file. To find where the file is located on your system, run:

```
>>> from instrumental.conf import data_dir
>>> data_dir
u'C:\\Users\\Lab\\AppData\\Local\\MabuchiLab\\Instrumental'
```

To save your instrument for repeated use, add its parameters to the `[instruments]` section of `instrumental.conf`. For our DAQ, that would look like:

```
# NI-DAQ device
myDAQ = {'nidaq_devname': 'Dev1'}
```

This gives our DAQ the alias `myDAQ`, which can then be used to open it easily:

```
>>> instrument('myDAQ')
<instrumental.drivers.daq.ni.NIDAQ at 0xb71...>
```

The default version of `instrumental.conf` also provides some commented-out example entries to help make things clear.

1.5 API Documentation

1.5.1 Drivers

Instrumental drivers allow you to control and read data from various hardware devices.

Some devices (e.g. Thorlabs cameras) have drivers that act as wrappers to their drivers' C bindings, using `ctypes` or `cffi`. Others (e.g. Tektronix scopes and AFGs) utilize VISA and `PyVISA`, its Python wrapper. `PyVISA` requires a local installation of the VISA library (e.g. NI-VISA) to interface with connected devices.

DAQs

Create `DAQ` objects using *`instrument()`*.

Supported Models

NI DAQs This module has been developed using an NI USB-6221 – the code should generally work for all DAQmx boards, but I'm sure there are plenty of compatibility bugs just waiting for you wonderful users to find and fix.

First, make sure you have NI's DAQmx software installed. Once that's set, you'll need PyDAQmx, a basic Python interface to DAQmx. You can get it via pip:

```
pip install PyDAQmx
```

The `NIDAQ` class lets you interact with your board and all its various inputs and outputs in a fairly simple way. Let's say you've hooked up digital I/O P1.0 to analog input AI0, and your analog out AO1 to analog input AI1:

```
>>> from instrumental.drivers.daq.ni import NIDAQ, list_instruments
>>> list_instruments()
[<NIDAQ 'Dev0'>]
>>> daq = NIDAQ('Dev0')
>>> daq.ai0.read()
<Quantity(0.0154385786803, 'volt')>
>>> daq.port1[0].write(True)
>>> daq.ai0.read()
<Quantity(5.04241962841, 'volt')>
>>> daq.ao1.write('2.1V')
>>> daq.ai1.read()
<Quantity(2.10033320744, 'volt')>
```

Now let's try using digital input. Assume P1.1 is attached to P1.2:

```
>>> daq.port1[1].write(False)
>>> daq.port1[2].read()
False
>>> daq.port1[1].write(True)
>>> daq.port1[2].read()
True
```

Let's read and write more than one bit at a time. To write to multiple lines simultaneously, pass an unsigned int to `write()`. The line with the lowest index corresponds to the lowest bit, and so on. If you read from multiple lines, read() returns an int. Connect P1.0-3 to P1.4-7:

```
>>> daq.port1[0:3].write(5)   # 0101 = decimal 5
>>> daq.port1[4:7].read()   # Note that the last index IS included
5
>>> daq.port1[7:4].read()   # This flips the ordering of the bits
10                          # 1010 = decimal 10
>>> daq.port1[0].write(False)   # Zero the ones bit individually
>>> daq.port1[4:7].read()   # 0100 = decimal 4
4
```

You can also read and write arrays of buffered data. Use the same `read()` and `write()` methods, just include your timing info (and pass in the data as an array if writing). When writing, you must provide either `freq` or `fsamp`, and

may provide either `duration` or `reps` to specify for how long the waveform is output. For example, there are many ways to output the same sinusoid:

```
>>> from instrumental import u
>>> from numpy import pi, sin, linspace
>>> data = sin( 2*pi * linspace(0, 1, 100, endpoint=False) )*5*u.V + 5*u.V
>>> daq.ao0.write(data, duration='1s', freq='500Hz')
>>> daq.ao0.write(data, duration='1s', fsamp='50kHz')
>>> daq.ao0.write(data, reps=500, freq='500Hz')
>>> daq.ao0.write(data, reps=500, fsamp='50kHz')
```

Note the use of `endpoint=False` in `linspace`. This ensures we don't repeat the start/end point (0V) of our sine waveform when outputting more than one period.

All this stuff is great for simple tasks, but sometimes you may want to perform input and output on multiple channels simultaneously. To accomplish this we need to use Tasks.

Note: Tasks in the `ni` module are similar, but not the same as Tasks in DAQmx (and PyDAQmx). Our Tasks allow you to quickly and easily perform simultaneous input and output with one Task without the hassle of having to create multiple and hook their timing and triggers up.

Here's an example of how to perform simultaneous input and output:

```
>>> from instrumental.drivers.daq.ni import NIDAQ, Task
>>> from instrumental import u
>>> from numpy import linspace
>>> daq = NIDAQ('Dev0')
>>> task = Task(daq.ao0, daq.ai0)
>>> task.set_timing(duration='1s', fsamp='10Hz')
>>> write_data = {'ao0': linspace(0, 9, 10) * u.V}
>>> task.run(write_data)
{u'ai0': <Quantity([  1.00000094e+01   1.89578724e-04   9.99485542e-01   2.00007917e+00
   3.00034866e+00   3.99964556e+00   4.99991698e+00   5.99954114e+00
   6.99981625e+00   7.99976941e+00], 'volt')>,
 u't': <Quantity([ 0.   0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9], 'second')>}
```

As you can see, we create a dict as input to the `run()` method. Its keys are the names of the input channels, and its values are the corresponding array Quantities that we want to write. Similarly, the `run()` returns a dict that contains the input that was read. This dict also contains the time data under key 't'. Note that the read and write happen concurrently, so each voltage read has not yet moved to its new setpoint.

Module Reference Driver module for NI-DAQmx-supported hardware.

class `instrumental.drivers.daq.ni.`**`AnalogIn`**(*dev*, *chan_name*)

Methods

read([duration, fsamp, n_samples])	Read one or more analog input samples.

`__init__`(*dev*, *chan_name*)

`read`(*duration=None*, *fsamp=None*, *n_samples=None*)

Read one or more analog input samples.

By default, reads and returns a single sample. If two of `duration`, `fsamp`, and `n_samples` are given, an array of samples is read and returned.

Parameters **duration** : Quantity

How long to read from the analog input, specified as a Quantity. Use with `fsamp` or `n_samples`.

fsamp : Quantity

The sample frequency, specified as a Quantity. Use with `duration` or `n_samples`.

n_samples : int

The number of samples to read. Use with `duration` or `fsamp`.

Returns **data** : scalar or array Quantity

The data that was read from analog input.

class `instrumental.drivers.daq.ni.`**`AnalogOut`**(*dev*, *chan_name*)

Methods

`write`(data[, duration, reps, fsamp, freq, ...])	Write a value or array to the analog output.

`__init__`(*dev*, *chan_name*)

`write`(*data*, *duration=None*, *reps=None*, *fsamp=None*, *freq=None*, *onboard=True*)

Write a value or array to the analog output.

If `data` is a scalar value, it is written to output and the function returns immediately. If `data` is an array of values, a buffered write is performed, writing each value in sequence at the rate determined by `duration` and `fsamp` or `freq`. You must specify either `fsamp` or `freq`.

When writing an array, this function blocks until the output sequence has completed.

Parameters **data** : scalar or array Quantity

The value or values to output, passed in Volt-compatible units.

duration : Quantity, optional

Used when writing arrays of data. This is how long the entirety of the output lasts, specified as a second-compatible Quantity. If `duration` is longer than a single period of data, the waveform will repeat. Use either this or `reps`, not both. If neither is given, waveform is output once.

reps : int or float, optional

Used when writing arrays of data. This is how many times the waveform is repeated. Use either this or `duration`, not both. If neither is given, waveform is output once.

fsamp: Quantity, optional

Used when writing arrays of data. This is the sample frequency, specified as a Hz-compatible Quantity. Use either this or `freq`, not both.

freq : Quantity, optional

Used when writing arrays of data. This is the frequency of the *overall waveform*, specified as a Hz-compatible Quantity. Use either this or `fsamp`, not both.

onboard : bool, optional

Use only onboard memory. Defaults to True. If False, all data will be continually buffered from the PC memory, even if it is only repeating a small number of samples many times.

class `instrumental.drivers.daq.ni.`**`Channel`**

class `instrumental.drivers.daq.ni.`**`Counter`**(*dev*, *chan_name*)

Methods

`output_pulses`(freq[, duration, reps, ...])	Generate digital pulses using the counter.

`__init__`(*dev*, *chan_name*)

`output_pulses`(*freq*, *duration=None*, *reps=None*, *idle_high=False*, *delay=None*, *duty_cycle=0.5*)

Generate digital pulses using the counter.

Outputs digital pulses with a given frequency and duty cycle.

This function blocks until the output sequence has completed.

Parameters **freq** : Quantity

This is the frequency of the pulses, specified as a Hz-compatible Quantity.

duration : Quantity, optional

How long the entirety of the output lasts, specified as a second-compatible Quantity. Use either this or `reps`, not both. If neither is given, only one pulse is generated.

reps : int, optional

How many pulses to generate. Use either this or `duration`, not both. If neither is given, only one pulse is generated.

idle_high : bool, optional

Whether the resting state is considered high or low. Idles low by default.

delay : Quantity, optional

How long to wait before generating the first pulse, specified as a second-compatible Quantity. Defaults to zero.

duty_cycle : float, optional

The width of the pulse divided by the pulse period. The default is a 50% duty cycle.

class `instrumental.drivers.daq.ni.`**`NIDAQ`**(*dev_name*)

Methods

`create_task`()	
`get_AI_channels`()	
`get_AI_max_range`()	Returns the min and max voltage of the widest AI range
`get_AI_ranges`()	
`get_AO_channels`()	
`get_AO_max_range`()	Returns the min and max voltage of the widest AO range

Continued on next page

Table 1.4 – continued from previous page

get_AO_ranges()
get_CI_channels()
get_CO_channels()
get_DI_lines()
get_DI_ports()
get_DO_lines()
get_DO_ports()
get_chassis_num()
get_product_type()
get_serial()
get_slot_num()
get_terminals()

__init__(*dev_name*)
Constructor for an NIDAQ object. End users should not use this directly, and should instead use *instrument()*

create_task()

get_AI_channels()

get_AI_max_range()
Returns the min and max voltage of the widest AI range

get_AI_ranges()

get_AO_channels()

get_AO_max_range()
Returns the min and max voltage of the widest AO range

get_AO_ranges()

get_CI_channels()

get_CO_channels()

get_DI_lines()

get_DI_ports()

get_DO_lines()

get_DO_ports()

get_chassis_num()

get_product_type()

get_serial()

get_slot_num()

get_terminals()

class instrumental.drivers.daq.ni.**Task**(**args*)
Note that true DAQmx tasks can only include one type of channel (e.g. AI). To run multiple synchronized reads/writes, we need to make one task for each type, then use the same sample clock for each.

Methods

run([write_data])	
set_timing([duration, fsamp, n_samples, ...])	

__init__(**args*)

Creates a task that uses the given channels.

Each arg can either be a Channel or a tuple of (Channel, name_str)

run(*write_data=None*)

set_timing(*duration=None*, *fsamp=None*, *n_samples=None*, *mode=u'finite'*, *clock=u''*, *rising=True*)

class instrumental.drivers.daq.ni.**VirtualDigitalChannel**(*dev*, *line_pairs*)

Methods

as_input()	
as_output()	
read()	
write(value)	Write a value to the digital output channel
write_sequence(data[, duration, reps, ...])	Write an array of samples to the digital output channel

__init__(*dev*, *line_pairs*)

as_input()

as_output()

read()

write(*value*)

Write a value to the digital output channel

Parameters value : int or bool

An int representing the digital values to write. The lowest bit of the int is written to the first digital line, the second to the second, and so forth. For a single-line DO channel, can be a bool.

write_sequence(*data*, *duration=None*, *reps=None*, *fsamp=None*, *freq=None*, *onboard=True*)

Write an array of samples to the digital output channel

Outputs a buffered digital waveform, writing each value in sequence at the rate determined by `duration` and `fsamp` or `freq`. You must specify either `fsamp` or `freq`.

This function blocks until the output sequence has completed.

Parameters data : array or list of ints or bools

The sequence of samples to output. For a single-line DO channel, samples can be bools.

duration : Quantity, optional

How long the entirety of the output lasts, specified as a second-compatible Quantity. If `duration` is longer than a single period of data, the waveform will repeat. Use either this or `reps`, not both. If neither is given, waveform is output once.

reps : int or float, optional

How many times the waveform is repeated. Use either this or `duration`, not both. If neither is given, waveform is output once.

fsamp: Quantity, optional

This is the sample frequency, specified as a Hz-compatible Quantity. Use either this or `freq`, not both.

freq : Quantity, optional

This is the frequency of the *overall waveform*, specified as a Hz-compatible Quantity. Use either this or `fsamp`, not both.

onboard : bool, optional

Use only onboard memory. Defaults to True. If False, all data will be continually buffered from the PC memory, even if it is only repeating a small number of samples many times.

`instrumental.drivers.daq.ni.`**`list_instruments`**`()`

Cameras

Create *`Camera`* objects using *`instrument()`*.

Supported Models

PCO Cameras This module is for controlling PCO cameras that use the PCO.camera SDK. Note that not all PCO cameras use this SDK, e.g. older Pixelfly cameras have their own SDK.

Installation This module requires the PCO SDK and the `cffi` package.

You should install the PCO SDK provided on PCO's website. Specifically, this module requires `SC2_Cam.dll` to be available in your PATH, as well as any interface-specific DLLs. Firewire requires `SC2_1394.dll`, and each type of Camera Link grabber requires its own DLL, e.g. `sc2_cl_me4.dll` for a Silicon Software microEnable IV grabber card.

Module Reference

Pixelfly Cameras This module is for controlling PCO Pixelfly cameras.

Installation This module requires the Pixelfly SDK and the `cffi` package.

You should install the Pixelfly SDK provided on PCO's website. Specifically, this module requires `pf_cam.dll` to be available in your PATH.

Module Reference

UC480 (Thorlabs DCx) Cameras This module is for controlling Thorlabs DCx cameras. You should install the corresponding drivers that can be found on the Thorlabs website. Specifically, this module requires either 'uc480.dll' or 'uc480_64.dll', depending on your system. The driver library must be visible to python, so you may need to add it to your PATH or copy it to your Windows system32 directory.

Module Reference

Generic Camera Interface

Package containing a driver module/class for each supported camera type.

class `instrumental.drivers.cameras.`**`Camera`**

A generic camera device.

Camera driver internals can often be quite different; however, Instrumental defines a few basic concepts that all camera drivers should have.

There are two basic modes: *finite* and *continuous*.

In *finite* mode, a camera performs a capture sequence, returning one or more images all at once, when the sequence is finished.

In *continuous* or *live* mode, the camera continuously retrieves images until it is manually stopped. This mode can be used e.g. to make a GUI that looks at a live view of the camera. The process looks like this:

```
>>> cam.start_live_video()
>>> while not_done():
>>>     frame_ready = cam.wait_for_frame()
>>>     if frame_ready:
>>>         arr = cam.latest_frame()
>>>         do_stuff_with(arr)
>>> cam.stop_live_video()
```

Attributes

Methods

get_captured_image([timeout, copy])	Get the image array(s) from the last capture sequence
grab_image([timeouts, copy])	Perform a capture and return the resulting image array(s)
latest_frame([copy])	Get the latest image frame in live mode
start_capture(**kwds)	Start a capture sequence and return immediately
start_live_video(**kwds)	Start live video mode
stop_live_video()	Stop live video mode
wait_for_frame([timeout])	Wait until the next frame is ready (in live mode)

`get_captured_image`(*timeout='1s', copy=True*)

Get the image array(s) from the last capture sequence

Returns an image numpy array (or tuple of arrays for a multi-exposure sequence). The array has shape *(height, width)* for grayscale images, and *(height, width, 3)* for RGB images. Typically the dtype will be `uint8`, or sometimes `uint16` in the case of 16-bit monochromatic cameras.

Parameters **timeout** : Quantity([time]) or None, optional

Max time to wait for wait for the image data to be ready. If *None*, will block forever. If timeout is exceeded, a TimeoutError will be raised.

copy : bool, optional

Whether to copy the image memory or directly reference the underlying buffer. It is recommended to use *True* (the default) unless you know what you're doing.

grab_image(*timeouts='1s'*, *copy=True*, ***kwds*)

Perform a capture and return the resulting image array(s)

This is essentially a convenience function that calls `start_capture()` then `image_array()`. See `image_array()` for information about the returned array(s).

Parameters timeouts : Quantity([time]) or None, optional

Max time to wait for wait for the image data to be ready. If *None*, will block forever. If timeout is exceeded, a TimeoutError will be raised.

copy : bool, optional

Whether to copy the image memory or directly reference the underlying buffer. It is recommended to use *True* (the default) unless you know what you're doing.

You can specify other parameters of the capture as keyword arguments. These include:

Other Parameters n_frames : int

Number of exposures in the sequence

vbin : int

Vertical binning

hbin : int

Horizontal binning

exposure_time : Quantity([time])

Duration of each exposure

width : int

Width of the ROI

height : int

Height of the ROI

cx : int

X-axis center of the ROI

cy : int

Y-axis center of the ROI

left : int

Left edge of the ROI

right : int

Right edge of the ROI

top : int

Top edge of the ROI

bot : int

Bottom edge of the ROI

latest_frame(*copy=True*)

Get the latest image frame in live mode

Returns the image array received on the most recent successful call to `wait_for_frame()`.

Parameters **copy** : bool, optional

Whether to copy the image memory or directly reference the underlying buffer. It is recommended to use *True* (the default) unless you know what you're doing.

start_capture(***kwds*)

Start a capture sequence and return immediately

Depending on your camera-specific shutter/trigger settings, this will either start the exposure immediately or ready the camera to start on an explicit (hardware or software) trigger.

It can be useful to invoke `capture()` and `image_array()` explicitly if you expect the capture sequence to take a long time and you'd like to perform some operations while you wait for the camera:

```
>>> cam.capture()
>>> do_other_useful_stuff()
>>> arr = cam.image_array()
```

See `grab_image()` for the set of available kwds.

start_live_video(***kwds*)

Start live video mode

Once live video mode has been started, images will automatically and continuously be acquired. You can check if the next frame is ready by using `wait_for_frame()`, and access the most recent image's data with `image_array()`.

See `grab_image()` for the set of available kwds.

stop_live_video()

Stop live video mode

wait_for_frame(*timeout=None*)

Wait until the next frame is ready (in live mode)

Blocks and returns True once the next frame is ready, False if the timeout was reached. Using a timeout of 0 simply polls to see if the next frame is ready.

Parameters **timeout** : Quantity([time]), optional

How long to wait for wait for the image data to be ready. If *None* (the default), will block forever.

Returns **frame_ready** : bool

True if the next frame is ready, *False* if the timeout was reached.

height

A decorator indicating abstract properties.

Requires that the metaclass is ABCMeta or derived from it. A class that has a metaclass derived from ABCMeta cannot be instantiated unless all of its abstract properties are overridden. The abstract properties can be called using any of the normal 'super' call mechanisms.

Usage:

class C: __metaclass__ = ABCMeta @abstractproperty def my_abstract_property(self):

...

This defines a read-only property; you can also define a read-write abstract property using the 'long' form of property declaration:

class C: __metaclass__ = ABCMeta def getx(self): ... def setx(self, value): ... x = abstractproperty(getx, setx)

max_height

A decorator indicating abstract properties.

Requires that the metaclass is ABCMeta or derived from it. A class that has a metaclass derived from ABCMeta cannot be instantiated unless all of its abstract properties are overridden. The abstract properties can be called using any of the normal 'super' call mechanisms.

Usage:

class C: __metaclass__ = ABCMeta @abstractproperty def my_abstract_property(self):

...

This defines a read-only property; you can also define a read-write abstract property using the 'long' form of property declaration:

class C: __metaclass__ = ABCMeta def getx(self): ... def setx(self, value): ... x = abstractproperty(getx, setx)

max_width

A decorator indicating abstract properties.

Requires that the metaclass is ABCMeta or derived from it. A class that has a metaclass derived from ABCMeta cannot be instantiated unless all of its abstract properties are overridden. The abstract properties can be called using any of the normal 'super' call mechanisms.

Usage:

class C: __metaclass__ = ABCMeta @abstractproperty def my_abstract_property(self):

...

This defines a read-only property; you can also define a read-write abstract property using the 'long' form of property declaration:

class C: __metaclass__ = ABCMeta def getx(self): ... def setx(self, value): ... x = abstractproperty(getx, setx)

width

A decorator indicating abstract properties.

Requires that the metaclass is ABCMeta or derived from it. A class that has a metaclass derived from ABCMeta cannot be instantiated unless all of its abstract properties are overridden. The abstract properties can be called using any of the normal 'super' call mechanisms.

Usage:

class C: __metaclass__ = ABCMeta @abstractproperty def my_abstract_property(self):

...

This defines a read-only property; you can also define a read-write abstract property using the 'long' form of property declaration:

class C: __metaclass__ = ABCMeta def getx(self): ... def setx(self, value): ... x = abstractproperty(getx, setx)

Scopes

Create `Scope` objects using *`instrument()`*.

Supported Models

Tektronix Oscilloscopes Driver module for Tektronix oscilloscopes. Currently supports

- TDS 3000 series
- MSO/DPO 4000 series

class `instrumental.drivers.scopes.tektronix.`**`MSO_DPO_4000`**(*name=None*, *visa_inst=None*)

A Tektronix MSO/DPO 4000 series oscilloscope.

Methods

`are_measurement_stats_on`()	Returns whether measurement statistics are currently enabled
`disable_measurement_stats`()	Disables measurement statistics
`enable_measurement_stats`([enable])	Enables measurement statistics.
`get_data`([channel])	Retrieve a trace from the scope.
`get_math_function`()	
`read_measurement_stats`(num)	Read the value and statistics of a measurement.
`read_measurement_value`(num)	Read the value of a measurement.
`run_acquire`()	Sets the acquire state to 'run'
`set_math_function`(expr)	Set the expression used by the MATH channel.
`set_measurement_nsamps`(nsamps)	Sets the number of samples used to compute measurements.
`set_measurement_params`(num, mtype, channel)	Set the parameters for a measurement.
`stop_acquire`()	Sets the acquire state to 'stop'

class `instrumental.drivers.scopes.tektronix.`**`TDS_3000`**(*name=None*, *visa_inst=None*)

A Tektronix TDS 3000 series oscilloscope.

Methods

`are_measurement_stats_on`()	Returns whether measurement statistics are currently enabled
`disable_measurement_stats`()	Disables measurement statistics
`enable_measurement_stats`([enable])	Enables measurement statistics.
`get_data`([channel])	Retrieve a trace from the scope.
`get_math_function`()	
`read_measurement_stats`(num)	Read the value and statistics of a measurement.
`read_measurement_value`(num)	Read the value of a measurement.
`run_acquire`()	Sets the acquire state to 'run'
`set_math_function`(expr)	Set the expression used by the MATH channel.
`set_measurement_nsamps`(nsamps)	Sets the number of samples used to compute measurements.
`set_measurement_params`(num, mtype, channel)	Set the parameters for a measurement.
`stop_acquire`()	Sets the acquire state to 'stop'

class `instrumental.drivers.scopes.tektronix.`**`TekScope`**(*name=None*, *visa_inst=None*)

A base class for Tektronix scopes. Supports at least TDS 3000 series as well as MSO/DPO 4000 series scopes.

Methods

are_measurement_stats_on()	Returns whether measurement statistics are currently enabled
disable_measurement_stats()	Disables measurement statistics
enable_measurement_stats([enable])	Enables measurement statistics.
get_data([channel])	Retrieve a trace from the scope.
get_math_function()	
read_measurement_stats(num)	Read the value and statistics of a measurement.
read_measurement_value(num)	Read the value of a measurement.
run_acquire()	Sets the acquire state to 'run'
set_math_function(expr)	Set the expression used by the MATH channel.
set_measurement_nsamps(nsamps)	Sets the number of samples used to compute measurements.
set_measurement_params(num, mtype, channel)	Set the parameters for a measurement.
stop_acquire()	Sets the acquire state to 'stop'

`__init__`(*name=None*, *visa_inst=None*)

Create a scope object that has the given VISA name *name* and connect to it. You can find available instrument names using the VISA Instrument Manager.

`are_measurement_stats_on`()

Returns whether measurement statistics are currently enabled

`disable_measurement_stats`()

Disables measurement statistics

`enable_measurement_stats`(*enable=True*)

Enables measurement statistics.

When enabled, measurement statistics are kept track of, including 'mean', 'stddev', 'minimum', 'maximum', and 'nsamps'.

Parameters **enable** : bool

Whether measurement statistics should be enabled

`get_data`(*channel=1*)

Retrieve a trace from the scope.

Pulls data from channel `channel` and returns it as a tuple `(t,y)` of unitful arrays.

Parameters **channel** : int, optional

Channel number to pull trace from. Defaults to channel 1.

Returns **t, y** : pint.Quantity arrays

Unitful arrays of data from the scope. `t` is in seconds, while `y` is in volts.

`get_math_function`()

`read_measurement_stats`(*num*)

Read the value and statistics of a measurement.

Parameters **num** : int

Number of the measurement to read from, from 1-4

Returns **stats** : dict

Dictionary of measurement statistics. Includes value, mean, stddev, minimum, maximum, and nsamps.

read_measurement_value(*num*)

Read the value of a measurement.

Parameters **num** : int

Number of the measurement to read from, from 1-4

Returns **value** : pint.Quantity

Value of the measurement

run_acquire()

Sets the acquire state to 'run'

set_math_function(*expr*)

Set the expression used by the MATH channel.

Parameters **expr** : str

a string representing the MATH expression, using channel variables CH1, CH2, etc. eg. 'CH1/CH2+CH3'

set_measurement_nsamps(*nsamps*)

Sets the number of samples used to compute measurements.

Parameters **nsamps** : int

Number of samples used to compute measurements

set_measurement_params(*num*, *mtype*, *channel*)

Set the parameters for a measurement.

Parameters **num** : int

Measurement number to set, from 1-4.

mtype : str

Type of the measurement, e.g. 'amplitude'

channel : int

Number of the channel to measure.

stop_acquire()

Sets the acquire state to 'stop'

Function Generators

Create `FunctionGenerator` objects using *`instrument()`*.

Supported Models

Tektronix Function Generators Driver module for Tektronix function generators. Currently supports:

- AFG 3000 series

class `instrumental.drivers.funcgenerators.tektronix.`**`AFG_3000`**(*visa_inst*)

Methods

AM_enabled([channel])	Returns whether amplitude modulation is enabled.
FM_enabled([channel])	Returns whether frequency modulation is enabled.
FSK_enabled([channel])	Returns whether frequency-shift keying modulation is enabled.
PM_enabled([channel])	Returns whether phase modulation is enabled.
PWM_enabled([channel])	Returns whether pulse width modulation is enabled.
burst_enabled([channel])	Returns whether burst mode is enabled.
disable_AM([channel])	Disable amplitude modulation mode.
disable_FM([channel])	Disable frequency modulation mode.
disable_FSK([channel])	Disable frequency-shift keying mode.
disable_PM([channel])	Disable phase modulation mode.
disable_PWM([channel])	Disable pulse width modulation mode.
disable_burst([channel])	Disable burst mode.
enable_AM([enable, channel])	Enable amplitude modulation mode.
enable_FM([enable, channel])	Enable frequency modulation mode.
enable_FSK([enable, channel])	Enable frequency-shift keying mode.
enable_PM([enable, channel])	Enable phase modulation mode.
enable_PWM([enable, channel])	Enable pulse width modulation mode.
enable_burst([enable, channel])	Enable burst mode.
get_dbm([channel])	Get the amplitude of the current waveform in dBm.
get_ememory()	Get array of data from edit memory.
get_frequency([channel])	Get the frequency to be used in fixed frequency mode.
get_frequency_mode([channel])	Get the frequency mode.
get_vpp([channel])	Get the peak-to-peak voltage of the current waveform.
get_vrms([channel])	Get the RMS voltage of the current waveform.
set_am_depth(depth[, channel])	Set depth of amplitude modulation.
set_arb_func(data[, interp, num_pts])	Write arbitrary waveform data to EditMemory.
set_dbm(dbm[, channel])	Set the amplitude of the current waveform in dBm.
set_frequency(freq[, change_mode, channel])	Set the frequency to be used in fixed frequency mode.
set_frequency_mode(mode[, channel])	Set the frequency mode.
set_function(**kwargs)	Set selected function parameters.
set_function_shape(shape[, channel])	Set shape of output function.
set_high(high[, channel])	Set the high voltage level of the current waveform.
set_low(low[, channel])	Set the low voltage level of the current waveform.
set_offset(offset[, channel])	Set the voltage offset of the current waveform.
set_phase(phase[, channel])	Set the phase offset of the current waveform.
set_sweep([channel])	Set selected sweep parameters.
set_sweep_center(center[, channel])	Set the sweep frequency center.
set_sweep_hold_time(time[, channel])	Set the hold time of the sweep.
set_sweep_return_time(time[, channel])	Set the return time of the sweep.
set_sweep_spacing(spacing[, channel])	Set whether a sweep is linear or logarithmic.
set_sweep_span(span[, channel])	Set the sweep frequency span.
set_sweep_start(start[, channel])	Set the sweep start frequency.
set_sweep_stop(stop[, channel])	Set the sweep stop frequency.
set_sweep_time(time[, channel])	Set the sweep time.
set_vpp(vpp[, channel])	Set the peak-to-peak voltage of the current waveform.
set_vrms(vrms[, channel])	Set the amplitude of the current waveform in dBm.
sweep_enabled([channel])	Whether the frequency mode is sweep.

AM_enabled(*channel=1*)
Returns whether amplitude modulation is enabled.

Returns bool

Whether AM is enabled.

FM_enabled(*channel=1*)
Returns whether frequency modulation is enabled.

Returns bool

Whether FM is enabled.

FSK_enabled(*channel=1*)
Returns whether frequency-shift keying modulation is enabled.

Returns bool

Whether FSK is enabled.

PM_enabled(*channel=1*)
Returns whether phase modulation is enabled.

Returns bool

Whether PM is enabled.

PWM_enabled(*channel=1*)
Returns whether pulse width modulation is enabled.

Returns bool

Whether PWM is enabled.

__init__(*visa_inst*)
Constructor for an AFG 3000 Function Generator object. End users should not use this directly, and should instead use `instrument()`

burst_enabled(*channel=1*)
Returns whether burst mode is enabled.

Returns bool

Whether burst mode is enabled.

disable_AM(*channel=1*)
Disable amplitude modulation mode.

disable_FM(*channel=1*)
Disable frequency modulation mode.

disable_FSK(*channel=1*)
Disable frequency-shift keying mode.

disable_PM(*channel=1*)
Disable phase modulation mode.

disable_PWM(*channel=1*)
Disable pulse width modulation mode.

disable_burst(*channel=1*)
Disable burst mode.

enable_AM(*enable=True*, *channel=1*)
Enable amplitude modulation mode.

Parameters **enable** : bool, optional

Whether to enable or disable AM

enable_FM(*enable=True*, *channel=1*)

Enable frequency modulation mode.

Parameters **enable** : bool, optional

Whether to enable or disable FM

enable_FSK(*enable=True*, *channel=1*)

Enable frequency-shift keying mode.

Parameters **enable** : bool, optional

Whether to enable or disable FSK

enable_PM(*enable=True*, *channel=1*)

Enable phase modulation mode.

Parameters **enable** : bool, optional

Whether to enable or disable PM

enable_PWM(*enable=True*, *channel=1*)

Enable pulse width modulation mode.

Parameters **enable** : bool, optional

Whether to enable or disable PWM

enable_burst(*enable=True*, *channel=1*)

Enable burst mode.

Parameters **enable** : bool, optional

Whether to enable or disable burst mode.

get_dbm(*channel=1*)

Get the amplitude of the current waveform in dBm.

Note that this returns a float, not a pint.Quantity

Returns **dbm** : float

The current waveform's dBm amplitude

get_ememory()

Get array of data from edit memory.

Returns numpy.array

Data retrieved from the AFG's edit memory.

get_frequency(*channel=1*)

Get the frequency to be used in fixed frequency mode.

get_frequency_mode(*channel=1*)

Get the frequency mode.

Returns 'fixed' or 'sweep'

The frequency mode

get_vpp(*channel=1*)

Get the peak-to-peak voltage of the current waveform.

Returns **vpp** : pint.Quantity

The current waveform's peak-to-peak voltage

get_vrms(*channel=1*)

Get the RMS voltage of the current waveform.

Returns **vrms** : pint.Quantity

The current waveform's RMS voltage

set_am_depth(*depth*, *channel=1*)

Set depth of amplitude modulation.

Parameters **depth** : number

Depth of modulation in percent. Must be between 0.0% and 120.0%. Has resolution of 0.1%.

set_arb_func(*data*, *interp=None*, *num_pts=10000*)

Write arbitrary waveform data to EditMemory.

Parameters **data** : array_like

A 1D array of real values to be used as evenly-spaced points. The values will be normalized to extend from 0 t0 16382. It must have a length in the range [2, 131072]

interp : str or int, optional

Interpolation to use for smoothing out data. None indicates no interpolation. Values include ('linear', 'nearest', 'zero', 'slinear', 'quadratic', 'cubic'), or an int to specify the order of spline interpolation. See scipy.interpolate.interp1d for details.

num_pts : int

Number of points to use in interpolation. Default is 10000. Must be greater than or equal to the number of points in `data`, and at most 131072.

set_dbm(*dbm*, *channel=1*)

Set the amplitude of the current waveform in dBm.

Note that this returns a float, not a pint.Quantity

Parameters **dbm** : float

The current waveform's dBm amplitude

set_frequency(*freq*, *change_mode=True*, *channel=1*)

Set the frequency to be used in fixed frequency mode.

Parameters **freq** : pint.Quantity

The frequency to be used in fixed frequency mode.

change_mode : bool, optional

If True, will set the frequency mode to `fixed`.

set_frequency_mode(*mode*, *channel=1*)

Set the frequency mode.

In fixed mode, the waveform's frequency is kept constant. In sweep mode, it is swept according to the sweep settings.

Parameters **mode** : {'fixed', 'sweep'}

Mode to switch to.

set_function(***kwargs*)

Set selected function parameters. Useful for setting multiple parameters at once. See individual setters for more details.

When setting the waveform amplitude, you may use up to two of `high`, `low`, `offset`, and `vpp`/`vrms`/`dbm`.

Parameters shape : {'SINusoid', 'SQUare', 'PULSe', 'RAMP', 'PRNoise', 'DC', 'SINC', 'GAUSsian', 'LORentz', 'ERISe', 'EDECay', 'HAVersine',

'USER1', 'USER2', 'USER3', 'USER4', 'EMEMory'}, optional

Shape of the waveform. Case-insenitive, abbreviation or full string.

phase : pint.Quantity or string or number, optional

Phase of the waveform in radian-compatible units.

vpp, vrms, dbm : pint.Quantity or string, optional

Amplitude of the waveform in volt-compatible units.

offset : pint.Quantity or string, optional

Offset of the waveform in volt-compatible units.

high : pint.Quantity or string, optional

High level of the waveform in volt-compatible units.

low : pint.Quantity or string, optional

Low level of the waveform in volt-compatible units.

channel : {1, 2}, optional

Output channel to modify. Some models may have only one channel.

set_function_shape(*shape*, *channel=1*)

Set shape of output function.

Parameters shape : {'SINusoid', 'SQUare', 'PULSe', 'RAMP', 'PRNoise', 'DC', 'SINC', 'GAUSsian', 'LORentz', 'ERISe', 'EDECay', 'HAVersine', 'USER1', 'USER2', 'USER3', 'USER4', 'EMEMory'}, optional

Shape of the waveform. Case-insenitive string that contains a valid shape or its abbreviation. The abbreviations are indicated above by capitalization. For example, `sin`, `SINUSOID`, and `SiN` are all valid inputs, while `sinus` is not.

channel : {1, 2}, optional

Output channel to modify. Some models may have only one channel.

set_high(*high*, *channel=1*)

Set the high voltage level of the current waveform.

This changes the high level while keeping the low level fixed.

Parameters high : pint.Quantity

The new high level in volt-compatible units

set_low(*low*, *channel=1*)

Set the low voltage level of the current waveform.

This changes the low level while keeping the high level fixed.

Parameters low : pint.Quantity

The new low level in volt-compatible units

set_offset (*offset*, *channel=1*)

Set the voltage offset of the current waveform.

This changes the offset while keeping the amplitude fixed.

Parameters **offset** : pint.Quantity

The new voltage offset in volt-compatible units

set_phase (*phase*, *channel=1*)

Set the phase offset of the current waveform.

Parameters **phase** : pint.Quantity or number

The new low level in radian-compatible units. Unitless numbers are treated as radians.

set_sweep (*channel=1*, ***kwargs*)

Set selected sweep parameters.

Automatically enables sweep mode.

Parameters **start** : pint.Quantity

The start frequency of the sweep in Hz-compatible units

stop : pint.Quantity

The stop frequency of the sweep in Hz-compatible units

span : pint.Quantity

The frequency span of the sweep in Hz-compatible units

center : pint.Quantity

The center frequency of the sweep in Hz-compatible units

sweep_time : pint.Quantity

The sweep time in second-compatible units. Must be between 1 ms and 300 s

hold_time : pint.Quantity

The hold time in second-compatible units

return_time : pint.Quantity

The return time in second-compatible units

spacing : {'linear', 'lin', 'logarithmic', 'log'}

The spacing in time of the sweep frequencies

set_sweep_center (*center*, *channel=1*)

Set the sweep frequency center.

This sets the sweep center frequency while keeping the sweep frequency span fixed. The start and stop frequencies will be changed.

Parameters **center** : pint.Quantity

The center frequency of the sweep in Hz-compatible units

set_sweep_hold_time (*time*, *channel=1*)

Set the hold time of the sweep.

The hold time is the amount of time that the frequency is held constant after reaching the stop frequency.

Parameters **time** : pint.Quantity

The hold time in second-compatible units

set_sweep_return_time(*time*, *channel=1*)

Set the return time of the sweep.

The return time is the amount of time that the frequency spends sweeping from the stop frequency back to the start frequency. This does not include hold time.

Parameters **time** : pint.Quantity

The return time in second-compatible units

set_sweep_spacing(*spacing*, *channel=1*)

Set whether a sweep is linear or logarithmic.

Parameters **spacing** : {'linear', 'lin', 'logarithmic', 'log'}

The spacing in time of the sweep frequencies

set_sweep_span(*span*, *channel=1*)

Set the sweep frequency span.

This sets the sweep frequency span while keeping the center frequency fixed. The start and stop frequencies will be changed.

Parameters **span** : pint.Quantity

The frequency span of the sweep in Hz-compatible units

set_sweep_start(*start*, *channel=1*)

Set the sweep start frequency.

This sets the start frequency while keeping the stop frequency fixed. The span and center frequencies will be changed.

Parameters **start** : pint.Quantity

The start frequency of the sweep in Hz-compatible units

set_sweep_stop(*stop*, *channel=1*)

Set the sweep stop frequency.

This sets the stop frequency while keeping the start frequency fixed. The span and center frequencies will be changed.

Parameters **stop** : pint.Quantity

The stop frequency of the sweep in Hz-compatible units

set_sweep_time(*time*, *channel=1*)

Set the sweep time.

The sweep time does not include hold time or return time. Sweep time must be between 1 ms and 300 s.

Parameters **time** : pint.Quantity

The sweep time in second-compatible units. Must be between 1 ms and 200 s

set_vpp(*vpp*, *channel=1*)

Set the peak-to-peak voltage of the current waveform.

Parameters **vpp** : pint.Quantity

The new peak-to-peak voltage

set_vrms(*vrms*, *channel=1*)
Set the amplitude of the current waveform in dBm.

Parameters **vrms** : pint.Quantity

The new RMS voltage

sweep_enabled(*channel=1*)
Whether the frequency mode is sweep.

Just a convenience method to avoid writing `get_frequency_mode() == 'sweep'`.

Returns bool

Whether the frequency mode is sweep

Power Meters

Create `PowerMeter` objects using *`instrument()`*.

Supported Models

Newport Power Meters Driver module for Newport power meters. Supports:

- 1830-C

class `instrumental.drivers.powermeters.newport.`**`Newport_1830_C`**(*inst*)
A Newport 1830-C power meter

Methods

attenuator_enabled()	Whether the attenuator is enabled
disable_attenuator()	Disable the power meter attenuator
disable_auto_range()	Disable auto-range
disable_hold()	Disable hold mode
disable_zero()	Disable the zero function
enable_attenuator([enabled])	Enable the power meter attenuator
enable_auto_range()	Enable auto-range
enable_hold([enable])	Enable hold mode
enable_zero([enable])	Enable the zero function
get_filter()	Get the current setting for the averaging filter
get_power()	Get the current power measurement
get_range()	Return the current range setting as an int
get_status_byte()	Query the status byte register and return it as an int
get_units()	Get the units used for displaying power measurements
get_wavelength()	Get the input wavelength setting
hold_enabled()	Whether hold mode is enabled
is_measurement_valid()	Whether the current measurement is valid
set_medium_filter()	Set the averaging filter to medium mode
set_no_filter()	Set the averaging filter to fast mode, i.e.
set_range(range_num)	Set the range for power measurements
set_slow_filter()	Set the averaging filter to slow mode
set_units(units)	Set the units for displaying power measurements

Continued on next page

Table 1.13 – continued from previous page

set_wavelength(wavelength)	Set the input signal wavelength setting
store_reference()	Store the current power input as a reference
zero_enabled()	Whether the zero function is enabled

__init__(*inst*)

attenuator_enabled()
Whether the attenuator is enabled

Returns **enabled** : bool

whether the attenuator is enabled

disable_attenuator()
Disable the power meter attenuator

disable_auto_range()
Disable auto-range

Leaves the signal range at its current position.

disable_hold()
Disable hold mode

disable_zero()
Disable the zero function

enable_attenuator(*enabled=True*)
Enable the power meter attenuator

enable_auto_range()
Enable auto-range

enable_hold(*enable=True*)
Enable hold mode

enable_zero(*enable=True*)
Enable the zero function

When enabled, the next power reading is stored as a background value and is subtracted off of all subsequent power readings.

get_filter()
Get the current setting for the averaging filter

Returns SLOW_FILTER, MEDIUM_FILTER, NO_FILTER

the current averaging filter

get_power()
Get the current power measurement

Returns **power** : Quantity

Power in units of watts, regardless of the power meter's current 'units' setting.

get_range()
Return the current range setting as an int

1 corresponds to the lowest range, while 8 is the highest range (least amplifier gain).

Note that this does not query the status of auto-range.

Returns **range** : int

the current range setting. Possible values are from 1-8.

get_status_byte()
Query the status byte register and return it as an int

get_units()
Get the units used for displaying power measurements

Returns units : str

'watts', 'db', 'dbm', or 'rel'

get_wavelength()
Get the input wavelength setting

hold_enabled()
Whether hold mode is enabled

Returns enabled : bool

True if in hold mode, False if in run mode

is_measurement_valid()
Whether the current measurement is valid

The measurement is considered invalid if the power meter is saturated, over-range or busy.

set_medium_filter()
Set the averaging filter to medium mode

The medium filter uses a 4-measurement running average.

set_no_filter()
Set the averaging filter to fast mode, i.e. no averaging

set_range(*range_num*)
Set the range for power measurements

range_num = 0 for auto-range range_num = 1 to 8 for manual signal range (1 is lowest, and 8 is highest)

Parameters n : int

Sets the signal range for the input signal.

set_slow_filter()
Set the averaging filter to slow mode

The slow filter uses a 16-measurement running average.

set_units(*units*)
Set the units for displaying power measurements

The different unit modes are watts, dB, dBm, and REL. Each displays the power in a different way.

'watts' displays absolute power in watts

'dBm' displays power in dBm (i.e. dBm = 10 * log(P / 1mW))

'dB' displays power in dB relative to the current reference power (i.e. dB = 10 * log(P / Pref). At power-up, the reference power is set to 1mW.

'REL' displays power relative to the current reference power (i.e. REL = P / Pref)

The current reference power can be set using 'store_reference'().

Parameters units : 'watts', 'dBm', 'dB', or 'REL'

Case-insensitive str indicating which units mode to enter.

set_wavelength(*wavelength*)
Set the input signal wavelength setting

Parameters **wavelength** : Quantity

wavelength of the input signal, in units of [length]

store_reference()
Store the current power input as a reference

Sets the current power measurement as the reference power for future dB or relative measurements.

zero_enabled()
Whether the zero function is enabled

MEDIUM_FILTER = 2

NO_FILTER = 3

SLOW_FILTER = 1

Thorlabs Power Meters Driver module for Thorlabs power meters. Supports:

- PM100D

class instrumental.drivers.powermeters.thorlabs.**PM100D**(*visa_inst*)
A Thorlabs PM100D series power meter

Methods

auto_range_enabled()	Whether auto-ranging is enabled
disable_auto_range()	Disable auto-ranging
enable_auto_range([enable])	Enable auto-ranging
get_num_averaged()	Get the number of samples to average
get_power()	Get the current power measurement
get_range()	Get the current input range's max power
get_wavelength()	Get the input signal wavelength setting
set_num_averaged(num_averaged)	Set the number of samples to average
set_wavelength(wavelength)	Set the input signal wavelength setting

__init__(*visa_inst*)

auto_range_enabled()
Whether auto-ranging is enabled

Returns **bool** : enabled

disable_auto_range()
Disable auto-ranging

enable_auto_range(*enable=True*)
Enable auto-ranging

get_num_averaged()
Get the number of samples to average

Returns **num_averaged** : int

number of samples that are averaged

get_power()
Get the current power measurement

Returns power : Quantity

the current power measurement

get_range()
Get the current input range's max power

get_wavelength()
Get the input signal wavelength setting

Returns wavelength : Quantity

the input signal wavelength in units of [length]

set_num_averaged(*num_averaged*)
Set the number of samples to average

Each sample takes approximately 3ms. Thus, averaging over 1000 samples would take about a second.

Parameters num_averaged : int

number of samples to average

set_wavelength(*wavelength*)
Set the input signal wavelength setting

Parameters wavelength : Quantity

the input signal wavelength in units of [length]

Wavemeters

Create `Wavemeter` objects using *`instrument()`*.

Supported Models

Burleigh Wavemeters Driver module for Burleigh wavemeters. Supports:

- WA-1000/1500

class `instrumental.drivers.wavemeters.burleigh.`**`WA_1000`**(*inst*)
A Burleigh WA-1000/1500 wavemeter

Methods

averaging_enabled()	Whether averaging mode is enabled
disable_averaging()	Disable averaging mode
enable_averaging([enable])	Enable averaging mode
get_deviation()	Get the current deviation
get_num_averaged()	Get the number of samples used in averaging mode
get_pressure()	Get the barometric pressure inside the wavemeter
get_setpoint()	Get the wavelength setpoint
get_temperature()	Get the temperature inside the wavemeter
get_wavelength()	Get the wavelength

Continued on next page

Table 1.15 – continued from previous page

`is_locked`()	Whether the front panel is locked or not
`lock`([lock])	Lock the front panel of the wavemeter, preventing manual input
`set_num_averaged`(num)	Set the number of samples used in averaging mode
`set_setpoint`(setpoint)	Set the wavelength setpoint
`unlock`()	Unlock the front panel of the wavemeter, allowing manual input

__init__(*inst*)

averaging_enabled()
Whether averaging mode is enabled

disable_averaging()
Disable averaging mode

enable_averaging(*enable=True*)
Enable averaging mode

get_deviation()
Get the current deviation

Returns **deviation** : Quantity

The wavelength difference between the current input wavelength and the fixed setpoint.

get_num_averaged()
Get the number of samples used in averaging mode

get_pressure()
Get the barometric pressure inside the wavemeter

Returns **pressure** : Quantity

The barometric pressure inside the wavemeter

get_setpoint()
Get the wavelength setpoint

Returns **setpoint** : Quantity

the wavelength setpoint

get_temperature()
Get the temperature inside the wavemeter

Returns **temperature** : Quantity

The temperature inside the wavemeter

get_wavelength()
Get the wavelength

Returns **wavelength** : Quantity

The current input wavelength measurement

is_locked()
Whether the front panel is locked or not

lock(*lock=True*)
Lock the front panel of the wavemeter, preventing manual input

When locked, the wavemeter can only be controlled remotely by a computer. To unlock, use `unlock()` or hit the 'Remote' button on the wavemeter's front panel.

set_num_averaged(*num*)
Set the number of samples used in averaging mode

When averaging mode is enabled, the wavemeter calculates a running average of the last `num` samples.

Parameters **num** : int

Number of samples to average. Must be between 2 and 50.

set_setpoint(*setpoint*)
Set the wavelength setpoint

The setpoint is a fixed wavelength used to compute the deviation. It is used for display and to determine the analog output voltage.

Parameters **setpoint** : Quantity

Wavelength of the setpoint, in units of [length]

unlock()
Unlock the front panel of the wavemeter, allowing manual input

---

Functions

class `instrumental.drivers.`**`Instrument`**
Base class for all instruments.

Methods

save_instrument(name[, force])	Save an entry for this instrument in the config file.

save_instrument(*name*, *force=False*)
Save an entry for this instrument in the config file.

Parameters **name** : str

The name to give the instrument, e.g. 'myCam'

force : bool, optional

Force overwrite of the old entry for instrument `name`. By default, Instrumental will raise an exception if you try to write to a name that's already taken. If `force` is True, the old entry will be commented out (with a warning given) and a new entry will be written.

class `instrumental.drivers.`**`InstrumentMeta`**
Instrument metaclass.

Implements inheritance of method and property docstrings for subclasses of Instrument. That way e.g. you don't have to repeat the docstring of an abstract method, though you can provide a docstring in case more specific documentation is useful.

If the child's docstring contains only a single-line function signature, it is prepended to its parent's docstring rather than overriding it comopletely. This is useful for the explicitly specifying signatures for methods that are wrapped by a decorator.

Methods

`__call__`(...) <==> x(...)	
`mro`(() -> list)	return a type's method resolution order

`instrumental.drivers.`**`instrument`**(*inst=None*, ***kwargs*)

Create any Instrumental instrument object from an alias, parameters, or an existing instrument.

```
>>> inst1 = instrument('MYAFG')
>>> inst2 = instrument(visa_address='TCPIP::192.168.1.34::INSTR')
>>> inst3 = instrument({'visa_address': 'TCPIP:192.168.1.35::INSTR'})
>>> inst4 = instrument(inst1)
```

`instrumental.drivers.`**`list_instruments`**(*server=None*)

Returns a list of info about available instruments.

May take a few seconds because it must poll hardware devices.

It actually returns a list of specialized dict objects that contain parameters needed to create an instance of the given instrument. You can then get the actual instrument by passing the dict to `instrument()`.

```
>>> inst_list = get_instruments()
>>> print(inst_list)
[<NIDAQ 'Dev1'>, <TEKTRONIX 'TDS 3032'>, <TEKTRONIX 'AFG3021B'>]
>>> inst = instrument(inst_list[0])
```

Parameters **server** : str, optional

The remote Instrumental server to query. It can be an alias from your instrumental.conf file, or a str of the form `(hostname|ip-address)[:port]`, e.g. '192.168.1.10:12345'. Is None by default, meaning search on the local machine.

`instrumental.drivers.`**`list_visa_instruments`**()

Returns a list of info about available VISA instruments.

May take a few seconds because it must poll the network.

It actually returns a list of specialized dict objects that contain parameters needed to create an instance of the given instrument. You can then get the actual instrument by passing the dict to `instrument()`.

```
>>> inst_list = get_visa_instruments()
>>> print(inst_list)
[<TEKTRONIX 'TDS 3032'>, <TEKTRONIX 'AFG3021B'>]
>>> inst = instrument(inst_list[0])
```

Example

```
>>> from instrumental import instrument
>>> scope = instrument('my_scope_alias')
>>> x, y = scope.get_data()
```

1.5.2 Fitting

Module containing utilities related to fitting.

Still very much a work in progress...

`instrumental.fitting.`**`curve_fit`**(*f*, *xdata*, *ydata*, *p0=None*, *sigma=None*, ***kw*)

Wrapper for scipy's curve_fit that works with pint Quantities.

`instrumental.fitting.`**`guided_ringdown_fit`**(*data_x*, *data_y*)

Guided fit of a ringdown. Takes *data_x* and *data_y* as `pint` Quantities with dimensions of time and voltage, respectively. Plots the data and asks user to manually crop to select the region to fit.

It then does a rough linear fit to find initial parameters and performs a nonlinear fit.

Finally, it plots the data with the curve fit overlayed and returns the full-width at half-max (FWHM) with units.

`instrumental.fitting.`**`guided_trace_fit`**(*data_x*, *data_y*, *EOM_freq*)

Guided fit of a cavity scan trace that has sidebands. Takes *data_x* and *data_y* as `pint` Quantities, and the EOM frequency *EOM_freq* can be anything that the `pint.Quantity` constructor understands, like an existing `pint.Quantity` or a string, e.g. `'5 Mhz'`.

It plots the data then asks the user to identify the three maxima by by clicking on them in left-to-right order. It then uses that input to estimate and then do a nonlinear fit of the parameters.

Finally, it plots the data with the curve fit overlayed and returns the parameters in a map.

The parameters are `A0`, `B0`, `FWHM`, `nu0`, and `dnu`.

`instrumental.fitting.`**`lorentzian`**(*x*, *A*, *x0*, *FWHM*)

Lorentzian curve. Takes an array `x` and returns an array $\frac{A}{1+(\frac{2(x-x0)}{FWHM})^2}$

`instrumental.fitting.`**`triple_lorentzian`**(*nu*, *A0*, *B0*, *FWHM*, *nu0*, *dnu*, *y0*)

Triple lorentzian curve. Takes an array `nu` and returns an array that is the sum of three lorentzians `lorentzian(nu, A0, nu0, FWHM) + lorentzian(nu, B0, nu0-dnu, FWHM) + lorentzian(nu, B0, nu0+dnu, FWHM)`.

1.5.3 Plotting

Module that provides unit-aware plotting functions that can be used as a drop-in replacement for matplotlib.pyplot.

Also acts as a repository for useful plotting tools, like slider-plots.

`instrumental.plotting.`**`param_plot`**(*x*, *func*, *params*, ***kwargs*)

Plot a function with user-adjustable parameters.

Parameters **x** : array_like

Independent (x-axis) variable.

func : function

Function that takes as its first argument an independent variable and as subsequent arguments takes parameters. It should return an output array the same dimension as `x`, which is plotted as the y-variable.

params : dict

Dictionary whose keys are strings named exactly as the parameter arguments to `func` are. [More info on options]

Returns **final_params** : dict

A dict whose keys are the same as `params` and whose values correspond to the values selected by the slider. `final_params` will continue to change until the figure is closed, at which point it has the final parameter values the user chose. This is useful for hand-fitting curves.

`instrumental.plotting.`**`plot`**(**args*, ***kwargs*)
Quantity-aware wrapper of pyplot.plot

`instrumental.plotting.`**`xlabel`**(*s*, **args*, ***kwargs*)
Quantity-aware wrapper of pyplot.xlabel

Automatically adds parenthesized units to the end of `s`.

`instrumental.plotting.`**`ylabel`**(*s*, **args*, ***kwargs*)
Quantity-aware wrapper of pyplot.ylabel.

Automatically adds parenthesized units to the end of `s`.

1.5.4 Optics

Instrumental's Optics package is useful for exploring and scripting basic gaussian optics using the ABCD matrix approach. The package is split up into three main categories: optical elements, beam tools, and beam plotting tools.

Optical Elements

Instrumental's optical elements are based on simple numerical ABCD matrix representations and include Mirrors, Lenses, Spaces, and Interfaces. Each provides a useful constructor to create them in a way that's conceptually simple and clear.

class `instrumental.optics.optical_elements.`**`ABCD`**(*A*, *B*, *C*, *D*)
A simple ABCD (ray transfer) matrix class.

ABCD objects support mutiplication with scalar numbers and other ABCD objects.

Methods

elems()	Get the matrix elements.

`__init__`(*A*, *B*, *C*, *D*)
Create an ABCD matrix from its elements.

The matrix is a 2x2 of the form:

```
[A B]
[C D]
```

Parameters **A,B,C,D** : Quantity objects

`A` and `D` are dimensionless. `B` has units of [length] (e.g. 'mm' or 'rad/mm'), and `C` has units of 1/[length].

`elems`()
Get the matrix elements.

Returns **A, B, C, D** : tuple of Quantity objects

The matrix elements

class `instrumental.optics.optical_elements.`**`Interface`**(*n1*, *n2*, *R=None*, *aoi=None*, *aot=None*)

An interface between media with different refractive indices

`__init__`(*n1*, *n2*, *R=None*, *aoi=None*, *aot=None*)

Parameters **n1** : number

The refractive index of the initial material

n2 : number

The refractive index of the final material

R : Quantity or str, optional

The radius of curvature of the interface's spherical surface, in units of length. Defaults to `None`, indicating a flat interface.

aoi : Quantity or str or number, optional

The angle of incidence of the beam relative to the interface, defined as the angle between the interface's surface normal and the _incident_ beam's axis. If not specified but `aot` is given, aot will be used. Otherwise, `aoi` is assumed to be 0, indicating normal incidence. A raw number is assumed to be in units of degrees.

aot : Quantity or str or number, optional

The angle of transmission of the beam relative to the interface, defined as the angle between the interface's surface normal and the *transmitted* beam's axis. See `aoi` for more details.

class `instrumental.optics.optical_elements.`**`Lens`**(*f*)

A thin lens

`__init__`(*f*)

Parameters **f** : Quantity or str

The focal length of the lens

class `instrumental.optics.optical_elements.`**`Mirror`**(*R=None*, *aoi=0*)

A mirror, possibly curved

`__init__`(*R=None*, *aoi=0*)

Parameters **R** : Quantity or str, optional

The radius of curvature of the mirror's spherical surface. Defaults to `None`, indicating a flat mirror.

aoi : Quantity or str or number, optional

The angle of incidence of the beam on the mirror, defined as the angle between the mirror's surface normal and the beam's axis. Defaults to 0, indicating normal incidence.

class `instrumental.optics.optical_elements.`**`Space`**(*d*, *n=1*)

A space between other optical elements

`__init__`(*d*, *n=1*)

Parameters **d** : Quantity or str

The axial length of the space

n : number, optional

The index of refraction of the medium. Defaults to 1 for vacuum.

Beam Tools

`instrumental.optics.beam_tools.`**`find_cavity_modes`**(*elems*)

Find the eigenmodes of an optical cavity.

Parameters **elems** : list of OpticalElements

ordered list of the cavity elements

Returns **qt_r, qs_r** : complex Quantity objects

1/q for the tangential and sagittal modes, respectively. Has units of 1/[length].

`instrumental.optics.beam_tools.`**`get_w0`**(*q_r*, *lambda_med*)

Get waist size w0 of light with in-medium wavelength lambda_med and reciprocal beam parameter q_r

`instrumental.optics.beam_tools.`**`get_z0`**(*q_r*)

Get z-location z0 of the focus from reciprocal beam parameter q_r

`instrumental.optics.beam_tools.`**`get_zR`**(*q_r*)

Get Rayleigh range zR from reciprocal beam parameter q_r

Beam Plotting

`instrumental.optics.beam_plotting.`**`plot_profile`**(*q_start_t_r*, *q_start_s_r*, *lambda0*, *elems*, *cyclical=False*, *names=()*, *clipping=None*, *show_axis=False*, *show_waists=False*, *zeroat=0*, *zunits='mm'*, *runits='um'*)

Plot tangential and sagittal beam profiles.

Parameters **q_start_t_r, q_start_s_r** : complex Quantity objects

Reciprocal beam parameters for the tangential and sagittal components. They have units of 1/[length].

lambda0 : Quantity

Vacuum wavelength of the beam in units of [length].

elems : list of OpticalElements

Ordered list of optical elements through which the beams pass and are plotted.

Other Parameters **cyclical** : bool

Whether `elems` loops back on itself, i.e. it forms a cavity where the last element is immediately before the first element. Used for labelling the elements correctly if `names` is used.

names : list or tuple of str

Strings used to label the non-`Space` elements on the plot. Vertical lines will be used to denote the element's position.

clipping : float

Clipping loss level to plot. Normally, the beam profile plotted is the usual spot size. However, if `clipping` is given, the profile indicates the distance from the beam axis at which knife-edge clipping power losses are equal to `clipping`.

show_axis : bool

If `show_axis` is `True`, sets the ylim to include the beam axis, i.e. y=0. Otherwise, y limits are automatically set by matplotlib.

show_waists : bool

If `True`, marks beam waists on the plot and labels their size.

zeroat : int

The *index* of the element in `elems` that we should consider as z=0. Useful for looking at distances from some element that's in the middle of the plot.

zunits : str or Quantity or UnitsContainer

Units to use for the z-axis. Must have units of [length]. Defaults to 'mm'.

runits : str or Quantity or UnitsContainer

Units to use for the radial axis. Must have units of [length]. Defaults to 'um'.

1.5.5 Tools

class `instrumental.tools.`**`DataSession`**(*name*, *meas_gen*, *overwrite=False*)

A data-taking session.

Useful for organizing, saving, and live-plotting data while (automatically or manually) taking it.

Methods

create_plot(vars, **kwargs)	Create a plot of the DataSession.
save_summary([overwrite])	
start()	Start collecting data.

`__init__`(*name*, *meas_gen*, *overwrite=False*)

Create a DataSession.

Parameters **name** : str

The name of the session. Used for naming the saved data file.

meas_gen : generator

A generator that, when iterated through, returns individual measurements as dicts. Each dict key is a string that is the name of what's being measured, and its matching value is the corresponding quantity. Most often you'll want to create this generator by writing a generator function.

overwrite : bool

If True, data with the same filename will be overwritten. Defaults to False.

create_plot (*vars*, ***kwargs*)
Create a plot of the DataSession.

This plot is live-updated with data points as you take them.

Parameters **vars** : list of tuples

vars to plot. Each tuple corresponds to a data series, with x-data, y-data, and optional format string. This is meant to be reminiscent of matplotlib's plot function. The x and y data can each either be a string (representing the variable in the measurement dict with that name) or a function that takes kwargs with the name of those in the measurement dict and returns its computed value.

****kwargs** : keyword arguments

used for formatting the plot. These are passed directly to the plot function. Useful for e.g. setting the linewidth.

save_summary (*overwrite=None*)

start ()
Start collecting data.

This function blocks until all data has been collected.

instrumental.tools.**FSRs_from_mode_wavelengths** (*wavelengths*)

instrumental.tools.**diff** (*unitful_array*)

instrumental.tools.**do_ringdown_set** (*set_name*, *base_dir=None*)

instrumental.tools.**find_FSR** ()

instrumental.tools.**fit_ringdown** (*scope*, *channel=1*, *FSR=None*)

instrumental.tools.**fit_ringdown_save** (*subdir=''*, *trace_num=0*, *base_dir=None*)
Read a trace from the scope, save it and fit a ringdown curve.

Parameters **subdir** : string

Subdirectory in which to save the data file.

trace_num : int

An index indicating which trace it is.

base_dir: string

The path of the toplevel data directory.

instrumental.tools.**fit_scan** (*EOM_freq*, *scope*, *channel=1*)

instrumental.tools.**fit_scan_save** (*EOM_freq*, *subdir=''*, *trace_num=0*, *base_dir=None*)

instrumental.tools.**get_photo_fnames** ()

instrumental.tools.**load_data** (*fname*, *delimiter='\t'*)

instrumental.tools.**qappend** (*arr*, *values*, *axis=None*)
Append values to the end of an array-valued Quantity.

1.6 Developer's Guide

A major goal of Instrumental is to try to unify and simplify a lot of common, useful operations. Essential to that is a consistent and coherent interface.

- Simple and common tasks should be simple to perform
- Provide options for more complex tasks
- Documentation is an essential, not a luxury
- Make units standard

1.6.1 The Manifesto

Simple and common tasks should be simple to perform

Tasks that are conceptually simple or commonly performed should be made easy. This means having sane defaults.

Provide options for more complex tasks

Along with sane defaults, provide options. Often this means using optional parameters in functions and methods.

Documentation is an essential, not a luxury

Documentation can be hard or boring to provide, but without it your carefully constructed interfaces are rendered useless. In particular, all functions and methods should have brief summary sentences, detailed explanations, and descriptions of their parameters and return values.

This also includes providing useful error messages and warnings that the average user can actually understand and do something with.

Make units standard

Units in scientific code can be a big issue. Look no further than the commonly-cited Mars Climate Orbiter mishap. Instrumental incorporates unitful quantities using the very nice Pint module. While units are great, it can seem like extra work to start using them. Instrumental strives to use units everywhere to encourage their widespread use. Part of this is making units a joy to use with matplotlib.

1.6.2 Coding Conventions

As with most Python projects, you should be keeping in mind the style suggestions in PEP8. In particular:

- Use 4 spaces per indent (not tabs!)
- Classes should be named using `CapWords` capitalization
- Functions and methods should be named using `lower_case_with_underscores`
  - As an exception, python wrapper (e.g. ctypes) code used as a _thin_ wrapper to an underlying library may stick with its naming convention for functions/methods. (See the docs for Attocube stages for an example of this)
- Modules and packages should have short, all-lowercase names, e.g. `drivers`
- Use a `_leading_underscore` for non-public functions, methods, variables, etc.

- Module-level constants are written in `ALL_CAPS`

Strongly consider using a plugin for your text editor (e.g. vim-flake8) to check your PEP8 compliance.

1.6.3 Docstrings

Code in Instrumental is primarily documented using python docstrings. Specifically, we follow the numpydoc conventions.

1.6.4 Python 2/3 Compatibility

Currently Instrumental is developed and tested using Python 2.7, with an eye towards Python 3 compatibility. The ultimate goal is to to have a code base that runs unmodified on Python 2 and 3. Moving straight to 3 would be nice, but there is still much code in the universe that has not yet been ported.

There are a number of backwards-incompatible changes that occurred, but perhaps the biggest and peskiest for Instrumental was the switchover to using unicode strings by default. This is probably the largest source of Python 3 incompatibility in the existing code.

Other notable changes include:

- `print` is now a function, no longer a statement
- All division now uses "true" division (e.g. `3/4 == 0.75`). Use `//` to denote integer division (e.g. `3//4 == 0`)

To help alleviate these transition pains, you should use python's built-in `__future__` module. E.g.:

```
>>> from __future__ import division, print_function, unicode_literals
```

Some useful links about Py2/3 compatibility

- https://docs.python.org/3/howto/pyporting.html
- http://python-future.org/

i

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z