wanglib Documentation

Release dev

Thomas Baldwin

May 27, 2015

Contents

Experimental control utilities for the Wang lab.

When writing a script to control your experiment (or working interactively using the interpreter), use `wanglib` to make instrument control and data gathering easy.

CHAPTER 1

Obtaining and Installing

wanglib is hosted on Github. Follow the instructions there on how best to install the package and its dependencies.

CHAPTER 2

Contents

2.1 Instrument control with `wanglib.instruments`

Class-based interfaces to various scientific instruments.

These interfaces are designed to complement the low-level instrument talking already provided by PySerial (for RS232) and PyVISA (for GPIB). Each instrument object defined here wraps a `serial.Serial` or `visa.instrument` instance and uses its `write`/`read`/`ask` methods to accomplish common commands and readings specific to that instrument.

2.1.1 Example usage

Here's an example. We want to talk to an Agilent model 8648 RF signal generator using GPIB. We have PyVISA installed, and that makes GPIB talking a snap:

```
>>> from visa import instrument
>>> agilent = instrument("GPIB::18")
>>> agilent.ask("FREQ:CW?")
'200000000'
```

Three lines of code isn't bad, but it's cumbersome to write raw commands to the instrument all the time. Plus, the returned value is a string, not a number, and we'll have to convert it.

Fortunately, wanglib defines a class that handles all these commands for us.

```
>>> from wanglib.instruments.signal_generators import ag8648
>>> rf = ag8648(agilent)
>>> rf.freq
200.
```

What happened here? Well, the ag8648 class we imported from wanglib has 'wrapped' the agilent object we made before, and returned an object representing our signal generator. This object has a variable attached to it (an 'attribute') representing the frequency. Whenever we access this attribute, the object performs the GPIB query behind the scenes and converts the instrument response to a number in MHz.

We can also change the attribute:

```
>>> rf.freq = 110
>>> rf.freq
110.
```

Again, all queries and commands are being handled behind the scenes. This makes your scripts much more readable, and is a especially useful for interactive use. If all of your instruments are supported by wanglib, you can feasibly run your whole experiment from a live python interpreter.

For a list of all you can do with this rf object, do

```
>>> dir(rf)
```

And, as always in Python, use the help() function for information on any object, attribute, or method.

```
>>> help(rf)
>>> help(rf.freq)
>>> help(rf.blink)
```

If the ag8648 class doesn't have an attribute or method for a given instrument function, you can still send raw GPIB queries by accessing the original PyVISA object as a sub-object of the ag8648 instance.

```
>>> rf.bus.ask("FREQ:CW?")
'110000000'
```

In this example, we wrapped a PyVISA instrument, but that's not required. The low-level instrument we started with can be anything that has similar read(), write(), and ask() methods:

PyVISA visa.instrument

linux-gpib wanglib.linux_gpib.Gpib

prologix wanglib.prologix.instrument

It's also easy to add functionality to the class. The help() function tells you where to find the source file for any object on your computer.

2.1.2 Supported instruments

Lock-in amplifiers: `wanglib.instruments.lockins`

Lock-in amplifiers are commonly used for sensitive detection of a modulated voltage signal. They enable us to measure both the amplitude and phase of a signal, which we can access in either the cartesian (X, Y) or polar (R, phase) basis.

This module provides interfaces to two brands of lock-in, using two corresponding classes.

EG&G 5110 *wanglib.instruments.lockins.egg5110*

SRS 830 *wanglib.instruments.lockins.srs830*

Note: The methods implemented by these two classes are named the same, but don't always behave the same way. For example, the EG&G 5110 returns a 2-tuple with unit when the ADC ports are queried, but the SRS 830 always returns a figure in volts.

Warning: I'm working toward parity between the return value formats of the two classes. This will make it easier to switch between lock-ins, but will break existing code!

class `wanglib.instruments.lockins.`**`egg5110`**(*bus*)

An EG&G model 5110 lock-in.

Typically controlled over GPIB, address 12. Instantiate like:

```
>>> li = egg5110(plx.instrument(12))
```

where `plx` is a prologix controller. pyVISA instruments should also work fine.

autophase()
Automatically adjust the reference phase to maximize the X signal and minimize Y.

get_ADC(*n*)
Read one of the four ADC ports. Return value in volts.

get_phase()
Get current value of the phase, in degrees.

get_r()
Get current value of R, in volts.

get_sensitivity(*unit='V'*)
Get the current sensitivity, in Volts.

```
>>> li.get_sensitivity()
0.1
```

If the *unit* kwarg is specified, the value will be converted to the desired unit instead.

```
>>> li.get_sensitivity(unit='uV')
100000.
```

Using *unit=True* will return a value in a 2-tuple along with the most sensible unit (as a string).

```
>>> li.get_sensitivity(unit=True)
(100, 'mV')
```

get_timeconst()
Get the current time constant (as a 2-tuple).

get_x()
Get current value of X, in volts.

get_y()
Get current value of Y, in volts.

lights
Boolean. Turns the front panel lights on or off.

measure(*command*, *unit='V'*)
Measure one of the usual voltage signals (X, Y, or MAG).

```
>>> li.measure('X')
0.0014
```

Results are given in volts. To specify a different unit, use the `unit` kwarg.

```
>>> li.measure('X', unit='mV')
1.4
```

To skip this conversion, and instead return the result as a fraction of the sensitivity (what the manual calls "percent of full-scale"), specify `unit=None`:

```
>>> li.measure('X', unit=None)
.14
```

You will need to multiply by the sensitivity (in this example, 10mV) to get a meaningful number. To perform this multiplication automatically, specify `unit=True`:

```
>>> li.measure('X', unit=True)
(1.4, 'mV')
```

This returns a 2-tuple containing the measurement and the unit string ("V", "mV", etc.),

---

Note: to provide an answer in real units, the EG&G 5110 needs to be queried for its sensitivity on every single measurement. This can slow things down. If you need to make measurements quickly, and are using a fixed sensitivity, specify `unit=None` for speed.

---

phase
Get current value of the phase, in degrees.

r
Get current value of R, in volts.

sensitivities = {0: (100, 'nV'), 1: (200, 'nV'), 2: (500, 'nV'), 3: (1, 'uV'), 4: (2, 'uV'), 5: (5, 'uV'), 6: (10, 'uV'), 7: (20

sensitivity
Get the current sensitivity, in Volts.

```
>>> li.get_sensitivity()
0.1
```

If the *unit* kwarg is specified, the value will be converted to the desired unit instead.

```
>>> li.get_sensitivity(unit='uV')
100000.
```

Using *unit=True* will return a value in a 2-tuple along with the most sensible unit (as a string).

```
>>> li.get_sensitivity(unit=True)
(100, 'mV')
```

set_sensitivity(*code*)
Set the current sensitivity (Using a code).

set_timeconst(*code*)
Set the current time constant (Using a code).

timeconst
Get the current time constant (as a 2-tuple).

timeconsts = {0: (0, 'MIN'), 1: (1, 'ms'), 2: (3, 'ms'), 3: (10, 'ms'), 4: (30, 'ms'), 5: (100, 'ms'), 6: (300, 'ms'), 7: (1, 's'),

x
Get current value of X, in volts.

y
Get current value of Y, in volts.

class wanglib.instruments.lockins.**srs830**(*bus*)
A Stanford Research Systems model SR830 DSP lock-in.

Typically controlled over GPIB, address 8. Instantiate like:

```
>>> li = srs830(plx.instrument(8))
```

where `plx` is a prologix controller. pyVISA instruments should also work fine.

ADC_cmd = 'OAUX?%d'

ADC_range = (1, 2, 3, 4)

DAC_range = (1, 2, 3, 4)

get_ADC(*n*)
read one of the ADC ports. Return value in volts.

get_DAC(*n*)
read one of the DAC ports. Return value in volts.

get_r()

get_x()

get_y()

measure(*command*)
Measure one of the usual signals (X, Y, or MAG).

Results are given in units of volts or degrees.

measurements = {'Y': 2, 'X': 1, 'R': 3, 'MAG': 3}

r

set_DAC(*n*, *value*)
set one of the DAC ports. Provide value in volts.

x

y

Jobin-Yvon spectrometers: `wanglib.instruments.spex750m`

This module contains utilities for controlling Jobin-Yvon "SPEX" series monochromators over RS232 serial.

Classes defined:

- *spex750m* – implements standard Jobin-Yvon serial commands for simple spectrometers like the 750M.
- *triax320* – extends the `spex750m` class with the more sophisticated commands used by the triax.

Tutorial

both *spex750m* and *triax320* behave roughly the same way. In this tutorial we use the 750M for example. Instantiate like:

```
>>> beast = spex750m()
```

This opens a connection with the larger of the two spectrometers (the SPEX 750M.)

Classes are configured to look for the spectrometer at the serial port where they are normally plugged in. To specify a different serial port, just pass it as an optional parameter when instantiating. For example, the following should be equivalent:

```
>>> beast = spex750m(addr=1)
>>> beast = spex750m(addr="COM2")    # on windows
>>> beast = spex750m(addr="/dev/ttyS1")    # on linux
```

It's a good idea to check that the spectrometer has been calibrated. This is unneccessary for the Triax, but on the 750M, we should check it. Having instantiated it as `beast`, run

```
>>> beast.wl
790.
```

This queries the current wavelength in nm, which ought to match the value displayed in the window. If it doesn't, run

```
>>> beast.calibrate(800)
```

This will re-set the calibration to 800nm:

```
>>> beast.wl
800.
```

You can control the spectrometer using the methods documented below. For example, this performs a wavelength scan:

```
>>> beast = spex750m('/dev/ttyUSB0')
>>> beast.calibrate(800)
>>> beast.set_wavelength(750)
>>> for i in range(200):
...     beast.rel_move(0.5)
...     while beast.is_busy():
...         sleep(0.1)
...     result = measure_something()
...     print beast.get_wavelength(), result
```

API documentation

class wanglib.instruments.spex750m.**spex750m**(*addr=None*)

A class implementing standard Jobin-Yvon serial commands for simple spectrometers like the SPEX 750M.

By default, this class is configured to look for the 750M plugged into /dev/ttyUSB0 (the first port on the USB-serial adapter, when run under linux).

boot_status_check()

Check the boot status of the controller.

- * : Just Autobauded
- B : Boot Acknowledged
- F : Just Flashed

calibrate(*wl_value*)

Read the current wavelength from the window and pass it to this method (units of nm) to recalibrate the 750M.

get_wavelength()

Query the current wavelength

get_wl()

Query the current wavelength

init_hardware()

Initialize 750M hardware. I don't know why this works, I just copied Yan's LabView routine.

is_busy()

Check if the motors are busy.

reboot()

Reboot the controller if it's not responding

rel_move(*distance_to_move*)

Move the grating by the given number of nanometers.

set_wavelength(*wl*)

Move to the wavelength value specified. contingent on proper calibration, of course.

set_wl(*wl*)

Move to the wavelength value specified. contingent on proper calibration, of course.

wait_for_ok(*expected_bytes=1*)
Wait indefinitely for the 'o' status byte.

This function waits until a certain number of bytes (usually just one) are present on the bus. When that data arrives, the first bye is read and checked to make sure it's the "okay" status.

wavelength
Query the current wavelength

wl
Query the current wavelength

class wanglib.instruments.spex750m.**triax320**(*addr=None*)
A class implementing Jobin-Yvon serial commands for more advanced spectrometers like the TRIAX 320.

Instantiate as you would a spex750m. If you don't specify a serial port, this class will assume the spectrometer is attached to `/dev/ttyUSB1`, the second port on the USB-serial converter.

Unlike the 750M, the Triax

- has motorized entrance and exit slits
- zeroes its grating on power-up.

To perform a motor init on the triax, call *motor_init()*.

calibrate(*wl*)
Set wavelength of Triax without moving motor

entr_slit

exit_slit

get_wavelength()
Query the current wavelength

get_wl()
Query the current wavelength

motor_init()
Move all motors to their power-up (autocal) positions.

This zeroes the wavelength grating and both entrance and exit slits. Don't forget to reopen the slits after calling this.

set_wavelength(*wl*)
Move to a new wavelength

set_wl(*wl*)
Move to a new wavelength

slits
Return the entrance and exit slit settings together.

```
>>> beast.slits
(30., 30.)
```

Indicates that both slits are at 30. They can also be set together:

```
>>> beast.slits = (20,30)
>>> beast.slits
(20., 30.)
```

sets the entrance slit to 20. There is a shortcut for setting the slits to equal values:

```
>>> beast.slits = 20
>>> beast.slits
(20., 20.)
```

wavelength
Query the current wavelength

wl
Query the current wavelength

Motion control: `wanglib.instruments.stages`

Interfaces to motion controllers.

Newport motion cotroller command syntax has significant overlap between models, so I've written several classes to keep things as modular as possible.

To control a stage attached to a Newport ESP300 controller, which is on address 9 of a GPIB network run by a prologix controller 'plx', you might do this:

```
>>> from wanglib.instruments.stages import ESP300_stage
>>> esp = plx.instrument(9)
>>> example_stage = ESP300_stage(1, esp)
```

Here the 'esp' object is the prologix GPIB connection, but pyVISA instruments should also work.

To control another axis of the stage, just make another ESP300_stage instance using the same instrument object.

```
>>> other_stage = ESP300_stage(2, esp)
```

This creates another stage on axis 2.

This class also defines handy classes specific to actual stages that are on the table, and you probably want those. For example:

```
>>> from wanglib.instruments.stages import long_stage
>>> from wanglib.instruments.stages import short_stage
```

Will get two delay stages, with extra properties representing the delay in picoseconds, etc.

class `wanglib.instruments.stages.`**`ESP300_stage`**(*axisnum*, *bus=None*)
A single stage controlled by the ESP300.

The ESP300 is typically on GPIB address 9.

To use over RS232, use the following parameters:

baudrate 19200

rtscts True

term_chars '\r\n'

These settings are used by default if you simply pass the name of the serial port as a string:

```
>>> my_stage = ESP300_stage(1, '/dev/ttyUSB')
```

This will make an object corresponding to axis 1 of the stage.

For full control over the RS232 communication, provide a `Serial` instance instead of an address. For example:

```
>>> from wanglib.util import Serial
>>> esp = Serial('/dev/ttyUSB', baudrate=19200, timeout=10,
...              rtscts=1, log='esp300.log', term_chars="\r\n")
>>> my_stage = ESP300_stage(1, esp)
```

This will work the same as above, but also log command traffic to the file `esp300.log`.

busy
ask whether motion is in progress

cmd(*string*)
Prepend the axis number to a command.

define_home(*loc=None*)
Define the origin of this stage to be its current position.

To define the current position to be some number other than zero, provide that number in the loc kwarg.

encoder_resolution
Get the distance represented by one encoder pulse.

find_zero()
Place the zero 1mm from the hardware limit. This follows Yan's old labview routine.

get_max_velocity()
Get the maximum motor velocity.

get_pos()
Query the absolute position of the stage

get_velocity()
Get the current motor velocity.

move(*delta*)
Move the stage (relative move)

move_to_limit(*direction=-1*)
Move to the hardware limit of the stage.

By default, finds the negative limit. To find the positive limit, provide a positive number as the argument.

NOTE: sometimes, the ESP300 will time out and give up looking for the hardware limit if it takes too long to get there. So, try to get reasonably close to the limit before using this function.

on
Turn motor on/off.

pos
Query the absolute position of the stage

set_max_velocity(*val*)
Set the max motor velocity.

set_pos(*val*)
Set the absolute position of the stage

set_unit(*key*)
Set the unit label for a given axis, by index.

Unit labels ('mm', 'um', etc) have corresponding integer indices. Look these up in the _unit_labels dictionary.

set_velocity(*val*)
Set the motor velocity.

step_size
Get the distance represented by one motor step.

unit
Get the unit label for a given axis.

wait(*lag=0.5*)
Stop the python program until motors stop moving.

optionally, specify a check interval in seconds (default: 0.5)

class wanglib.instruments.stages.**MM3000_stage**(*axisnum*, *bus=None*)
A single stage controlled by the Newport MM3000 motion controller.

Firmware version: 2.2

The MM3000 is typically on GPIB address 8.

busy

cmd(*string*)
Prepend the axis number to a command.

define_home()
Define the origin of this stage to be its current position.

find_zero()
Place the zero 1mm from the hardware limit. This follows Yan's old labview routine.

get_pos()
Query the absolute position of the stage

motor_status(*bit=None*)
Request the MM3000 motor status byte.

Optionally, pick out a specific bit. bit 0: True if axis is moving bit 1: True if motor is *off* bit 2: True if direction of move is positive bit 3: True if positive travel limit is active bit 4: True if negative travel limit is active bit 5: True if positive side of home bit 6: always True bit 7: always False

move(*delta*)
Move the stage (relative move)

move_to_limit(*direction=-1*)
Move to the hardware limit of the stage.

By default, finds the negative limit. To find the positive limit, provide a positive number as the argument.

NOTE: sometimes, the ESP300 will time out and give up looking for the hardware limit if it takes too long to get there. So, try to get reasonably close to the limit before using this function.

on

pos
Query the absolute position of the stage

set_pos(*val*)
Set the absolute position of the stage

wait(*lag=0.5*)
Stop the python program until motors stop moving.

optionally, specify a check interval in seconds (default: 0.5)

class wanglib.instruments.stages.**delay_stage**(*axisnum*, *bus=None*)
Mixin class for stages used primarily to delay pulses.

Defines one extra feature: the 't' attribute, which is basically the position of the stage in picosecond units.

When mixing in, you need to define two extra attributes:

stage_length length of the stage, in its natural length units.

c speed of light, in natural length units per picosecond.

Important: make sure to call 'find_zero' on the stage before using t to control motion. Otherwise you might run out of range.

cmd(*string*)
Prepend the axis number to a command.

find_zero()
Place the zero 1mm from the hardware limit. This follows Yan's old labview routine.

get_pos()
Query the absolute position of the stage

get_t()
Convert stage position in mm to delay in ps

move(*delta*)
Move the stage (relative move)

move_to_limit(*direction=-1*)
Move to the hardware limit of the stage.

By default, finds the negative limit. To find the positive limit, provide a positive number as the argument.

NOTE: sometimes, the ESP300 will time out and give up looking for the hardware limit if it takes too long to get there. So, try to get reasonably close to the limit before using this function.

pos
Query the absolute position of the stage

set_pos(*val*)
Set the absolute position of the stage

set_t(*new_val*)

t
Convert stage position in mm to delay in ps

wait(*lag=0.5*)
Stop the python program until motors stop moving.

optionally, specify a check interval in seconds (default: 0.5)

class wanglib.instruments.stages.**long_stage**(*axisnum*, *bus=None*)

busy
ask whether motion is in progress

c = 0.3

cmd(*string*)
Prepend the axis number to a command.

define_home(*loc=None*)
Define the origin of this stage to be its current position.

To define the current position to be some number other than zero, provide that number in the loc kwarg.

encoder_resolution
Get the distance represented by one encoder pulse.

find_zero()
Place the zero 1mm from the hardware limit. This follows Yan's old labview routine.

get_max_velocity()
Get the maximum motor velocity.

get_pos()
Query the absolute position of the stage

get_t()
Convert stage position in mm to delay in ps

get_velocity()
Get the current motor velocity.

move(*delta*)
Move the stage (relative move)

move_to_limit(*direction=-1*)
Move to the hardware limit of the stage.

By default, finds the negative limit. To find the positive limit, provide a positive number as the argument.

NOTE: sometimes, the ESP300 will time out and give up looking for the hardware limit if it takes too long to get there. So, try to get reasonably close to the limit before using this function.

on
Turn motor on/off.

pos
Query the absolute position of the stage

set_max_velocity(*val*)
Set the max motor velocity.

set_pos(*val*)
Set the absolute position of the stage

set_t(*new_val*)

set_unit(*key*)
Set the unit label for a given axis, by index.

Unit labels ('mm', 'um', etc) have corresponding integer indices. Look these up in the _unit_labels dictionary.

set_velocity(*val*)
Set the motor velocity.

stage_length = 600

step_size
Get the distance represented by one motor step.

t
Convert stage position in mm to delay in ps

unit
Get the unit label for a given axis.

wait(*lag=0.5*)
Stop the python program until motors stop moving.

optionally, specify a check interval in seconds (default: 0.5)

class wanglib.instruments.stages.**short_stage**(*axisnum*, *bus=None*)

busy

c = 3000.0

cmd(*string*)
Prepend the axis number to a command.

define_home()
Define the origin of this stage to be its current position.

find_zero()
Place the zero 1mm from the hardware limit. This follows Yan's old labview routine.

get_pos()
Query the absolute position of the stage

get_t()
Convert stage position in mm to delay in ps

mm = 10000.0

motor_status(*bit=None*)
Request the MM3000 motor status byte.

Optionally, pick out a specific bit. bit 0: True if axis is moving bit 1: True if motor is *off* bit 2: True if direction of move is positive bit 3: True if positive travel limit is active bit 4: True if negative travel limit is active bit 5: True if positive side of home bit 6: always True bit 7: always False

move(*delta*)
Move the stage (relative move)

move_to_limit(*direction=-1*)
Move to the hardware limit of the stage.

By default, finds the negative limit. To find the positive limit, provide a positive number as the argument.

NOTE: sometimes, the ESP300 will time out and give up looking for the hardware limit if it takes too long to get there. So, try to get reasonably close to the limit before using this function.

on

pos
Query the absolute position of the stage

set_pos(*val*)
Set the absolute position of the stage

set_t(*new_val*)

stage_length = 1000000.0

t
Convert stage position in mm to delay in ps

wait(*lag=0.5*)
Stop the python program until motors stop moving.

optionally, specify a check interval in seconds (default: 0.5)

class wanglib.instruments.stages.**shorty_stage**(*axisnum*, *bus=None*)
Newport UTM100pp.1 stage, when plugged into ESP300

busy
ask whether motion is in progress

cmd(*string*)
Prepend the axis number to a command.

define_home(*loc=None*)
Define the origin of this stage to be its current position.

To define the current position to be some number other than zero, provide that number in the loc kwarg.

encoder_resolution
Get the distance represented by one encoder pulse.

find_zero()
Place the zero 1mm from the hardware limit. This follows Yan's old labview routine.

get_max_velocity()
Get the maximum motor velocity.

get_pos()
Query the absolute position of the stage

get_t()
Convert stage position in mm to delay in ps

get_velocity()
Get the current motor velocity.

initialize()

move(*delta*)
Move the stage (relative move)

move_to_limit(*direction=-1*)
Move to the hardware limit of the stage.

By default, finds the negative limit. To find the positive limit, provide a positive number as the argument.

NOTE: sometimes, the ESP300 will time out and give up looking for the hardware limit if it takes too long to get there. So, try to get reasonably close to the limit before using this function.

on
Turn motor on/off.

pos
Query the absolute position of the stage

set_max_velocity(*val*)
Set the max motor velocity.

set_pos(*val*)
Set the absolute position of the stage

set_t(*new_val*)

set_unit(*key*)
Set the unit label for a given axis, by index.

Unit labels ('mm', 'um', etc) have corresponding integer indices. Look these up in the _unit_labels dictionary.

set_velocity(*val*)
Set the motor velocity.

step_size
Get the distance represented by one motor step.

t
Convert stage position in mm to delay in ps

unit
Get the unit label for a given axis.

wait(*lag=0.5*)
Stop the python program until motors stop moving.

optionally, specify a check interval in seconds (default: 0.5)

class wanglib.instruments.stages.**thorlabs_Z612B**(*axisnum*, *bus=None*)
Thorlabs Z612B motorized actuator.

busy
ask whether motion is in progress

cmd(*string*)
Prepend the axis number to a command.

define_home(*loc=None*)
Define the origin of this stage to be its current position.

To define the current position to be some number other than zero, provide that number in the loc kwarg.

encoder_resolution
Get the distance represented by one encoder pulse.

find_zero()
Place the zero 1mm from the hardware limit. This follows Yan's old labview routine.

get_max_velocity()
Get the maximum motor velocity.

get_pos()
Query the absolute position of the stage

get_velocity()
Get the current motor velocity.

initialize()

move(*delta*)
Move the stage (relative move)

move_to_limit(*direction=-1*)
Move to the hardware limit of the stage.

By default, finds the negative limit. To find the positive limit, provide a positive number as the argument.

NOTE: sometimes, the ESP300 will time out and give up looking for the hardware limit if it takes too long to get there. So, try to get reasonably close to the limit before using this function.

on
Turn motor on/off.

pos
Query the absolute position of the stage

set_max_velocity (*val*)
Set the max motor velocity.

set_pos (*val*)
Set the absolute position of the stage

set_unit (*key*)
Set the unit label for a given axis, by index.

Unit labels ('mm', 'um', etc) have corresponding integer indices. Look these up in the _unit_labels dictionary.

set_velocity (*val*)
Set the motor velocity.

step_size
Get the distance represented by one motor step.

unit
Get the unit label for a given axis.

wait (*lag=0.5*)
Stop the python program until motors stop moving.

optionally, specify a check interval in seconds (default: 0.5)

RF signal generators: wanglib.instruments.signal_generators

Interfaces to Agilent RF signal generators.

class wanglib.instruments.signal_generators.**ag8648** (*bus*)
The Agilent 8648A/B/C/D RF signal generator.

These instruments are configured to work on GPIB address 18 by factory default. To instantiate one plugged in to prologix controller plx, do this:

```
>>> rf = ag8648(plx.instrument(18))
```

Other instrument objects (e.g. pyVISA) should also work.

Attributes:

on – boolean indicating whether RF output is on or off.

amp – RF output amplitude, in DBM.

The signal generator can be controlled remotely by setting these attributes of the instance.

```
>>> rf.amp
-5.0
>>> rf.amp = -10
>>> rf.amp
-10.0
```

amp
RF amplitude in dBm.

blink (*interval=1.0*)
Blink the RF output on and off with time.

Useful when aligning AOMs.

freq
RF frequency in MHz.

get_amp()
RF amplitude in dBm.

get_freq()
RF frequency in MHz.

get_on()
is RF output on?

get_pulse()
is pulse modulation enabled?

on
is RF output on?

pulse
is pulse modulation enabled?

set_amp(*val*, *unit='DBM'*)
Set the output amplitude. Assumes units of DBM.

Available units: DBM, MV, UV, MVEMF, UVEMF, DBUV, DBUVEMF.

set_freq(*val*, *unit='MHZ'*)
Set the RF frequency in MHz.

set_on(*val*)

set_pulse(*val*)

Newport Diode Lasers: wanglib.instruments.lasers

Interfaces to New Focus diode laser controllers.

class wanglib.instruments.lasers.**velocity6300**(*bus*)
A New Focus Velocity 6300 diode laser controller.

To instantiate, pass an instrument object to the constructor. e.g., for a controller with GPIB address 1, attached to a prologix GPIB controller:

```
>>> laser = velocity6300(plx.instrument(1, auto=False))
```

where `plx` is the prologix object. If you're using prologix, it's very important to turn off read-after-write!

To use with RS232, use `\r` as the termination character. For example:

```
>>> from wanglib.util import Serial
>>> laser = velocity6300(Serial('/dev/ttyUSB0', baudrate=19200, term_chars='\r'))
```

busy
is an operation in progress?

current
return current level in the diode (mA)

get_piezo()
Return piezo voltage as % of full-scale.

get_wl()
Get current wavelength of laser (nm)

on
is the laser on or off?

piezo
Return piezo voltage as % of full-scale.

power
return front-facet laser power level (mW)

set_piezo(*val*)
Set piezo voltage as % of full-scale.

set_wl(*val*)
Set current wavelength of laser (nm).

Can take a numerical value, or 'min' or 'max'

stop_tracking()
exit track mode to ready mode.

wl
Get current wavelength of laser (nm)

wl_max
max wavelength of this diode

wl_min
min wavelength of this diode

write(*cmd*)
Issue a command to the laser.

This takes care of two things:

- Formats the command with `@` if using RS-232
- Verifies that the laser responds with `OK`

Burleigh Wavemeter: wanglib.instruments.wavemeter

class wanglib.instruments.wavemeter.**Serial**

class wanglib.instruments.wavemeter.**burleigh**(*port='/dev/ttyUSB0'*)
encapsulates a burleigh wavemeter

display

display_masks = {64: 'wavelength', 128: 'deviation'}

get_display()

get_response()

get_unit()

get_wl(*strict=True*)
Get the current wavelength (or frequency).

If strict, cast approximate measurements to None.

parse(*response*)
parse the wavemeter's broadcast string

parse_code(*code*, *dic*)

purge()
purge the buffer of old measurements

query()
Request a single response string.

The first time the wavemeter receives this command, it will switch from broadcast to query mode.

unit

unit_masks = {9: 'nm', 18: 'cm-1', 36: 'GHz'}

wl
Get the current wavelength (or frequency).

If strict, cast approximate measurements to None.

Tektronix Oscilloscopes: `wanglib.instruments.tektronix`

Interfaces to Tektronix oscilloscopes.

class `wanglib.instruments.tektronix.`**TDS3000**(*bus=None*)
A Tektronix oscilloscope from the TDS3000 series.

Can be controlled over GPIB, RS-232, or Ethernet. Just pass an object with `write`, `read`, and `ask` methods to the constructor.

```
>>> from wanglib.util import Serial
>>> bus = Serial('/dev/ttyS0', rtscts=True)
>>> scope = tds3000(bus)
```

If using RS-232 (as above), be sure to use rtscts, and connect using a null modem cable. You will probably need to use the highest baud rate you can.

acquire = {}

acquire_restart()
Discards collected data and restarts acquisition.

data_source
Determines the default data curve returned by *get_curve()*.

Possible values include *CH1*, *CH2*, *CH3*, *CH4*, *MATH*, *MATH1* (same as *MATH*), *REF1*, *REF2*, *REF3*, and *REF4*.

get_curve(*source=None*)
Fetch a trace.

Parameters **source** – Channel to retrieve. Defaults to value of *data_source*. Valid channels are *CH1*, *CH2*, *CH3*, *CH4*, *MATH*, *MATH1* (same as *MATH*), *REF1*, *REF2*, *REF3*, or *REF4*.

Returns A numpy array representing the current waveform.

get_timediv()
Get time per division, in seconds.

Returns Seconds per division, as a floating-point number.

get_wfm(*source=None*)
Fetch a trace, scaled to actual units.

Parameters **source** – Channel to retrieve. Defaults to value of *data_source*. Valid channels are *CH1*, *CH2*, *CH3*, *CH4*, *MATH*, *MATH1* (same as *MATH*), *REF1*, *REF2*, *REF3*, or *REF4*.

Returns Two numpy arrays: *t* and *y*

is_active(*channel*)
Ask whether a given waveform is active

Parameters **channel** – *CH1*, *CH2*, *CH3*, *CH4*,

MATH, *MATH1* (same as *MATH*), *REF1*, *REF2*, *REF3*, or *REF4*.

save_wfm(*source*, *dest*)
Store a waveform locally on the oscilloscope.

Parameters

- **source** (*str*) – channel to transfer data from. *CH<x>*, *MATH<x>*, or *REF<x>*.
- **dest** (*str*) – Which one of the four *REF<x>* to save into.

set_timediv(*to*)
Set time per division, in seconds.

Parameters **to** (*float*) – Desired seconds per division. Acceptable values range from 10 seconds to 1, 2, or 4ns, depending on model, in a 1-2-4 sequence.

timediv
Get time per division, in seconds.

Returns Seconds per division, as a floating-point number.

2.2 Alternative GPIB drivers

If PyVISA is not installed, *wanglib* provides two alternative GPIB interfaces that emulate the behavior of `visa.instrument`.

2.2.1 Prologix controllers with `wanglib.prologix`

This module enables you to control various instruments over GPIB using low-cost Prologix controllers. The interface aims to emulate that of PyVISA, such that *wanglib.prologix.instrument* objects can be a drop-in replacement for `visa.instrument()`.

For example, the canonical PyVISA three-liner

```
>>> import visa
>>> keithley = visa.instrument("GPIB::12")
>>> print keithley.ask("*IDN?")
```

is just one line longer with *wanglib.prologix*:

```
>>> from wanglib import prologix
>>> plx = prologix.prologix_USB('/dev/ttyUSBgpib')
>>> keithley = plx.instrument(12)
>>> print keithley.ask("*IDN?")
```

This extra verbosity is necessary to specify which GPIB controller to use. Here we are using a Prologix GPIB-USB controller at `/dev/ttyUSBgpib`. If we later switch to using a Prologix GPIB-Ethernet controller, we would instead use

```
>>> plx = prologix.prologix_ethernet('128.223.xxx.xxx')
```

for our `plx` controller object (replace the ‘‘xxx‘‘es with the controller’s actual ip address, found using the Prologix Netfinder tool).

class `wanglib.prologix.`**`PrologixEthernet`**(*ip*)
Interface to a Prologix GPIB-Ethernet controller.

To instantiate, use the `prologix_ethernet` factory:

```
>>> plx = prologix.prologix_ethernet('128.223.xxx.xxx')
```

Replace the ‘‘xxx‘‘es with the controller’s actual ip address, found using the Prologix Netfinder tool.

`addr`
The Prologix controller can calk to one instrument at a time. This sets the GPIB address of the currently addressed instrument.

Use this attribute to set or check which instrument is currently selected:

```
>>> plx.addr
9
>>> plx.addr = 12
>>> plx.addr
12
```

`ask`(*query*, **args*, ***kwargs*)
Write to the bus, then read response.

`auto`
Boolean. Read-after-write setting.

The Prologix ‘read-after-write’ setting can automatically address instruments to talk after writing to them. This is usually convenient, but some instruments do poorly with it.

`instrument`(*addr*, ***kwargs*)
Factory function for *`instrument`* objects.

```
>>> plx.instrument(12)
```

is equivalent to

```
>>> instrument(plx, 12)
```

***addr* – the GPIB address for an instrument** attached to this controller.

`readall`()

`savecfg`
Boolean. Determines whether the controller should save its settings in EEPROM.

It is usually best to turn this off, since it will reduce wear on the EEPROM in applications that involve talking to more than one instrument.

`version`()
Check the Prologix firmware version.

`write`(*command*, *lag=0.1*)

class `wanglib.prologix.`**`PrologixUSB`**(*port='/dev/ttyUSBgpib'*, *log=False*)
Interface to a Prologix GPIB-USB controller.

To instantiate, specify the virtual serial port where the controller is plugged in:

```
>>> plx = prologix.prologix_USB('/dev/ttyUSBgpib')
```

On Windows, you could use something like

```
>>> plx = prologix.prologix_USB('COM1')
```

addr
The Prologix controller can calk to one instrument at a time. This sets the GPIB address of the currently addressed instrument.

Use this attribute to set or check which instrument is currently selected:

```
>>> plx.addr
9
>>> plx.addr = 12
>>> plx.addr
12
```

ask(*query*, **args*, ***kwargs*)
Write to the bus, then read response.

auto
Boolean. Read-after-write setting.

The Prologix 'read-after-write' setting can automatically address instruments to talk after writing to them. This is usually convenient, but some instruments do poorly with it.

instrument(*addr*, ***kwargs*)
Factory function for *instrument* objects.

```
>>> plx.instrument(12)
```

is equivalent to

```
>>> instrument(plx, 12)
```

***addr* – the GPIB address for an instrument** attached to this controller.

readall()

savecfg
Boolean. Determines whether the controller should save its settings in EEPROM.

It is usually best to turn this off, since it will reduce wear on the EEPROM in applications that involve talking to more than one instrument.

version()
Check the Prologix firmware version.

write(*command*, *lag=0.1*)

class wanglib.prologix.**instrument**(*controller*, *addr*, *delay=0.1*, *auto=True*)
Represents an instrument attached to a Prologix controller.

Pass the controller instance and GPIB address to the constructor. This creates a GPIB instrument at address 12:

```
>>> plx = prologix_USB()
>>> inst = instrument(plx, 12)
```

A somewhat nicer way to do the second step would be to use the *instrument()* factory method of the Prologix controller:

```
>>> inst = plx.instrument(12)
```

Once we have our instrument object `inst`, we can use the *ask()* and *write()* methods to send GPIB queries and commands.

ask(*command*)
Send a query the instrument, then read its response.

Equivalent to *write()* then *read()*.

For example, get the 'ID' string from an EG&G model 5110 lock-in:

```
>>> inst.ask('ID')
'5110'
```

Is the same as:

```
>>> inst.write('ID?')
>>> inst.read()
'5110'
```

delay = 0.1

read()
Read a response from an instrument.

write(*command*)
Write a command to the instrument.

wanglib.prologix.**prologix_USB**(*port='/dev/ttyUSBgpib'*, *log=False*)
Factory for a Prologix GPIB-USB controller.

To instantiate, specify the virtual serial port where the controller is plugged in:

```
>>> plx = prologix.prologix_USB('/dev/ttyUSBgpib')
```

On Windows, you could use something like

```
>>> plx = prologix.prologix_USB('COM1')
```

wanglib.prologix.**prologix_ethernet**(*ip*)
Factory function for a Prologix GPIB-Ethernet controller.

To instantiate, specify the IP address of the controller:

```
>>> plx = prologix.prologix_ethernet('128.223.xxx.xxx')
```

2.2.2 linux_gpib compatibility layer: wanglib.linux_gpib

Utilities for computers using linux-gpib.

linux-gpib (http://linux-gpib.sourceforge.net/) is an open-source driver for various GPIB cards. It includes two python modules:

- the low-level *gpib* module
- an object-oriented *Gpib* module.

The Gpib module defines a *Gpib* class representing individual instruments. This module modifies that class to make it behave a little more like PyVISA's *instrument* class, for better compatibility with the rest of wanglib.

This will only work if your linux-gpib installation has been patched with the following enhancement:

http://sourceforge.net/tracker/?func=detail&aid=3437534&group_id=42378&atid=432942

This should apply to any linux-gpib released since January 2012.

class `wanglib.linux_gpib.`**Gpib**
Extension of the linux-gpib Gpib class to act more like a PyVISA instrument object.

ask(*query*)
Write then read.

Shadows the usual Gpib.ask() method, which does something weird.

read(**args*, ***kwargs*)
Read from Gpib device, stripping trailing space.

2.3 Improved interactive plotting with `wanglib.pylab_extensions`

This module contains some useful extensions to the pylab interface.

2.3.1 Plotting while acquiring data

To plot while acquiring data, you will need to implement your data gathering using Python generators. A generator is like a Python function that, rather than returning data all at once (with the `return` statement), returns it point by point (with the `yield` statement).

Suppose we have a `spex` spectrometer object, and a `lockin` object that we are using to detect the signal at the output slit of the spectrometer. Here is an example of a generator we might use to scan a spectrum, while yielding values along the way:

```
def scan_wls(wls):
    for wl in wls:
        spex.set_wl(wl)
        sleep(0.1)
        val = lockin.get_x()
        yield wl, val
```

Note: This pattern is so common that a shorthand is provided for it in *`wanglib.util.scanner()`*.

Then, if we wanted to scan from 800nm to 810nm, we would do

```
scan = scan_wls(numpy.arange(800, 810, 0.1))
for x,y in scan:
    print x,y
```

This will print the data to STDOUT, but we could also:

- save it to a python list using list comprehensions
- save it as a numpy object using `numpy.fromiter()`
- plot it progressively using *`wanglib.pylab_extensions.live_plot.plotgen()`*

The `plotgen` function reads from your generator, plotting data as it goes along. In the case of our `scan_wls` example above, we can view the spectrum as it gets collected:

```
wls = arange(800, 810, 0.1))
plotgen(scan_wls(wls))
# ... measures data, plotting as it goes along ...
```

After your generator yields its last value, it will return the complete array of measured X and Y values. Sometimes we want to do extra things with that:

```
_,ref = plotgen(scan_wls(wls))    # measure reference spectrum and plot it
_,trn = plotgen(scan_wls(wls))    # measure transmission spectrum and plot it
plot(wls, log(ref/trn), 'k--')    # plot absorption spectrum
```

Additionally, `plotgen` can generate multiple lines. Say we wanted to plot both the X and the Y quadrature from our lockin. We'd write our generator like:

```
def scan_wls(wls):
    for wl in wls:
        spex.set_wl(wl)
        sleep(0.1)
        x = lockin.get_x()
        y = lockin.get_y()
        yield wl, x, wl, y
```

This is the same as before, but we're yielding two X,Y pairs: one with the X quadrature `(wl, x)` and one with the Y quadrature `(wl, y)`. `plotgen` recognizes this and plots two lines:

```
_,x,_,y = plotgen(scan_wls(wls))
```

By default, `plotgen` plots these on the same axes, but sometimes that doesn't make sense. For example, if we're measuring the signal magnitude (in Volts) and phase (in degrees), then those two units have nothing to do with each other, and we should plot them on separate axes.

```
def scan_wls(wls):
    for wl in wls:
        spex.set_wl(wl)
        sleep(0.1)
        r = lockin.get_r()
        p = lockin.get_phase()
        yield wl, r, wl, p

# create two axes and pass them to plotgen explicitly
ax1 = pylab.subplot(211)
ax2 = pylab.subplot(212)
plotgen(scan_wls(wls), ax=(ax1,ax2))
```

Finally, you can limit the length of the plotted lines using the `maxlen` parameter. This is useful for generators which yield infinitely - such as monitoring the last five minutes of a signal.

```
def monitor_signal():
    start = time()
    while True:
        yield time() - start, lockin.get_phase()
        sleep(0.1)
```

Note: This pattern is so common that a shorthand is provided for it in *`wanglib.util.monitor()`*.

This will yield about 10 points per second forever. To cut it short after about 5 minutes of data, try this:

```
plotgen(monitor_signal(), maxlen=10*60*5)
```

Full documentation for `plotgen`:

`wanglib.pylab_extensions.live_plot.`**`plotgen`**(*gen*, *ax=None*, *maxlen=None*, ***kwargs*)

Take X,Y data from a generator, and plot it at the same time.

Parameters

- **gen** – a generator object yielding X,Y pairs.
- **ax** – an axes object (optional).
- **maxlen** – maximum number of points to retain (optional).

Returns an array of the measured X and Y values.

Any extra keyword arguments are passed to the plot function.

`gen` can yield any even number of values, and these are interpreted as a set of X,Y pairs. That is, if the provided generator yields 4 values each time, plotgen will plot two lines - with the first line updated from the [0:2] slice and the second line updated from the [2:4] slice.

`ax` is the axes object into which the line(s) are plotted. By default, the current axes. `ax` can also be a tuple of axes objects, as long as the tuple is the same length as the number of lines being plotted. Each line is then plotted into the corresponding axes.

`maxlen`, when provided, limits the buffer size. When the plotted lines each grow to this number of points, the oldest data points will start being trimmed off the line's trailing end.

2.3.2 Saving/clearing traces

`wanglib.pylab_extensions.misc.`**apply_mask**(*line*, *mask*)
mask x and y (to remove bad points, etc).

`wanglib.pylab_extensions.misc.`**apply_offset**(*line*, *offset*)
move the line up or down

`wanglib.pylab_extensions.misc.`**apply_reference**(*line*, *ref*)
apply reference data (for absorption spectra)

`wanglib.pylab_extensions.misc.`**bll**(*index=-1*, *lag=0.3*)
blink the last line, identifying it

`wanglib.pylab_extensions.misc.`**cll**(*index=-1*)
clear last line. removes the last line from the figure.

To remove a different line, specify the index.

`wanglib.pylab_extensions.misc.`**dualtick**(*func*)
Decorator for dual-tick functions.

A dual-tick function is a function that, when called on an axis, adds a second set of tick to it in different units. This decorator creates such functions when applied to the unit conversion function.

For example:

```
>>> @dualtick
>>> def eV(wl):
>>>     return 1240. / wl
```

Now, when working with plots of spectral data in units of nm, calling

```
>>> eV()
```

will add a second axis along the top in units of eV. To explicitly apply to some other axis `ax`, use `eV(ax)`. Returns a reference to the twiny axis that was made.

Another example, for a delay stage:

```
>>> @dualtick
>>> def ns(pos):
>>>     c = 300.  # mm / ns
>>>     zd = 521. # mm
>>>     return 2 * (zd - pos) / c
```

When plotting delay stage data enumerated in mm, this function will add an axis in ps delay from the zero-delay point at 521mm.

wanglib.pylab_extensions.misc.**gll**(*index=-1*, *blink=True*)
Get last line.

Retrieves x,y data of the last line and returns them as a tuple of two numpy arrays.

To get a different line, specify the index.

wanglib.pylab_extensions.misc.**relim**(*line*)
redraw the line's figure to include the line.

wanglib.pylab_extensions.misc.**sll**(*fname*, *index=-1*, *blink=True*)
Save last line.

Saves x,y data of the last line, in .npy format. Specify the file name.

To save a different line, specify the index.

2.3.3 Density plots

wanglib.pylab_extensions.density.**density_plot**(*two_dimensional*, *horiz*, *vert*, *ax=None*, ***kwargs*)
Display a 2D density plot - like imshow, but with axes labels corresponding to the two axes provided.

This will only be accurate for regularly-spaced x and y axes (e.g., as generated by arange or linspace).

Parameters

- **two_dimensional** – data to plot. shape: (M, N)
- **horiz** – x-axis. should have length N.
- **vert** – y-axis. should have length M.
- **ax** – axis upon which to plot.

Keyword arguments are passed to imshow.

2.4 Miscellaneous utilities — wanglib.util

This file provides useful utilities for the wanglib package.

exception wanglib.util.**InstrumentError**
Raise this when talking to instruments fails.

class wanglib.util.**Serial**(**args*, ***kwargs*)
Extension of PySerial's serial.Serial class that implements a few extra features:

•an *ask()* method

•a *readall()* method

•auto-appended termination characters

•in/out logging.

To log whatever's written or read to a serial port, pass a filename into the `log` kwarg:

```
>>> port = Serial('/dev/ttyS0', log='wtf.log')
```

To automatically append a newline to each command, specify `term_chars`:

```
>>> port.term_chars = '/n'
```

This can also be supplied as a keyword argument.

ask (*query*, *lag=0.05*)
Write to the bus, then read response.

This doesn't seem to work very well.

read (*size=1*)

readall (*term_chars=None*)
Automatically read all the bytes from the serial port.

if `term_chars` is set, this will continue to read until the terminating bytes are received. This can be provided as a keyword argument.

start_logging (*fname*)
start logging read/write data to file.

write (*data*)

wanglib.util.**averager** (*func*, *n*, *lag=0.1*)
Given a function `func`, returns an implementation of that function that just repeats it `n` times, and returns an average of the result.

Parameters

- **func** (*function*) – function returning a measurement
- **n** (*int*) – number of times to call `func`.
- **lag** (*float*) – seconds to sleep between measurements.

Returns the average of the `n` measurements.

This is useful when scanning. For example, if scanning a spectrum with the lockin like so:

```
>>> gen = scanner(wls, set=tr.set_wl, get=li.get_x)
```

We can implement a version that averages three lockin measurements with a 0.3s delay like so:

```
>>> acq = averager(li.get_x, 3, lag=0.3)
>>> gen = scanner(wls, set=tr.set_wl, get=acq)
```

wanglib.util.**gaussian** (*p*, *x*)
gaussian function.

p is a 4-component parameter vector defining:

```
0 -- a baseline offset
1 -- the area between curve and baseline
2 -- the location of the maximum
3 -- the standard deviation
```

wanglib.util.**monitor** (*function*, *lag=0.3*, *absolute=False*)
Periodically yield output of a function, along with timestamp. Compatible with *wanglib.pylab_extensions.live_plot.plotgen()*.

Parameters

- **function** (*function*) – function to call
- **lag** (*float*) – interval between calls to `function` (default 0.3 seconds)
- **absolute** (*boolean*) – if True, yielded x values are seconds since epoch. otherwise, time since first yield.

Returns a generator object yielding t,y pairs.

`wanglib.util.`**notraceback**(**args*, ***kwds*)

Context manager to swallow keyboard interrupt.

Execute any infinitely-looping process in this context, like:

```
>>> from time import sleep
>>> with notraceback():
...     while True:
...         sleep(0.1)
```

If you are planning to interrupt it anyway then you are not interested in the traceback and this prevents your output from being cluttered.

`wanglib.util.`**num**(*string*)

convert string to number. decide whether to convert to int or float.

`wanglib.util.`**save**(*fname*, *array*)

Save a Numpy array to file.

Parameters

- **fname** – Filename, as a string
- **array** – Numpy array to save.

Unlike `numpy.save()`, this function will raise ValueError if overwriting an existing file.

class `wanglib.util.`**saver**(*name*, *verbose=False*)

Sequential file saver.

after initializing *saver* with the base filename, use the *save()* method to save arrays to sequentially-numbered files.

```
>>> s = saver('foo')
>>> s.save(arange(5)) # saves to 'foo000.npy'
>>> s.save(arange(2)) # saves to 'foo001.npy'
```

save(*array*)

Save an array to the next file in the sequence.

`wanglib.util.`**scanner**(*xvals*, *set*, *get*, *lag=0.3*)

Generic scan generator - useful for spectra, delay scans, whatever. Compatible with *wanglib.pylab_extensions.live_plot.plotgen()*.

Parameters

- **xvals** (*iterable*) – values of x over which to scan.
- **set** (*function*) – Function to call on each step that advances the independent variable to the next value of `xvals`. This function should take that value as an argument.
- **get** (*function*) – Function to call on each step that performs the measurement. The return value of this function should be the measurement result.
- **lag** (*float*) – seconds to sleep between setting and measuring

Returns a generator object yielding x,y pairs.

Example: while scanning triax wavelength, measure lockin x

```
>>> from triax.instruments.lockins import egg5110
>>> from triax.instruments.spex750m import triax320
>>> from wanglib.pylab_extensions import plotgen
>>> tr = triax320()
>>> li = egg5110(instrument(plx,12))
>>> wls = arange(770, 774, .1)
>>> gen = scanner(wls, set=tr.set_wl, get=li.get_x)
>>> result = plotgen(gen)
```

Sometimes we will want to set/measure an attribute of an object on each step, instead of calling a method. In this case, we can provide an (object, attribute_name) tuple in lieu of a function for `set` or `get`. For example, in place of the `gen` used above, we could do:

```
>>> gen = scanner(wls, set=(tr,'wl'), get=(li,'x'))
```

Avoid this if you can, though.

`wanglib.util.`**`sciround`**(*number*, *sigfigs=1*)
Round a number to a desired significant figure precision.

```
>>> sciround(.000671438, 3)
.000671
```

`wanglib.util.`**`show_newlines`**(*string*)
replace CR+LF with the words "CR" and "LF". useful for debugging.

2.5 Jobin-Yvon CCD client — `wanglib.ccd`

Client routines for use with the CCD-2000 camera (generally attached to the Spex 750M spectrometer). These utilities talk to the CCD server LabView program running on the old computer over TCP/IP.

To configure the CCD server, refer to README file on the desktop of the CCD controller computer.

2.5.1 Command-line invocation

For a simple live display from the CCD, invoke this module as a script:

```
$ python -m wanglib.ccd --ip 128.223.xxx.xxx 800
```

where 800 is the center wavelength of the grating (as read from the window). You will need to specify the real IP address of the CCD server using the `--ip` flag.

A more sophisticated GUI for the CCD is available. This program can save data, zoom in and out, and move the grating on the Spex 750M.

2.5.2 Client library

To integrate the CCD client into your own script, use *`labview_client`*.

class `wanglib.ccd.`**`labview_client`**(*center_wl*, *host=None*, *port=3663*)
TCP client for Tim's labview ccd server.

Instantiate like so:

```
>>> ccd = labview_client(700, '128.223.xxx.xxx')
```

where `121.223.xxx.xxx` is the IP address of the computer running the Labview server, and 700 is the current wavelength of the spectrometer in nanometers (read from the window).

This info is needed because the labview program calculates wavelength values (from dispersion calibration info) on the server-side. Tye Hetherington wrote that sub-routine.

This client implements no control whatsoever of the SPEX 750m spectrometer to which the CCD is attached. For proper wavelength and dispersion info, you'll need to keep the client informed.

Whenever you move the spectrometer, set `center_wl` attribute to match:

```
>>> ccd.center_wl = 750
```

To get a spectrum, use *get_spectrum()*.

connect()
Establish a connection with the labview server.

If the labview program is ever stopped and restarted (as it should be when not taking data, to avoid wearing out the shutter), this should be called to reestablish the connection.

get_spectrum()
Takes a shot on the CCD.

Returns a 2-tuple `(wl, ccd)`.

`wl`: a 1-D array of the horizontal (wavelength) axis. `ccd`: a 2-D array of CCD counts.

To collapse `ccd` into a 1D array matching `wl`, sum over axis 0:

```
>>> wl,ccd = clnt.get_spectrum()
>>> line, = pylab.plot(wl,ccd.sum(axis=0))
```

CHAPTER 3

To Do

I still haven't documented the following:

`wanglib.grating` a bunch of old stuff for making SLM gratings/pulse shapers

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

W

A

B

C

D

E

F

G

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y