PyCMDS documentation

Contents

Todo list

1 Introduction

PyCMDS is a program designed to run frequency domain (multi-resonant) coherent multidimensional spectroscopy (MR-CMDS) experiments. PyCMDS provides a full stack solution for all of the software capabilities required in MR-CMDS:

- Basic hardware motion
- Hardware calibration
- Controlling and addressing detection devices
- Active and passive correction
- Multidimensional scanning
- Data recording and processing

Experimental MR-CMDS needs software that can be rapidly updated as new hardware, devices, and experimental strategies are developed. PyCMDS is built modularly using object oriented design principles. Due to this modularity, PyCMDS can be easily customized to the ever-changing needs of the MR-CMDS experimentalist.

In the Wright Group, PyCMDS replaces the old acquisition softwares 'ps control', written by Kent Meyer and 'Control for Lots of Research in Spectroscopy' written by Schuyler Kain.

This document is meant as a high-level reference for users of PyCMDS. Because PyCMDS is intended to be a platform for development, this document does attempt to familiarize the reader with the internal structure of the program where appropriate. Of course, more information can be found in comments and docstrings within the code itself. PyCMDS is written in python using open source libraries where possible. PyCMDS itself is licensed under the open-source MIT license which allows anyone to use, modify, and distribute the program for any reason.

1.1 Instillation

Dependencies:

1. comtypes
2. numpy
3. matplotlib version 1.4 or later
4. psutil
5. pyserial
6. pyvisa
7. PyQT4
8. pyqtgraph
9. scipy
10. WrightTools version 2.2 or later

Hardware/device specific dependencies:

1. National Instruments DAQ cards
   (a) PyDAQmx
2. PMC motors
   (a) MCAPI

Optional:

1. imageio version 1.6 or later (also need ffmpeg)
2. PyDrive version 1.2.1 or later
3. SlackClient
4. slacker

1.2 Terminology & Design

After starting PyCMDS a full-screen window will appear, as in Figure 1. This single window contains the entire PyCMDS graphical user interface (GUI). The GUI is organized into a series of tabs and vertical scroll-areas. There are four top-level elements in the GUI, two rows and two columns. In the left column there is the SHUT DOWN button and the manual hardware control / hardware status scroll area. In the right column there is the program status display and the main tab widget.

The main tab widget has six top-level tabs:

1. Program
2. Hardware
3. Devices
4. Autonomic
5. Sonomic
6. Plot

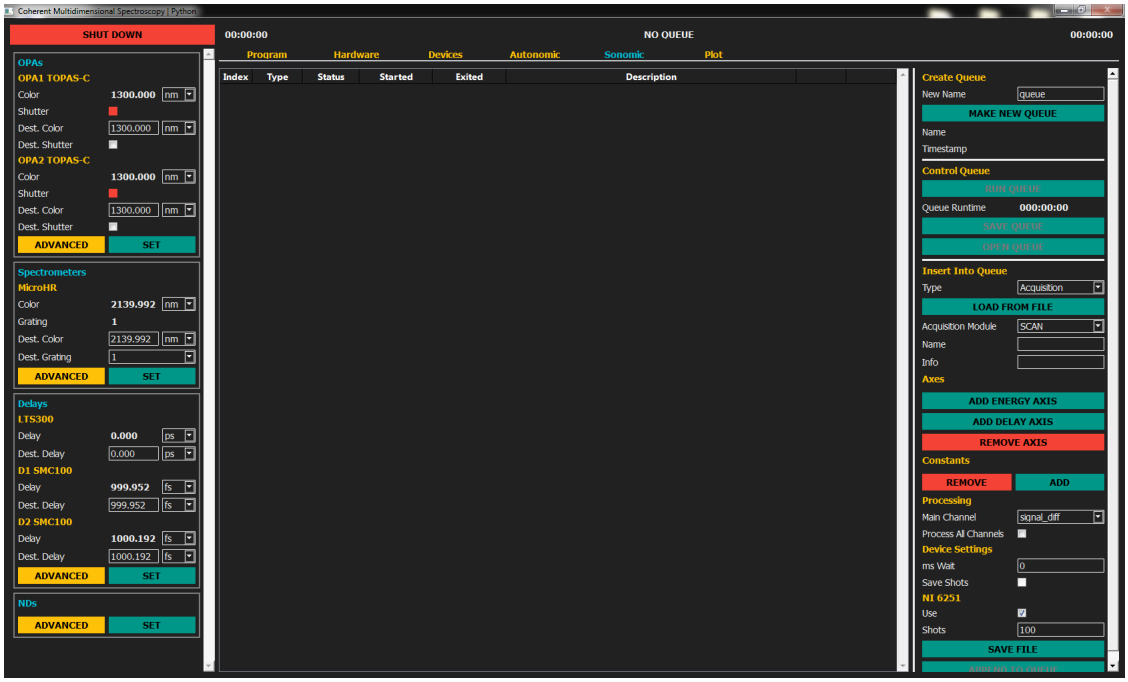

Figure 1: PyCMDS immediately after startup.

2 Overview

2.1 Startup

PyCMDS is started by running in python the `PyCMDS.py` module, located in the main directory. Once started, a long chain of operations must occur to start the program. Many of these operations are order dependent. These must be treated delicately due to PyCMDS' semi-synchronous threaded design. An outline of PyCMDS startup is presented below.

1. `PyCMDS_ui` ensures that folders not synced through git are present (only important when running for the first time after install / upgrade)
2. `PyCMDS_ui` imports `project.project_globals`
3. `project.project_globals` imports `project.ini_handler`
4. Project globals are instantiated, including `logger`
5. `logger` logs PyCMDS startup (from `PyCMDS_ui`)
6. `PyCMDS_ui` defines `__version__`
7. `PyCMDS_ui.main` is executed, instantiating `PyCMDS_ui.MainWindow` and the master PyQT app.
8. In `PyCMDS_ui.MainWindow.__init__`, `PyCMDS_ui._create_main_frame` is called, defining the top-level gui elements of PyCMDS (the module combobox / container, the hardware container, the large tab structure, the shut-down button, the progress bar)
9. In `PyCMDS_ui.MainWindow.__init__`, `PyCMDS_ui.MainWindow._initialize_hardware` is executed
   (a) `opas.opas`
   (b) `spectrometers.spectrometers`
   (c) `delays.delays`
   (d) `nds.nds`
   (e) `daq.daq`
10. In `PyCMDS_ui.MainWindow.__init__`, `PyCMDS_ui.MainWindow._initialize_widgets` is executed
    (a) `coset.coset`
11. In `PyCMDS_ui.MainWindow.__init__`, `PyCMDS_ui.MainWindow._load_modules` is executed - scan thread is created, and then scan modules are imported according to their order of appearance in `modules\modules.ini`
12. In `PyCMDS_ui.MainWindow.__init__`, `PyCMDS_ui.MainWindow._load_witch` is executed - the slack bot is created and the slack poll timer is started
13. Startup is complete - `logger` logs that `PyCMDS_ui.MainWindow.__init__` is complete

3 Hardware

Each `driver` object needs to have a series of hooks for PyCMDS to grab. The following is a minimum viable `Driver` and `GUI` class for PyCMDS.

3.1 OPAs

3.1.1 OPA800

3.1.2 TOPAS-C

TOPAS API error codes

Code	Meaning
0	No error
1	Unknown error
2	No TOPAS USB devices found
3	Invalid device instance
4	Invalid device index
5	Buffer too small
6	Failed to get TOPAS USB serial number
7	Device already opened
8	Device failed to open
9	USB communication channel failed to open
10	USB read error
11	Motor configuration failed to load
12	Configuration file doesn't match board configuration
13	Transmission of parameters failed
14	Device with this serial number not found
15	Invalid interface card type
16	Device has not been opened
17	USB command failed to receive response
18	Wavelength cannot be set
19	Invalid motor index
20	Function is not supported by LPT card
21	Invalid stage code
22	Invalid stage code
23	Tuning curve file failed to load
24	Tuning curve file read error
25	Wrong tuning curve file version
26	Wrong tuning curve type
27	Invalid number of motors
28	Invalid number of interactions
29	OPA type mismatch
30	Invalid wavelength
31	Invalid grating motor index in OPA tuning file
32	Tuning curve type mismatch
33	Configuration file not found
34	Invalid wavelength for this combination

The TOPAS-C units have DRM which lock them in software to only support certain tuning interactions. This means that to do certain operations (sum frequency signal, fourth harmonic idler etc) we need to let the TOPAS drivers know that we are 'allowed' to do such interactions using special keys.

The TOPAS motors do not have encoders and need to be homed periodically. The Light Conversion motor homing procedure is as follows:

1. Send the motor to the left reference switch.
2. Send the motor to 400 steps away from left reference switch.
3. Go to the left reference switch again, but at half speed.

4. Send the motor to 400 steps away from the left reference switch.
5. Tell the driver to call the current motor position zero steps.

3.2 Delays

In addition to the attributes and methods shared by all drivers, delay-type drivers must posses the following:

Methods:

1. `set_zero(zero)`

Attributes:

1. `zero_position` (`pc.Number`)

3.3 Newport SMC100

3.4 Thorlabs LTS300

3.5 Aerotech 101SMC2EN

Step size about 1 micron. Resolution as a delay stage about 5 fs.

Serial number: 108022

Motor model number: 101SMC2EN

Motor type: Dynacron

Controller model number: DM4005-P-A-60-F5

4 Devices

Like scan hardware, PyCMDS' DAQ is designed modularly. PyCMDS defines certain properties that must be achieved...

Here the individual instances are called 'device'.

Devices may have shots-level communication with PyCMDS.

Readings from DAQ devices are stored in memory as NumPy Arrays. See **VanderWalt2011** for more information.

4.1 NI PCI-6251

The NI PCI-6251 is a multichannel data acquisition card sold by National Instruments with the following properties.

analog channels	16
samples per second	1.25 million (1 channel), 1 million (multichannel)
resolution	16 bit
dynamic range	$\pm$ 10 V

The card is much faster than a 1 KHz laser, so many samples can be taken during each laser shot. This allows us to 'simultaneously' monitor multiple signals at once in PyCMDS.

In PyCMDS, parameterization of the NI 6251 is broken up into three timescales: samples, shots and values. Each timescale feeds into the one following it via a processing procedure. These procedures and the parameters that are exposed to users through the GUI and backed of PyCMDS are discussed here.

On the samples level,

In NI documentation...

http://digital.ni.com/public.nsf/allkb/42484E84DA98053686256D32006E0494

4.2 Matt's InGaAs Array

. . . see Matt's documentation . . .

5 Autonomic

The autonomic system controls automatic motion of hardware in PyCMDS. This includes active correction like spectral delay correction or active power compensation. The autonomic system is built on a set of rules and relationships between hardware. It constantly works behind the scenes to ensure that the rules and relationships are held.

5.1 Coset

Multiple codependent coset - it's not clear what will happen...

6 Somatic

The somatic system allows users of PyCMDS to do experiments. MR-CMDS experiments involve hardware motion, signal detection, and data processing. The experimental capabilities of PyCMDS require the coordinated handling of the `Hardware` objects described in Section 3 and the `Device` objects described in Section 4. Hardware motion and signal detection take time – MR-CMDS experiments can require hours or even days of continuous acquisition. The lasers and optomechanic systems used in MR-CMDS have limited usable lifetimes after alignment & calibration. To ensure that valuable instrument time can be used fully, PyCMDS has been designed using a queue structure.

A queue is a list of items that are addressed sequentially (first in, first out). Since a queue can be filled with many acquisitions, the queue structure is a very convenient design strategy to ensure that PyCMDS is always acquiring while the instrument remains calibrated.

The somatic system is organized into a triply-nested structure:

1. queue
2. acquisition
3. scan

A scan is an automatic traversal of experimental space. It involves the coordinated motion of hardware(s) and collection of device signal(s). Scans are done for experiment and calibration. This section describes how PyCMDS allows users to define scans, and how scans are accomplished within the software.

Users interface with PyCMDS' scan capabilities through acquisition modules. Each acquisition modules is set up to do allow for different kinds of scans. Acquisition modules may be very general (SCAN, MOTORTUNE) or quite specific (TUNE TEST, AUTOTUNE). PyCMDS endeavors to make it easy for end users to write their own acquisition module should they find the default ones unsatisfactory.

6.1 Acquisition Modules

Pre-wait methods.

6.2 Scans

Within the backend of PyCMDS, all scan modules are sequestered within the generalized scan system. The scan system defines a general set of rules that are true of possible scans within PyCMDS. In principle one could create a scan module that exposes the entire capabilities of the scan system to the user. Such a scan module would be too unwieldy for practical use. Because all scans live within the scan system, scans *cannot* be entirely arbitrary in PyCMDS. This means that it is possible to write automatic processing tools such as WrightTools that accept any PyCMDS output.

This section will start by describing the exact requirements of the scan system. It will describe the capabilities (and limitations) that the general rule set imposes. Next the details of scan module construction will be laid out. Then it will contain a complete discussion of what happens when a scan is run in PyCMDS. This will discuss the formating details of PyCMDS savefiles. Last, some words about choice scan modules.

In PyCMDS, scan modules do not directly address the hardware and device objects during a scan. Instead, scan modules define the properties of the desired scan and hand those properties to the scan handler (located in modules/scan.py) to carry out. Scan properties are defined in terms of axes and constants.

The scan system is designed with multidimensional scans in mind, although one dimensional scans are still possible. All scans must be **regular**, meaning that all other axes are **entirely** explored for each axis coordinate. For instance, in a 2D frequency ($\omega 1$ vs $\omega 2$) scan, for each $\omega 1$ value all $\omega 2$ values will be explored. This "regular requirement" is the primary rule in PyCMDS scans. The regular requirement is somewhat relaxed, however, by the flexibility of axes themselves.

Besides axes, the scan system has constants.

6.2.1 Axes

In PyCMDS axes are instances of the `Axis` class, defined in modules/scan.py. The scan handler requires a list of axes from the acquisition module. The principle attributes of `Axis` are shown below.

attribute	type	description
`points`	`numpy.ndarray`	The destinations, in units.
`units`	`str`	Axis units.
`name`	`str`	Unique identifier.
`identity`	`str`	Mapping of axis onto hardware.
`hardware_dict`	`dict`	Hardware objects and information.

`points`, `units`, and `name` are self explanatory. `name` must be "python friendly", meaning that it is a valid python identifier. `identity` is a specially formatted string that uses **operators** to describe how specific hardware instances participate in the axis. `hardware_dict` contains, for each participating hardware instance, a list containing the hardware object itself, the name of the method to be called during a scan, and some information about arguments to that method. To fully understand `identity` and `hardware_dict`, a more complete discussion of the flexibility of PyCMDS' axes is required. Consider the following examples. Remember that each example is a single axis, perhaps one of several making up a scan.

Example 1 - a single delay axis

In their simplest usage, axes correspond directly to the motion of a single hardware instance. Consider a simple delay axis with only 5 points.

attribute	type
`points`	`[-1., -0.5, 0., 0.5, 1.]`
`units`	`'ps'`
`name`	`'d2'`
`identity`	`'d2'`
`hardware_dict`	`{'d2': [<Hardware>, 'set_position', None]}`

Here <Hardware> is an instance of the PyCMDS.project.classes.Hardware class. Note that points is a one dimensional array - this is always true. In this simple case name and identity are taken to be identical, but this is not necessarily true - name is an entirely arbitrary unique identifier.

Example 2 - a diagonal delay axis

It is sometimes desirable to direct the motion of multiple hardwares along the same scan axis. In this example two delays are scanned together with an offset. Here hardware_dict is expanded into its individual components for readability.

attribute	type
points	[-100., -75., -50., -25., 0., 25., 50]
units	'fs'
name	'ds'
identity	'd1=d2-15'
hardware_dict['d1']	[<Hardware>, 'set_position', None]
hardware_dict['d2']	[<Hardware>, 'set_position', None]

Note the = and − characters in identity. These are **operator** characters. The scan handler will parse these characters to decide exactly where to send the hardwares. Valid operator characters are =, +, and −. The following table shows the parsed output from this axis - these are the positions that PyCMDS will actually send hardware to during the scan.

axis index	0	1	2	3	4	5	6
axis coordinate	-100.	-75.	-50.	-25.	0.	25.	50.
d1 coordinate	-100.	-75.	-50.	-25.	0.	25.	50.
d2 coordinate	-85.	-60.	-35.	-10.	15.	40.	65.

Example 3 - a tune test

It is often useful to define an axis as about some center position. The central position will change as a function of the scan coordinate in other dimensions. In PyCMDS, such axes are called **differential axes**. The opposite of a differential axis is a direct axis, which we have seen in the previous examples. Differential axes are especially useful in calibration scans, although they can be useful in experimental scans as well. A differential axis cannot be alone, it must be a member of a multidimensional scan. Consider a scan where an OPA is set to a series of output colors and a monochromator is scanned about each subsequent OPA output color. This scan is incredibly useful for calibration purposes - the TUNE TEST scan module is designed to accomplish these scans. For completeness both axes are shown below. First the direct OPA axis.

attribute	type
points	[600., 650., 700., 750., 800.]
units	'nm'
name	'w1'
identity	'w1'
hardware_dict	{'w1': [<Hardware>, 'set_position', None]}

And now the differential monochromator axis.

attribute	type
points	[-150., -100., -50., 0., 50., 100., 150.]
centers	[16666.666, 15384.615, 14285.714, 13333.333, 12500.]
units	'wn'
name	'wm'
identity	'Dwm'
hardware_dict	{'wm': [<Hardware>, 'set_position', None]}

Note that the differential monochromator axis has a new attribute `centers`. It is in the same units as `points`, but its shape `(5,)` is the shape of the rest of the scan coordinates not of this axis. In PyCMDS, differential axis centers are entirely arbitrary. Because of this, a center must be specified for each coordinate in all other scan axes. The character `D` at the very beginning of the `identity` string indicates a differential axis. Importantly, this means that hardware names should not start with `D`.

Example 4 - an OPA motor axis

In previous examples the second and third element in each `hardware_dict` item have been `'set_position'` and `None` respectively. This tells the scan handler to pass the destinations directly into the `set_position` method of the hardware object. PyCMDS is designed to offer finer control to the scan module when required. For example, a scan module may want to directly set an OPA motor position, as shown in the example below.

attribute	type
points	[1., 1.1, 1.2, 1.3, 1.4, 1.5, 1.6]
units	None
name	'w1_Crystal_1'
identity	'w1_Crystal_1'
hardware_dict	{'wm': [<Hardware>, 'set_motor', [motor_name, 'destination']]}

The second element in the `hardware_dict` list is the name of the method to be called, while the third item describes the arguments to be passed to that method. `'destination'` is a special string that will be replaced with the actual destination for each coordinate during the scan.

6.2.2 Constants

Constants tell the scan module to endeavor to keep a specific relationship between hardwares. Constants are almost identical to axes, except that constants are determined

6.3 Saving Data

It is now useful to describe the format of the savefiles that PyCMDS generates over the course of an acquisition. First a description of the files and folders that PyCMDS creates. PyCMDS assembles file and folder names out of the following elements.

element	maximum length	description
"short" timestamp	16	ISO 8061 date and seconds since local midnight
queue name	10	may be empty
acquisition index	3	
module name	10	
name	10	may be empty
data name	15	may be empty
axis names	30	
shape	20	
module reserved	10	may be empty
scan index	3	
file index	3	

Here are the elements of each file/folder name.

queue folder	acquisition folder	scan folder	data file	shots file
"short" timestamp queue name	acquisition index module name name	scan index axis names module reserved	file index .data	'shots' .hdf5
27	25	45	8	10

The maximum filepath to the scan folder(including all 3 separators) is 100 characters. PyCMDS will allow the acquisition modules to write filepaths up to 50 characters within the scan folder during post-processing, meaning the absolute maximum pathlength for any PyCMDS filepath is 150 characters. On Windows the maximum path length is 260 characters. For this reason PyCMDS queues should be stored within paths containing less than 100 characters.

Due to its modular design, PyCMDS data files may look very different from scan to scan and system to system. In light of this diversity, PyCMDS uses headers to record a large amount of information about the acquisition, including the identity of each column and the scan axes. PyCMDS always saves flattened arrays.

PyCMDS has two data types: `.data` and `.shots.hdf5`.

Headers are dictionaries of information that get written to the top of data files written by PyCMDS. PyCMDS endeavors to record enough information in the headers to allow for very easy processing of the data. Headers may contain bools, numbers, strings, lists, and multidimensional arrays. Header formatting in the Wright group is a little bit unusual, but tools exist to simplify working with them. Conjugate functions for reading and writing headers are contained in WrightTools: `dictionary = wt.kit.read_headers(filepath)` and `wt.kit.write_headers(filepath, dictionary)`.

For each axis...

Centers arrays necessarily have the same units as points arrays. Generally, they are multidimensional arrays - they have a defined value for every point except for the scan dimension. Axes are never defined as differential for 1D data. Following are some specific examples.

data shape	axis index	centers shape
(51, 256)	1	(51,)
(51, 101, 41)	0	(101, 41)
(51, 101, 41)	1	(51, 41)

Within the program, headers are assembled in several steps in both PyCMDS\modules\scan.py and PyCMDS\daq\daq.py. The headers are stored in an instance of the daq.Headers class, which contains as attributes a series of seperate OrderedDicts:

- pycmds_info
- scan_info
- data_info
- axis_info
- constant_info
- channel_info
- daq_info
- data_cols
- shots_cols

These dictionaries are joined together in daq.FileAddress when a new data or shots file is created. Since these dictionaries are populated in multiple places throughout PyCMDS, a chronological listing is useful:

1. scan.Address.run populates data_info
   (a) data name
   (b) data info
   (c) data origin
2. scan.Address.run populates axis_info. Here, *scan* axis info is entered. These entries may be edited later if *acquisition* axes are present in the scan.
   (a) axis names
   (b) axis identities
   (c) axis units
   (d) axis points
   (e) axis centers
3. daq.Control.initialize_scan populates pycmds_info
   (a) PyCMDS version

(b) system name

(c) file created (this is where the master file timestamp is defined)

4. `daq.Control.initialize_scan` populates `axis_info`. Acquisition axes are added, if present. This may involve expanding the dimensionality of axis centers for axes defined in `scan.Address.run`

   (a) axis names

   (b) axis identities

   (c) axis units

   (d) axis points

   (e) axis centers

   (f) axis interpolate

5. `daq.Control.update_cols` populates `data_cols`, `shots_cols`. These values are lists containing an object for each column in the file.

   (a) kind (`None`, `'hardware'`, or `'channel'`)

   (b) tolerance

   (c) units

   (d) label

   (e) name

6. `daq.Control.initialize_scan` populates `channel_info`

   (a) channel signed

7. `daq.Control.initialize_scan` populates `daq_info`. This is entirely dependent on what the daq hardwares return in `hardware.get_headers()`

6.4 Acquisition

Once the destinations that will be covered in a scan are calculated, the order of acquisitions must be decided. Order can be important for multidimensional scans including hardwares that move at different speeds, and for experiments in which signals drift due to slowly varying lab conditions. Setting the rules for acquisition ordering is a technical and parameter rich process - it is not practical to create a general GUI for the user to decide how scans are ordered. Instead, users may write relatively light-weight Python modules that interface with the back end of PyCMDS. These modules are meant to be parameter free. In the module GUI, the user may choose which order-handling module they wish to use to decide how acquisition ordering is handled.

Order-handling modules must contain a Python function `process` with exactly one argument: `destinations_list`. When called by `PyCMDS/modules/scan.py`, the `destinations_list` will be passed onto the method. The method must return a tuple containing two lists, `out` and `slices`. `out` is a list of tuples containing, in order, the scan indices for each acquisition. `slices` is a list of dictionaries, in order. Each dictionary will contain keys `index`, `name`, `units`, and `points`. `index` refers to an integer value in the flattened acquisition space.

The number of shots for each device must be constant over the entire scan.

6.5 SCAN

6.6 MOTORTUNE