manual

FAB10 LIZARD

(based on PyMoDAQ 1.6.3)

https://github.com/Attolab/FAB10_LIZARD

29. avr. 2021

I. Introduction

This PID model is associated to the following physical system. In a Mach-Zender interferometer, one arm is dedicated to the production of XUV attosecond pulse trains, the other one is dedicated to the IR dressing.

As for the MichelsonDemo model, we get two sinusoidal signals with the interferometer length, that are in phase quadrature to produce an error signal. The main difference come from the physical nature of those signals, that come from modulation of two sidebands in a RABBIT setup, but the program is basically the same. Details can be found in the references given at the root of the repository.

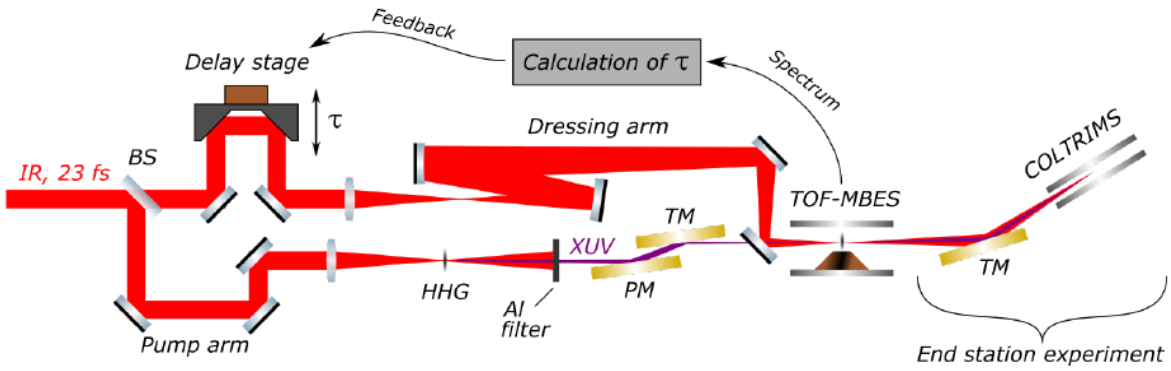


Illustration 1: Physical system associated to the PIDModelLIZARD class

- The actuator is a delay stage that is controlled through a National Instruments card (daq_move_NI... module).
- The detector is an oscilloscope (daq_1Dviewer_LecroyWaverunner6Zi module).

The modulated signals come from two ROIs defined in the scope spectrum (ROI_00 and ROI_01). A third ROI is used to define the baseline of the spectrum/the offset (ROI_02).

II. Hardware and setup

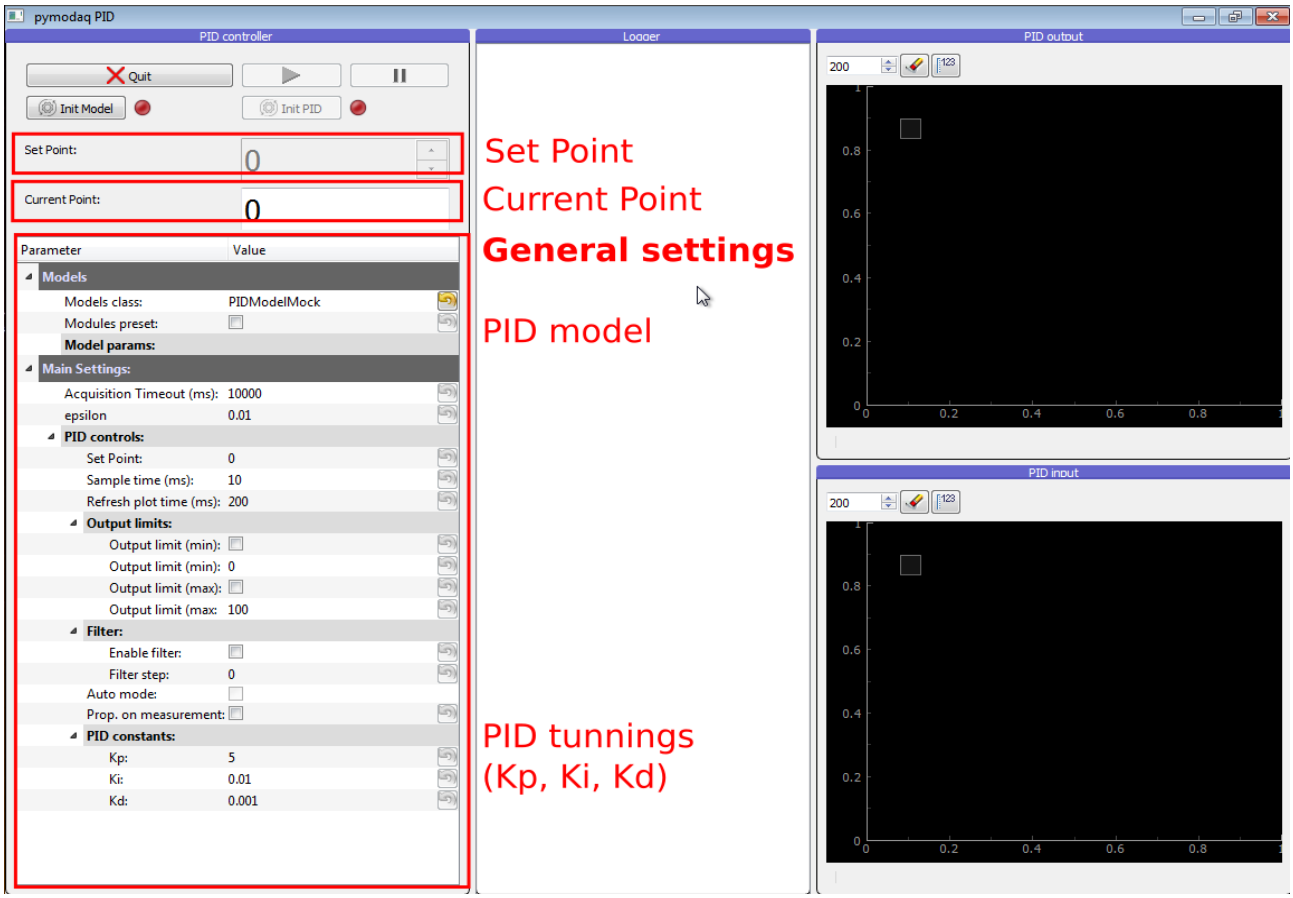
- Check that the actuator (Smaract) and the oscilloscope (Lecroy Waverunner 6Zi) communication are correctly plugged.
- We suppose to have optimized RABBIT conditions.
- The oscilloscope communication should be configured as USBTMC.

- Configure the oscilloscope: in Sequence Mode with typically 500 sweeps averaging (2Hz) : if the acquisition frequency is too high, it may cause some desynchronization of the acquisition (see in the scope viewer) which is very bad for a feedback loop. Choose properly your window range, offset : once the oscilloscope module is initialized, do not touch the oscilloscope parameters ! You will probably have to restart the program if you do.

III. Use the program

- We suppose you have followed the installation instructions from the repository: https://github.com/Attolab/FAB10_LIZARD
- In a python IDE (like PyCharm) run pymodaq/daq_utils/pid/pid_controller.py
- You can also use the following command in a terminal where your environment is activated: python -m pymodaq.daq_utils.pid.pid_controller

User interface



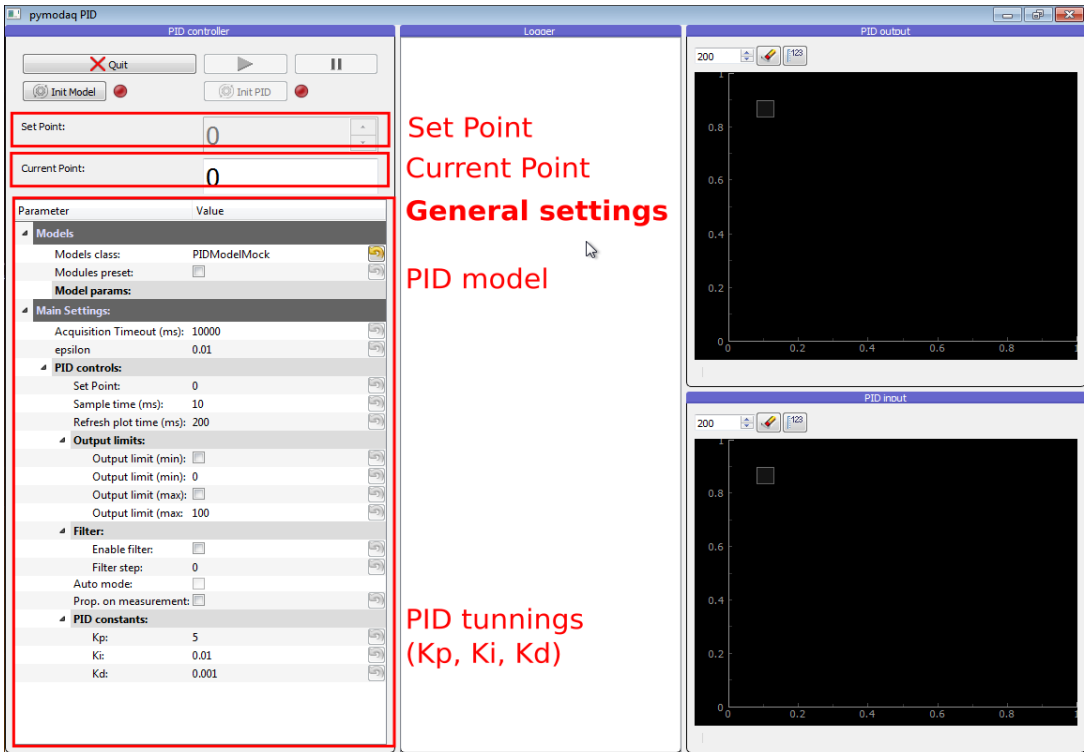


*Illustration 2: User interface at startup. The **Set Point** is the targeted position, and **Current Point** is the current measurement of the position. On the left pannel the user can configure some general settings. In particular a **model class** should be chosen, which defines the physical system to control.*

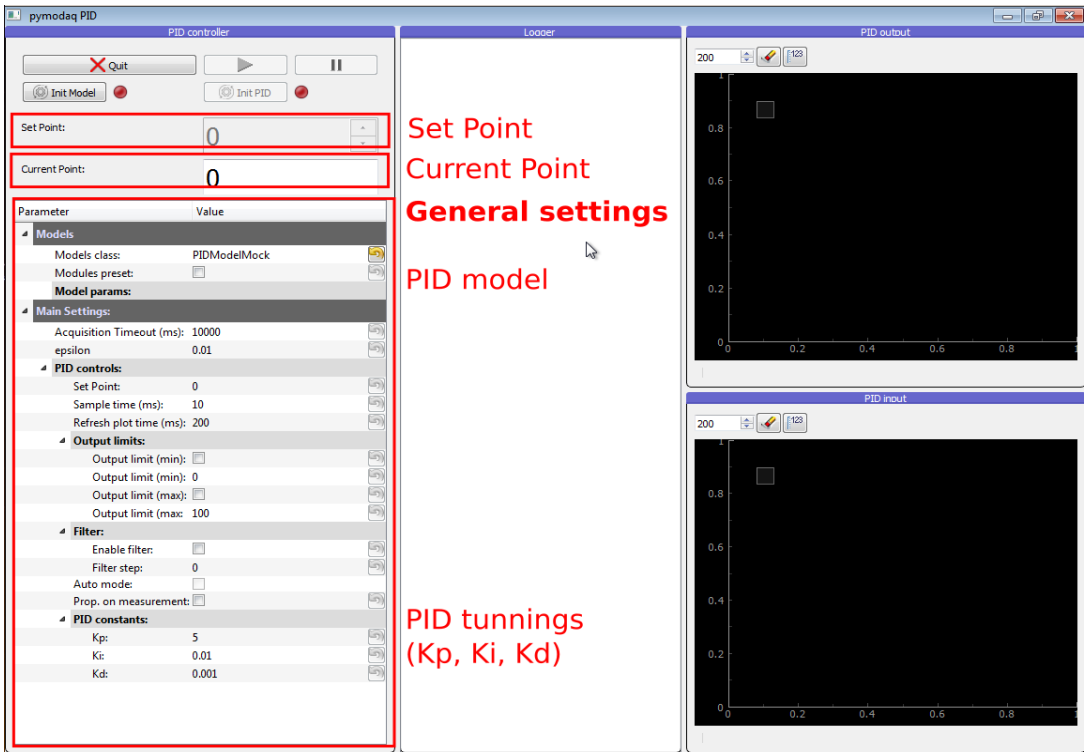


*Illustration 3: User interface at startup. The **Set Point** is the targeted position, and **Current Point** is the current measurement of the position. On the left pannel the user can configure some general settings. In particular a **model class** should be chosen, which defines the physical system to control.*

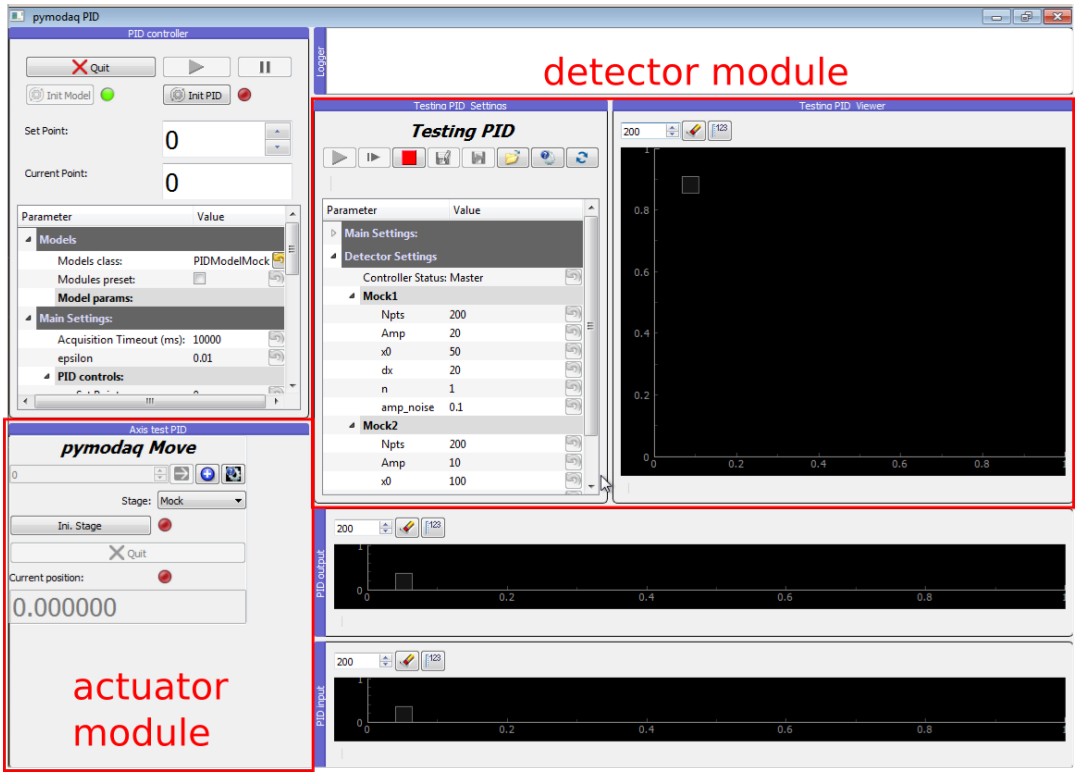


Illustration 4: User interface after the initialization of the PID model. The detector/actuator modules are now loaded.

Initialization

- In **Parameter/Models/Models class** choose **PIDModelLIZARD_FAB10** and click **Init model**. You will be asked to select (or create) a .h5 file where the data of the session will be saved.
- Initialize the piezo and put 0,002 < epsilon < 0,01 micron. Which defines the precision you require on the target position. Do not put it lower than the precision of your actuator!
- Initialize the oscilloscope module. Check that you choose the correct channel. Start grabing.

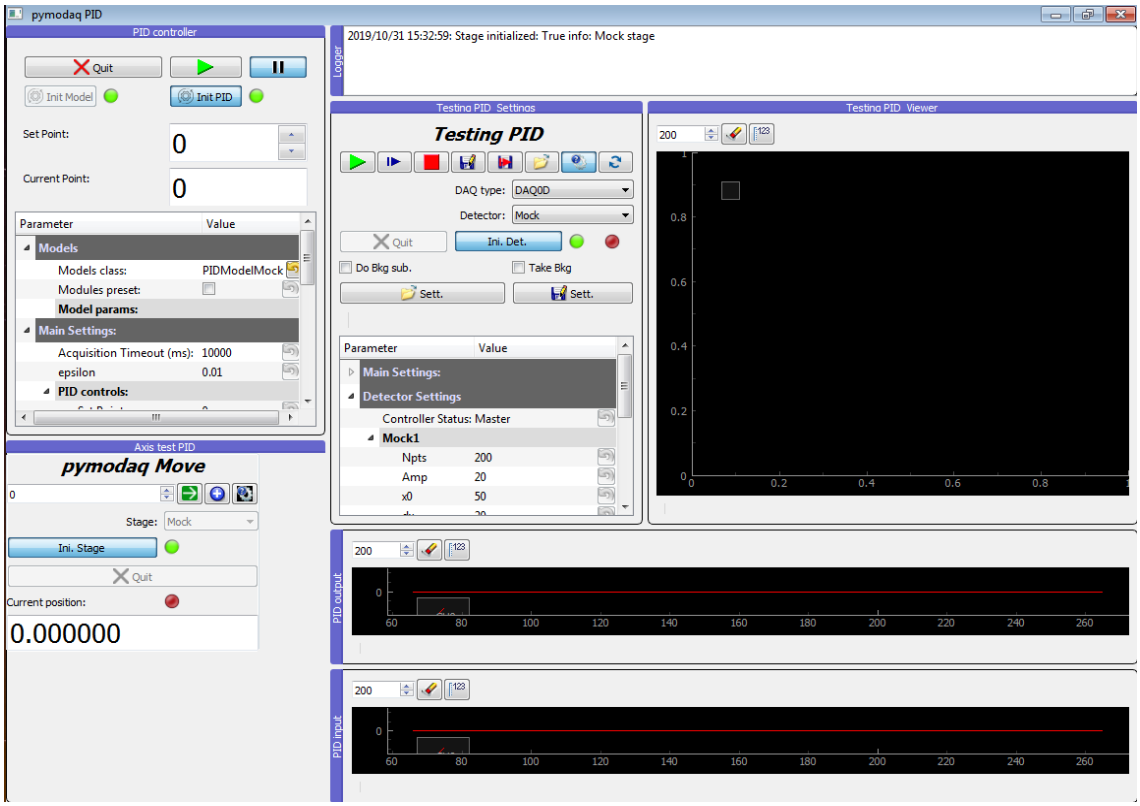


Illustration 5: User interface after the initialization of the actuator/detector modules and the PID module.

Define 3 ROI

Define 3 ROI (region of interest) as following:

- click on the calculator icon on top of the scope viewer
- click "Add" to add a new ROI. Define left/right positions, color, and Math type = Mean
- ROI_00 should correspond to the first modulated signal M1. Typically sideband 16 (yellow color recommended).
- ROI_01 corresponds to M2. Typically sideband 22 (green color recommended)
- ROI_02 corresponds to the offset of the signal. Should be taken in a region where no electrons are detected. It is very important to respect those assignments otherwise the error signal cannot be properly calculated. (blue color recommended).
- Once your ROIs are correctly defined click on "Save ROIs" and save the file somewhere so it will prevent you from doing the ROI definition next time by clicking on "Load ROIs".

Calibration

- Go to Parameter/Models/Model params/Calibration/Calibration. Define start/stop positions (typically 2 microns range to get 10 oscillations) and step size (typically 20nm). The start position corresponding to your time zero.

- Tick Do calibration. At the end of the scan should appear in a different window on the left the ROIs values within the scanning range, on the right the corresponding XY representation and the ellipsis fit (green). At the end of the scan the delay stage should return to the starting position (time zero), the parameters of the fitting ellipsis should be set (Dx, Dy, x0, y0, phi) and the phase of the time zero.
- At the end of the scan the data and metadata will be saved in a new Scan node of the h5 file.

- Unfortunatelly if the scan does not terminate the saving will not work !

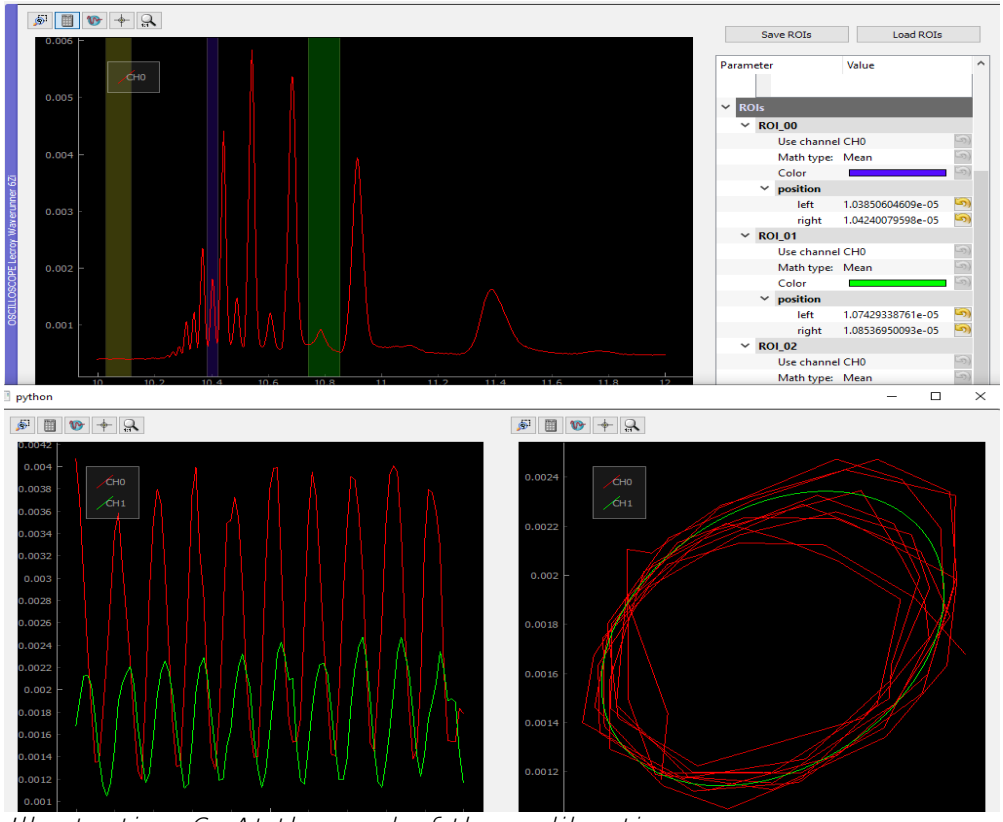


Illustration 6: At the end of the calibration scan.

Stabilization loop

- set the parameters of the PID: Kp, Ki...
- click **Init PID**: this will not produce any effect to the user except that the PLAY and PAUSE button are now available.
- click **PLAY**: clicking this button, the feedback loop is launched, in the sense that all the phases are calculated etc. But the orders are not sent to the actuator for now because the paused attribute is still True.
- click **PAUSE** (uncheck)

Stabilized scan

The stabilized scan should be launched while the stabilizing loop is running. It will just change the setpoint at regular intervals, as defined by the user.

- Go to **Parameter/Models/Model params/Stabilized scan/Stabilized scan parameters**. Define the **length** and the **step size** in microns. The stabilized scan will necessarily start at the current position. Define **Iterations per setpoints** which corresponds to the number of feedback corrections you want per setpoint. The time between each correction will depend on your configuration, in particular the accumulation time of the oscilloscope.
- Tick **Do stabilized scan.**
- At the end of the stabilized scan the data and metadata will be saved in a new Scan node of the h5 file.
- Unfortunatelly if the scan does not terminate the saving will not work !

IV. Data saving

The data are stored in the h5 file that is defined at the initialization of the model.

To read this file you can use the **HDFView** program (https://www.hdfgroup.org/downloads/hdfview/) or the PyMoDAQ **H5Browser** module (http://pymodaq.cnrs.fr/en/latest/usage/modules/Utility_Modules.html#h5browser).

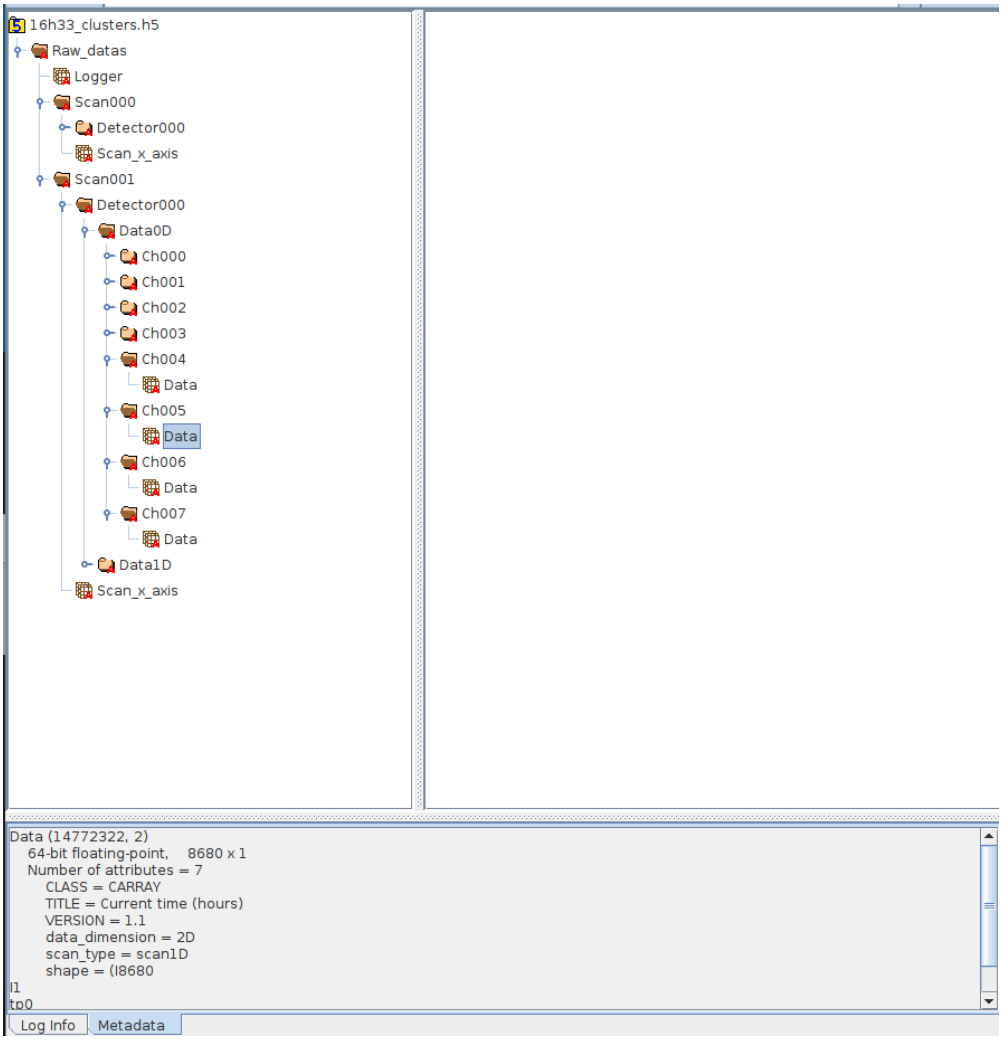


Illustration 7: Reading the h5 file (with HDFView). The metadata are read in the bottom panel.

For each scan (calibration or stabilized) a node ScanXXX is created.

Detector000 node corresponds to the oscilloscope.

Data0D node corresponds to data of dimension 0 : time of the acquisition, phase, error...

Data1D corresponds to spectra of the oscilloscope.

The meaning of the different arrays stored should be found in the metadata associated.

V. DAQ_PID module (developper documentation)

The PID module is useful if you would like to control a parameter of a physical system (control a temperature, the length of an interferometer, the beam pointing of a laser...).

In order to achieve this you need a set of detectors to read the current state of the system, an interpretation of this reading and a set of actuators to perform the correction.

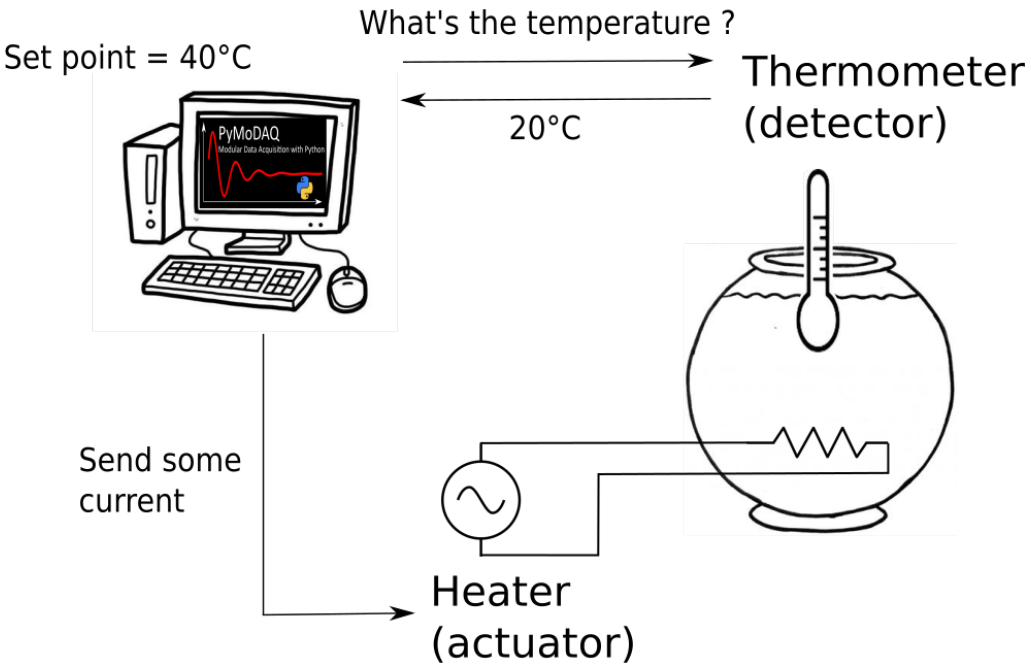


Illustration 8: For example, the PID module can be used to control the temperature of water in a container.

The PID module is supposed to be adaptable to many physical systems. If you want to adapt it to your specific need, you would have to code the plugins for your detector(s)/actuator(s) and what is called a PID model to translate the readings of the detectors as a state of your system.

V.I. Workflow

To test the module you do not need any actuator nor detector. Some 'Mock' modules will simulate some actuators and detectors.

For the execution of the PID module you just need to run

pymodaq/pymodaq/daq_utils/pid/pid_controller.py

You should get the following window

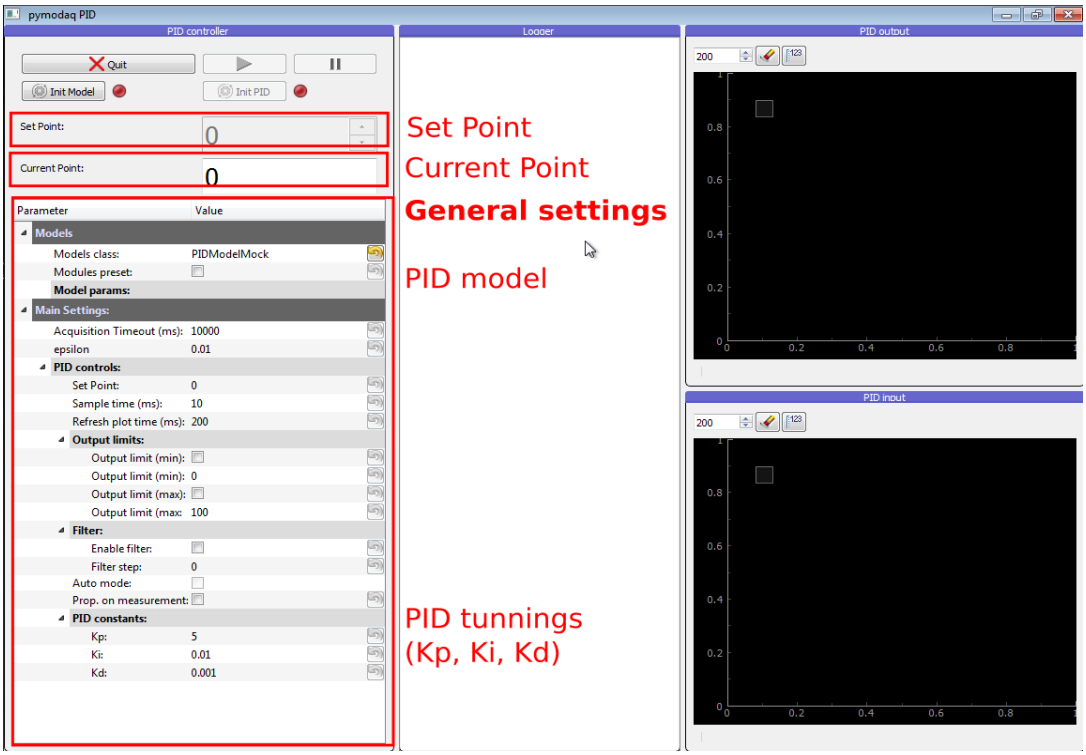


*Illustration 9: User interface at startup. The **Set Point** is the targeted position, and **Current Point** is the current measurement of the position. On the left pannel the user can configure some general settings. In particular a **model class** should be chosen, which defines the physical system to control.*

By default the Model class is PIDModelMock. The model contains the information about the detectors and actuators that are involved in our project. The PIDModelMock is made to be able to perform some tests on the module without the need of any detector or actuator. So keep it like that.

Then click on **Init Model**.

As you can see it has loaded mainly two important windows:

- Axis test PID which represents an actuator. Let us say a heating system.
- Testing PID settings which represents a detector. Like a thermometer.

Behind them are some actuator and detector modules (Mock modules in our case) which have to be initialized. Let's do so by clicking on Ini. Stage and Ini. Detector (for this one you have to click first on Open settings to see the button).

Then click Init PID. You should get something like that

Then run the PID with the Play button and play with the Set Point, which represents the targeted position (like the temperature in °C)

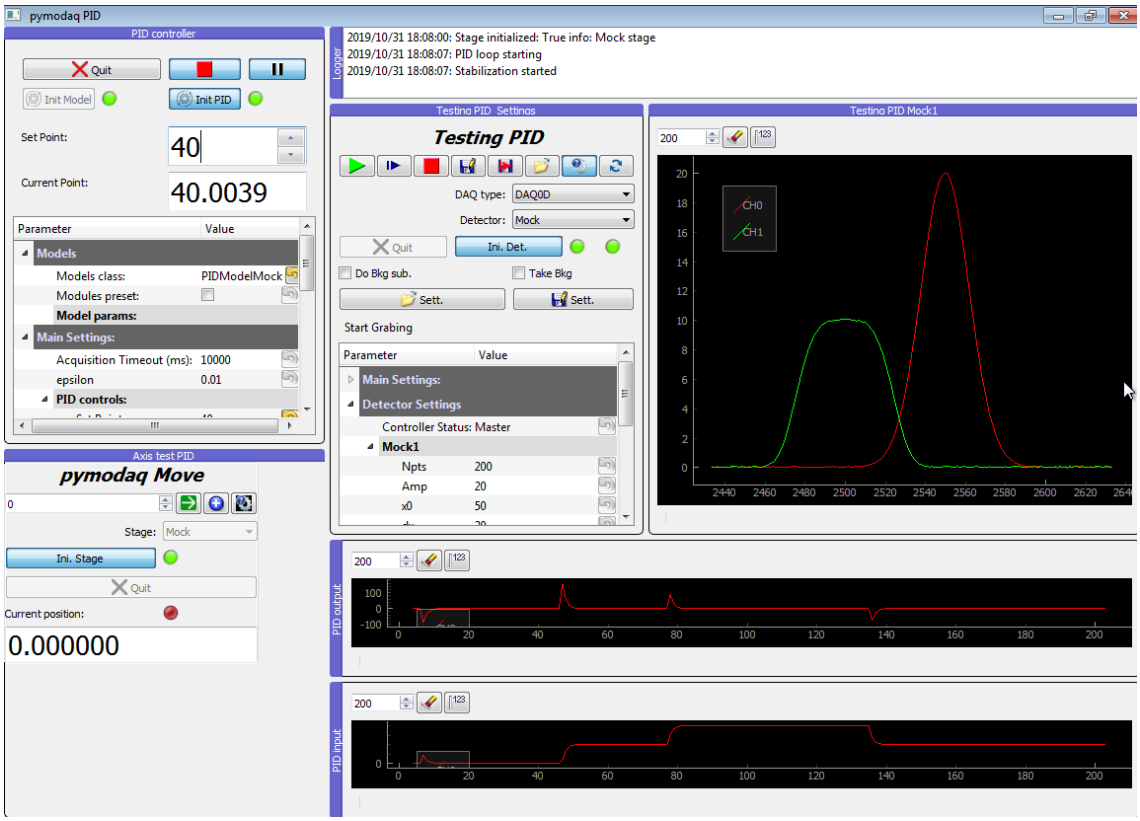


Illustration 10: PID reaction after changing the Set Point. It has been changed from 0 to 40, then to 80, and back to 40.

This is displayed in the PID input window, which displays the acquisition of the detector with time.

The PID output window gives the position of the actuator with time. We can notice that it was particularly active at each change of the targeted position (Set point).

V.II. Files locations

The DAQ_PID and PIDRunner classes are defined in

pymodaq/pymodaq/daq_utils/pid/pid_controller.py

and the PIDModelGeneric class is defined in

pymodaq/pymodaq/daq_utils/daq_utils.py

The PID models are in an independant repository

pymodaq_pid_models

V.III. Structure of the module

Overview

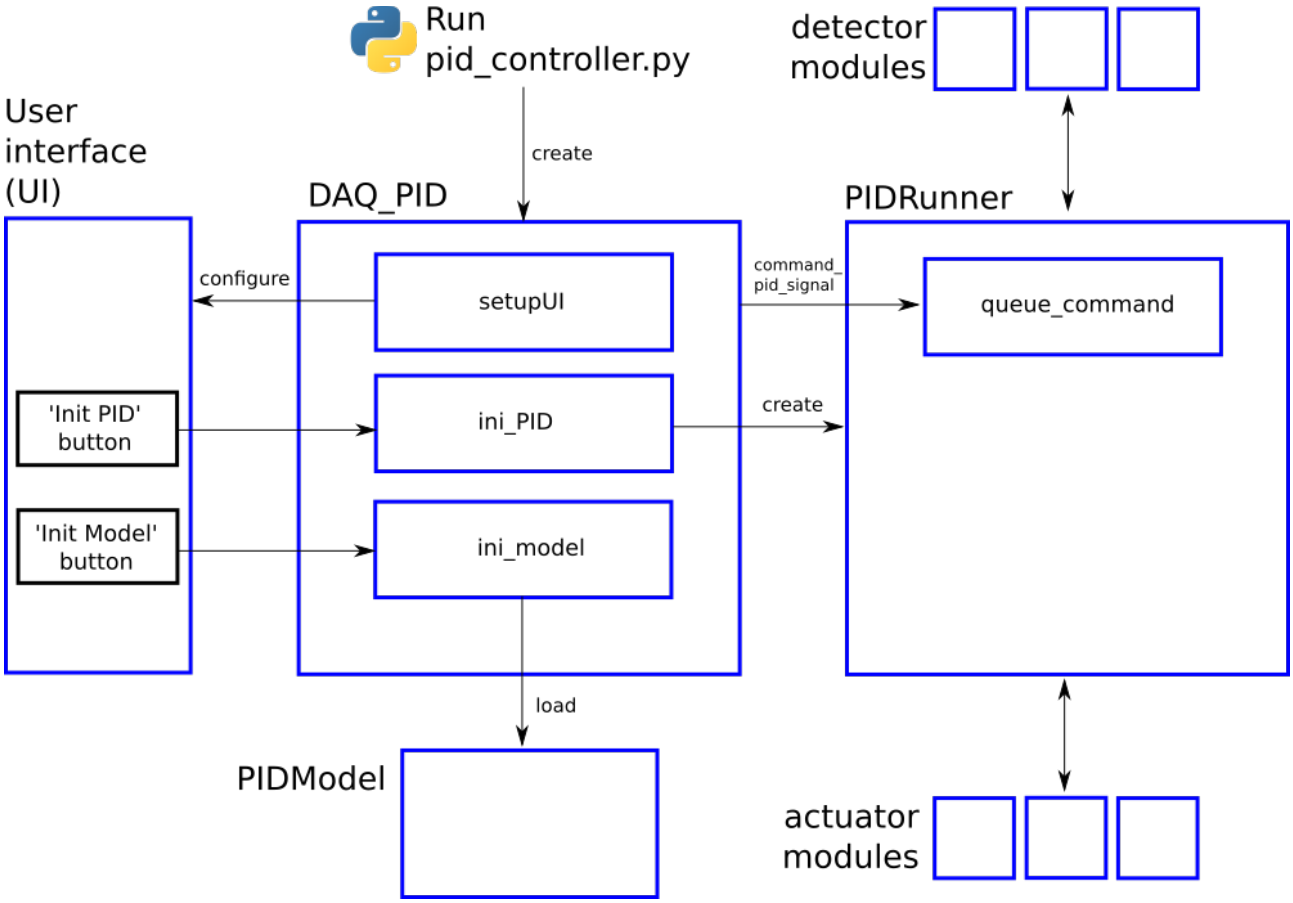


Illustration 11: General structure of the PID module

The entrance point of the module is reached by running the **pid_controller.py** file that will create a DAQ_PID instance, which is the main class of the program.

Firstly, the **DAQ_PID** class manages the initialization of the program: settings of the user interface, loading of the corrersponding PIDModel, as set by the user, instanciation of the PIDRunner class... It also makes the interface between the user, who acts through the **user interface**, and the PIDRunner class. The communication between the DAQ_PID and the PIDRunner is mainly done by the **command_pid_signal** and the **queue_command** method.

The **PIDRunner** class is created and configured by the DAQ_PID. It is in direct relation with the detectors and actuators. It synchronizes the instruments to perform the PID loop (see next section).

A **PIDModel** class is defined for each specific physical system the user wants to control. Here are defined the actuator/detector modules involved, and the methods to convert the information received from the detectors as orders to the actuators to perform the desired control (see next section).

PID loop

The conductor of the PID loop is the **PIDRunner** class. In particular the start_PID method. It synchronizes the actions of the detector/actuator modules, and the PID module. The workflow for each iteration can be mapped as in the following figure.

The starting of the PID loop is triggered by the user through the Play button.

The PIDRunner will ask the detector(s) to start an acquisition. When all are done, the wait_for_det_done method will send the data (**det_done_datas**) to the PIDModel class.

A **PIDModel** class should be defined for each specific physical system the user wants to control. Here are defined how much detectors/actuators are involved, and how should the information sent by the detector(s) should be converted as orders to the actuators (**output_to_actuators**) to reach the targeted position (the **Set point**). The PIDModel class is thus an iterface between the PID class and the detectors/actuators.

The **PID** class is defined in an external module (simple_pid: https://github.com/m-lundberg/simple-pid). It performs a pid controller. The tunnings (Kp, Ki, Kd) and the Set point are configured by the user through the user interface. From the **input**, which corresponds to the current position of the system measured by the detectors, it will return an **output** that corresponds to the order to send to the actuators to stabilize the system around the **Set point** (given that the configuration has been done correctly).

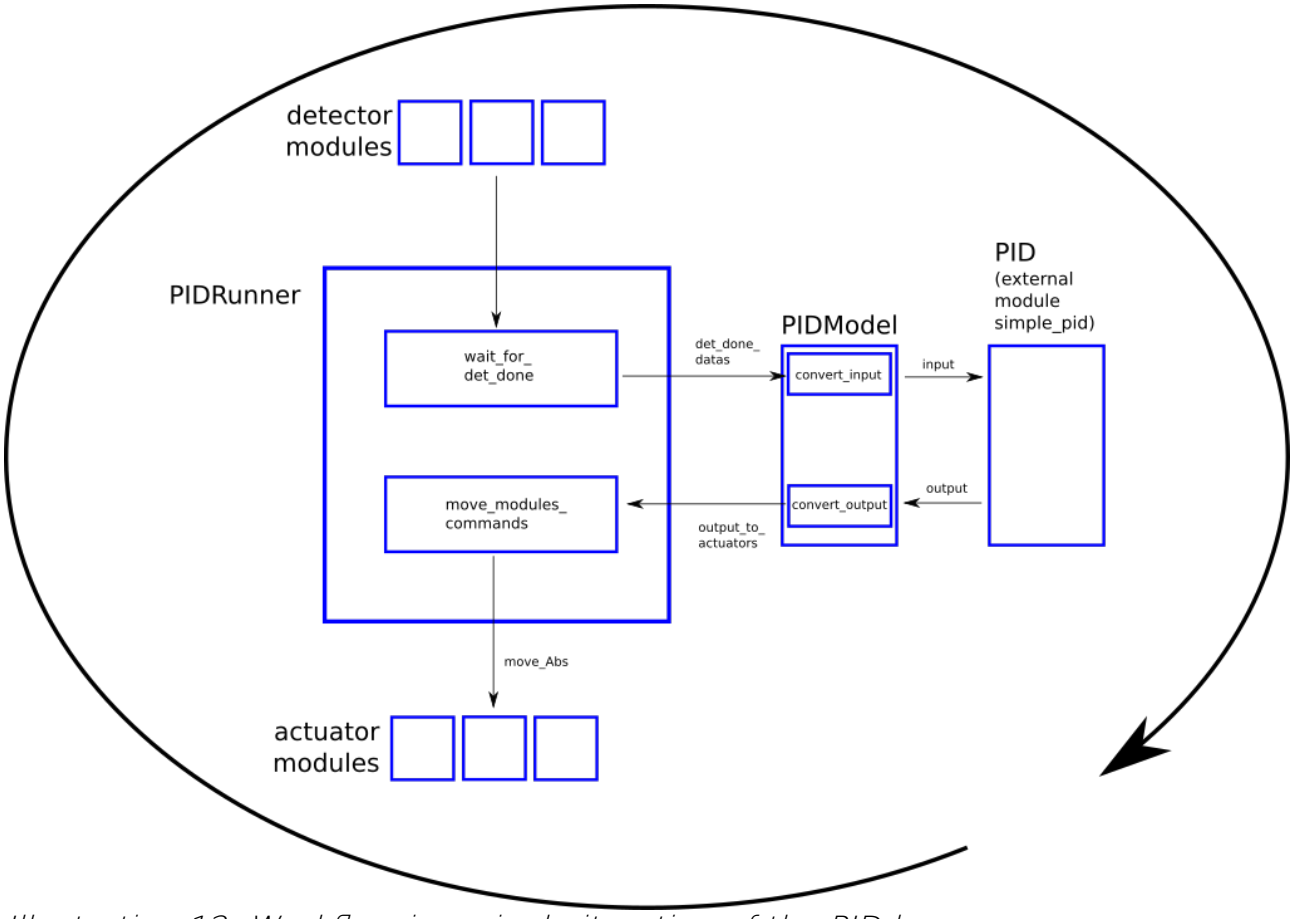


Illustration 12: Workflow in a single iteration of the PID loop

V.IV. Classes

DAQ_PID class

pymodaq/pymodaq/daq_utils/pid/pid_controller.py

This class represents the main class that links all the needed submodules and the user interface (UI). It is at the top of the hierarchy of the PID module.

In the following we could call it '**main pid**' or '**pid controller**'.

When you run pid_controller.py as __main__ you create a DAQ_PID class.

There is no user interface file for this class. The user interface is constructed with the setupUI method by putting the user interfaces of the submodules in a pyqtgraph DockArea container widget.

signals		
	log_signal	Connected to **add_log** method.
	command_pid	Connected to **PIDRunner.queue_command**.
	command_stage	Connected to the method **move_Abs** in the constructor. This signal is emitted when we want to move the actuator.
	move_done_signal	
attributes		
	setpoint_sb	SpinBox object that displays the target point 'Set point' of the PID in the user interface. Changing the value will automatically change the corresponding value of the settings.
	dock_area Instance of **daq_utils.gui_utils.DockArea** (similar to pyqtgraph DockArea)	Set in the constructor. Widget container.
	currpoint_sb	SpinBox object that displays the current point 'Set point' of the PID in the user interface.
	input_viewer	Viewer0D object that displays the PID input signal in the user interface.
	output_viewer	Viewer0D object that displays the PID output signal in the user interface.
	PID_Thread	Thread initialized in ini_PID to welcome the **PIDRunner** object.
	ini_PID_action	Represents the push button "Init PID" of the UI. Connected to the **ini_PID** method.
	run_action	Represents the **PLAY/STOP** button of the UI. Connected to the **run_PID** method.
	pause_action QPushButton	Represents the **PAUSE** button of the UI. Initialized as 'checked' in **setupUI**. Connected to **PIDRunner.pause_PID** method.
	model_class Instanciation of the class	Set in **ini_model** method.

	associated to the model. For example, instanciation of PIDModelMichelsonDemo.	
	quit_action	Represents the Quit button of the UI. Connected to the **quit_fun** method.
	move_modules_commands	???
	detector_modules array of **DAQ_Viewer**, one for each detector	Set in the constructor if given as an argument, or in **ini_model** if a corresponding preset file is found.
	actuator_modules array of **DAQ_Move**, one for each actuator	Set in the constructor if given as an argument, or in **ini_model** if a corresponding preset file is found.
	settings subclass of type 'group' of the pyqtgraph Parameter object	Initialized in the constructor.
	settings_tree pyqtgraph **ParameterTree** object	
methods		
	__init__ arguments **area** (instance of daq_utils.gui_utils.DockArea, similar to pyqtgraph DockArea) **detector_modules**=[] (array of DAQ_Viewer) one for each detector **actuator_modules**=[] (array of DAQ_Move) one for each actuator	Initialize some attributes. Connect the **command_stage** signal. Call the methods setupUI, enable_controls_pid, enable_controls_pid_run
	ini_PID	Triggered by '**Init PID**' button of the UI. Instanciate a **PIDRunner** object. Move it to the **PIDThread**. Connect signals between DAQ_PID and PIDRunner. Start PIDThread. Call **enable_controls_pid_run**.
	process_output	Called by **pid_output_signal** from

		PIDRunner class. Update the display of PID input/output viewers and 'Current point' in the UI.
	move_Abs	Triggered by **command_stage** signal. Change the value of setpoint_sb according to the input command ???
	enable_controls_pid	Unlock the "Init PID" button and the "Set point" spin box. Called after having initialized the PID model with the ini_model method.
	enable_controls_pid_run	Unlock the Play/Stop and the Pause buttons. Called after having initialized the PID with the ini_PID method.
	setupUI	Initialization of the **user interface**. Connect the buttons with the methods.
	get_set_model_params	Set the PID model parameters ???
	run_PID	Called by the **Play/Stop** button of the UI. Send the "start_PID" and "run_PID" commands to the queue_command method of the PIDRunner object.
	pause_PID	Called by the **Pause** button of the UI. Send the "pause_PID" command to the queue_command method of the PIDRunner object.
	update_status	Show the txt message in the **status bar** with a delay of wait_time ms. This method is called each time the program wants to communicate to the user. For example it will be called each time an **Exception** is raised.
	add_log	Add the QListWidgetItem initialized with txt informations to the User Interface logger_list and to the save_parameters.logger array. Called by **self.log_signal** and by the log_signal from all the detectors and actuators connected.
	set_file_preset	Set a file preset from the converted xml file given by the filename parameter. ??? Returns actuator_modules, detector_modules. Called by **ini_model** method.
	stop_moves	Foreach module of the move module object list, stop motion. Called by **bound_signal** from actuator

		modules or from **overshoot_signals** from detector modules. ???
	set_default_preset	??? Similarities with set_file_preset... Returns actuator_modules, detector_modules. Called by **ini_model** method.
	ini_model	Triggered by the "**Init model**" button of the UI. Set **model_class** by calling the constructor with self as argument ?? Load the PID model that has been set in the settings and try to get the corresponding preset file. ??? A PID model corresponds mainly to a set of detector and actuator modules. Build **self.actuator_modules** and **self.detector_modules**. Connect the **log_signals** to the **add_signal** method.
	quit_fun	Called by the "**Quit**" button of the UI. Exit PIDThread. Quit the detector/actuator modules.
	parameter_tree_changed	Called by **self.settings.sigTreeStateChanged.** Update the settings.
	thread_status	Called by **status_sig** signal from PIDRunner class. General function to get datas/infos from all threads back to the main.

PIDRunner class

pymodaq/pymodaq/daq_utils/pid/pid_controller.py

signals		
	status_sig	Connected to **thread_status** method of the DAQ_PID class.
	pid_output_signal	Connected to the **process_output** method of the DAQ_PID class.

		Emitted by timerEvent method.
	grab_done_signals	Array that contains all the **grab_done_signal** from the detectors. A detector emits this signal when it has finished is acquisition.
	move_done_signals	Array that contains all the **move_done_signal** from the actuators. An actuator emits this signal when it has reached his position.
attributes		
	input	Stores the input to the pid controller constructed from the detectors acquisition at the begining of each iteration of the pid loop (see **start_PID** method).
	output	Constructed in **start_PID**. Basically the output of the pid module. It will be used as the input of **model_class.convert_output**.
	output_to_actuator	Set as the output of **model_class.convert_output**. Array that contains the absolute values to send to the actuators in the end of each iteration of the pid loop (see **start_PID** method).
	timeout_scan_flag	Set to True if a timeout occured.
	running	Set to True by the constructor (__init__). Set to True by **run_PID**. Set to False by **stop_PID**.
	pid A PID object from the external module **simple_pid** (https://github.com/m-lundberg/simple-pid). This module is automatically installed with pymodaq (see setup.py 'install_requires' entry).	Instance of PID class (defined in the external module **simple_pid**).
	model_class A PIDModelxxx object	The corresponding **PIDModel** class. Pass as an argument in the constructor.
	det_done_datas OrderedDict(	Stores temporarily (for one iteration of the pid loop) the acquisitions of the detectors.

	'Det000' = data_Det000, 'Det001' = data_Det001, ...) with data_Detxxx having the same structure as the argument of DAQ_Viewer.grab_done_signal.	They are then send to **PIDModel.convert_input** to be converted as an understandable input for the pid controller.
	det_done_flag	Initially set to False. Set to True by the **det_done** method each time the acquisition from all the detectors has been received.
	move_done_positions	Stores temporarily (for one iteration of the pid loop) the positions of the actuators.
	move_done_flag	Initially set to False. Set to True by the **move_done** method each time the position of all the actuators has been reached.
	move_modules_commands	
	paused boolean	Set to True in the constructor. Its value can only be set to False if the user uncheck the PAUSE button (through the **pause_PID** method). No order will be sent to the actuator(s) if it is True (see **start_PID**). The user has to uncheck the **PAUSE** button to actually start the correction.
methods		
	__init__	Initialize some attributes.
	timerEvent	Emit **pid_output_signal.**
	timeout	Emit **status_sig** signal. Put **timeout_scan_flag** to True.
	wait_for_det_done	Process events until the detectors are ready or timeout.
	wait_for_move_done	Process events until the actuators are ready or timeout.
	queue_command	Called by **DAQ_PID.command_pid**. This method is an interface between DAQ_PID and PIDRunner classes. It receives commands from DAQ_PID to execute a PIDRunner method.
	update_input	Update **self.input** using **model_class.convert_input** method.

	arguments measurements (Ordereddict) Ordereded dict of object from which the model extract a value of the same units as the setpoint	
	start_PID arguments input (no used ??)	Triggered by the Play button of the user interface. This loop should run after the initialization and the calibration. Loop (while self.running): - get detectors acquisitions - convert acquisitions to pid input - calculate pid output - convert pid output to actuators
	det_done	Triggered by all the detectors signals contained in grab_done_signals. Construct det_done_datas. Set det_done_flag to True when all the detectors are ready.
	move_done	Triggered by all the actuators signals contained in move_done_signals. Construct move_done_positions. Set move_done_flag to True when all the actuators have reached their positions.
	set_option	???
	run_PID arguments last_value	Set running to True. ??? play with pid.set_auto_mode
	pause_PID arguments pause_state bool	Triggered by the 'Pause' button of the GUI. pause_state argument is True if the button is checked. ??? play with pid.set_auto_mode
	stop_PID	Set running to False.

PIDModelGeneric class

pymodaq/daq_utils/daq_utils.py

The PIDModelGeneric class is an abstract class that is intended to give a general template for all PIDModel classes. Every PIDModel class inherits from it.

IS THIS CLASS USEFULL ???

signals		
	status_sig arguments Instance of daq_utils.ThreadCommand	
attributes		
	pid_controller Instance of DAQ_PID.	
	data_names	Updated by update_detector_names. Declared in the parent class. It seems like this attribute is made to automatically get the strings that has to be used to retrieve the ROI values. For now we do not exactly get how it is supposed to work and skip it. In ModelLIZARD we write those strings by hand in the code. See 'How to get the value of a measurement (ROI) ?' in the Cookbook section.
	settings	Set of parameters. Set in the constructor from the settings of pid_controller.
	curr_output	
	curr_input	
methods		
	__init__ arguments pid_controller instance of DAQ_PID.	Set pid_controller.

	update_detector_names	Update self.data_names
	update_settings	Get a parameter instance whose value has been modified by a user on the UI To be overwritten in child class
	ini_model	To be overwritten in child class.
	convert_input arguments measurements (Ordereddict) Ordereded dict of object from which the model extract a value of the same units as the setpoint	Convert the measurements in the units to be fed to the PID (same dimensionality as the setpoint)
	convert_output arguments output (float) output value from the PID from which the model extract a value of the same units as the actuator dt (float) ellapsed time in seconds since last call	Convert the output of the PID in units to be fed into the actuator

PIDModelMock class

pymodaq_pid_models/pymodaq_pid_models/models/PIDModelMock.py

A PID model represents the actual system on which we will apply a feedback loop.

It defines the actuators and the detectors that will be used. This is done at the instanciation of the class within the actuators and detectors variables.

This PID model is very specific since it is a 'mock' one. Which means that it is intended to be used in order to test the model without any real actuator/detector. So we should not be surprised that the actuators and detectors variables are not used.

This PID model intends to simulate the following physical system.

In this

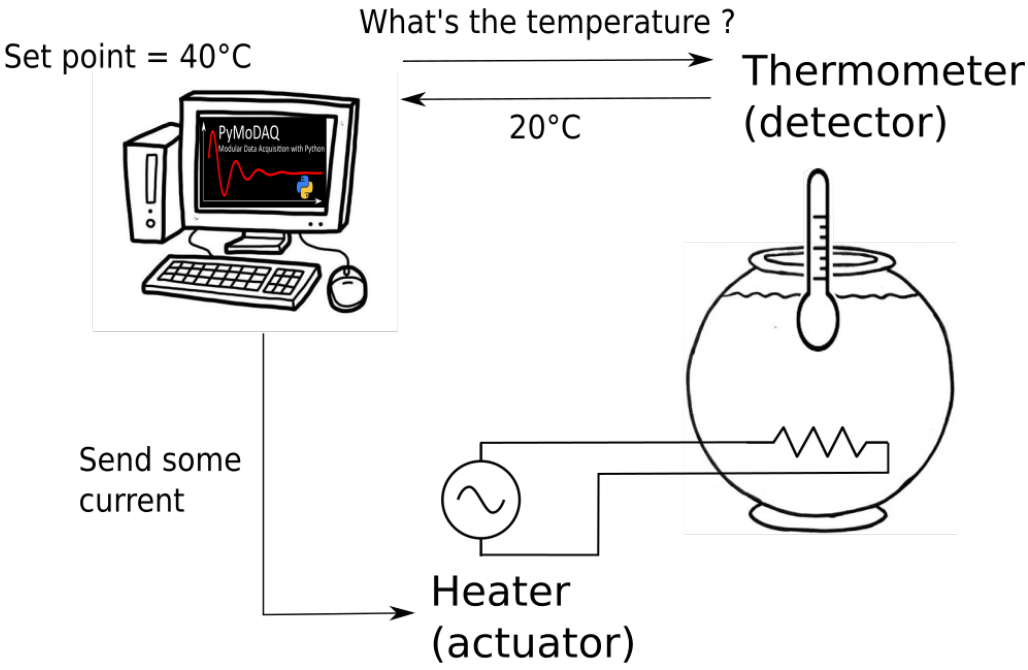


Illustration 13: Physical system simulated by the PIDModelMock

particular case there is

- One actuator: the **heater** that is made of a current generator and a resistor in the water. It is able to heat the water but not to cool it down.
- One detector: the **thermometer**, that measure the temperature of the water at each iteration of the PID loop.

The user wants to keep the water at a temperature given by the **Set point**.

signals		
	status_sig	Emits in the status bar of the UI.
attributes		
	pid_controller Intance of DAQ_PID.	
	water_temp	Water temperature in °C at the current iteration.
	curr_time	Current time.
	curr_input	Current input. To be fed to the PID.
	curr_output	Current output from the PID.
methods		
	__init__	Initialize some attributes.

	update_settings	Get a parameter instance whose value has been modified by a user on the UI.
	ini_model	Initialize some settings.
	convert_input	Convert the measurements in the units to be fed to the PID (same dimensionality as the setpoint). Since we are in a mock model there are no measurements. We simulate the thermometer by returning the **water_temp** variable.
	convert_output arguments **output** output value from the PID **dt** time delay between two iterations of the loop	Convert the output of the PID in units to be fed into the actuator. Set **water_temp** to simulate the effect of the actuator (a heater): dT = output*dt and the effect of temperature dissipation dT = - dissipation*dt if output > 0: pass ???

PIDModelLIZARD class

pymodaq_pid_models/pymodaq_pid_models/models/PIDModelLIZARD.py

This class is very similar to PIDModelMichelsonDemo class.

This PID model is associated to the following physical system. In a Mach-Zender interferometer, one arm is dedicated to the production of XUV attosecond pulse trains, the other one is dedicated to the IR dressing.

As for the MichelsonDemo model, we get two sinusoidal signals with the interferometer length, that are in phase quadrature to produce an error signal. The main difference come from the physical nature of those signals, that come from modulation of two sidebands in a RABBIT setup, but the program is basically the same. Details can be found in the following reference[1].

1 Luttmann *et al.*, In situ sub-50 attosecond active stabilization of the delay between infrared and extreme ultraviolet light pulses (2020) https://arxiv.org/abs/2012.09528

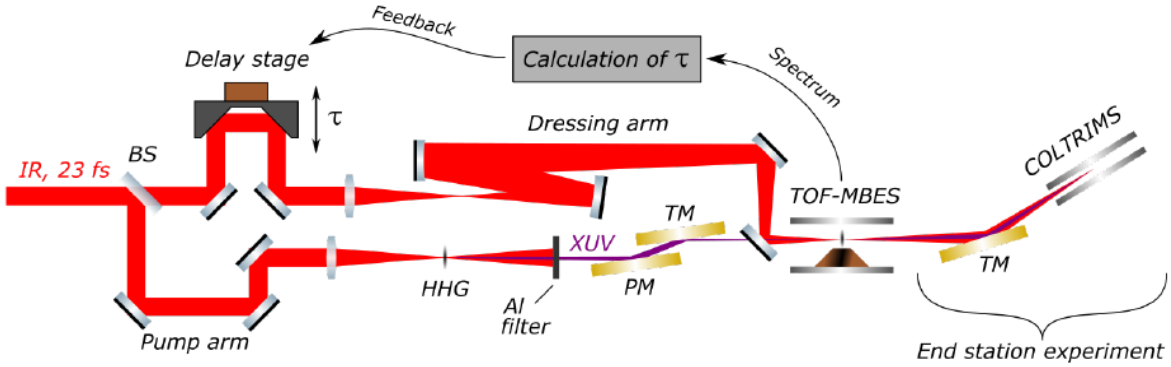


Illustration 14: Physical system associated to the PIDModelLIZARD class

- The actuator is a delay stage that is controlled through a National Instruments card (daq_move_NI... module).
- The detector is an oscilloscope (daq_1Dviewer_LecroyWaverunner6Zi module).

The modulated signals come from two ROIs defined in the scope spectrum (ROI_00 and ROI_01). A third ROI is used to define the baseline of the spectrum/the offset (ROI_02).

Workflow

Remarks

To access the value of a measurement from a viewer (let say a ROI mean value from a 1DViewer), one has to consider the Viewer1D.data_to_save_export attribute. The results of the measurements are stored in the data0D key of the dictionary, since the result of a measurement will be a scalar. Thus for example to access the value of ROI_00 measurement one use

signals			
parameters			
	Stabilization stabilization		
		Ellipsis phase (rad) ellipsis_phase (float)	Value of the last phase that has been measured using the modulated signals. This phase is not offset. It corresponds to the **t** angle in the figure of the ellipsis.

			Set by **get_phi_from_xy**.	
		Laser wavelength (microns) **laser_wl** (float)	Laser central wavelength in microns.	
		Correction sign **correction_sign** (int)	+1 or -1. Convenient to change the sign of the correction while the PID loop is running.	
		Offsets **offsets**		
			Phase time zero (rad) **time_zero_phase** (float)	Phase at time zero. Set at the end of the calibration scan (in **do_stabilization**).
	Calibration **calibration**			
		Do calibration **do_calibration** (boolean)	When this parameter is clicked, the **do_calibration** method is called.	
		Timeout **timeout** (int)		
		Calibration **calibration_move**		
			Start pos **start** (float)	Starting position of the calibration scan (micrometers).
			Stop pos **stop** (float)	Ending position of the calibration scan (micrometers).
			Step size **step** (float)	Step size of the scan (micrometers).
			Averaging **average** (int)	Number of times the scanning range is travelled to perform the calibration.
		Ellipse params **calibration_ellipse**		
			Dx **dx** (float)	Width of the fitted ellipse.
			Dy **dy** (float)	Height of the fitted ellipse.
			x0 **x0** (float)	Abscisse of the center of the fitted ellipse.
			y0 **y0** (float)	Ordinate of the center of the fitted ellipse.
			theta (°) **theta** (float)	Angle of the axis of the fitted ellipse with the x axis (see Figure).

attributes		
	calibration_scan_running boolean	Set to False in the constructor. It can only be set to True if the user has changed the 'do_calibration' parameter to True. This parameter is True only when the program is performing a calibration scan.
	stabilized_scan_running boolean	Set to False in the constructor. It can only be set to True if the user has changed the 'do_stabilized_scan' parameter to True. This parameter is True only when the program is running a stabilized scan.
	det_done_flag boolean	Set to False in the constructor. It can only be set to True if the det_done method is triggered.
	move_done_flag boolean	Set to False in the constructor. It can only be set to True if the move_done method is triggered.
	timeout_scan_flag boolean	Set to False in the constructor. It can only be set to True if the timeout method is triggered.
	curr_time float	Set in the constructor with time.perf_counter() (https://docs.python.org/3/library/time.html). "The reference point of the returned value is undefined, so that only the difference between the results of consecutive calls is valid."
	lsqe Instance of daq_utils.ellipses.LSqEllipse	Object useful to perform least square fitting of an ellipse and get the fitting parameters. Used in do_calibration.
	phase_buffer array of floats	Store the measurements of the phases (within [-pi,+pi]) that are done at each measurement. Thus an element is added each time the convert_input method is called.
	time_zero_phase float	Measured phase at time zero at the first call of convert_input. This phase is then offset to all the following measurements of ellipsis_phase to construct the delay_phase.
	ellipsis_phase	Measured phase at each iteration by

	float	get_phi_from_xy. It corresponds to t in the ellipsis figure.
	delay_phase float	Displayed in Current point box. This phase is the measured phase deduced from the two modulated signals. It is unwrapped, which means that it can be outside [-pi,+pi] because it keeps track of the past. It is also offset. Thus the zero of this phase corresponds to the time zero. There is a direct correspondence between this phase and the time delay between the pulses. Updated by convert_input. Its value is in radian. The value of this phase is valid as long as the PID loop is running and no phase jumps among pi are undergone between two measurements.
	actuator_absolute_position float	Initialized to 0 in the constructor. Set by move_done. This corresponds to the current position of the actuator in microns.
	curr_input float	Set to 0 in the constructor. Current measured phase position on the ellipsis.
	curr_output float	Set in convert_output. Current output of the pid module.
	dock_calib Instance of pyqtgraph.dockarea.Dock (https://pyqtgraph.readthedocs.io/en/latest/dockarea.html)	Widget that contains viewer_calib and viewer_ellipse. Itself is put into pid_controller.dock_area.
	viewer_calib Instance of Viewer1D.	Set in the constructor. Put into dock_calib. viewer_calib.x_axis is set in do_calibration.
	viewer_ellipse Instance of Viewer1D.	Set in the constructor. Put into dock_calib. Set in do_calibration. This viewer displays the XY representation of the two ROIs and the fit of the ellipse.
	pid_controller	This attribute is set while passing

	Instance of DAQ_PID.	through the constructor of PIDModelGeneric.
	timer Instance of PyQt5.QtCore.Qtimer (https://doc.qt.io/qt-5/qtimer.html)	"The QTimer class provides a high-level programming interface for timers. To use it, create a QTimer, connect its timeout() signal to the appropriate slots, and call start(). From then on, it will emit the timeout() signal at constant intervals." timer.timeout signal is connected to timeout in the constructor. The timer is started in wait_for_det_done and wait_for_move_done.
	detector_data Array of length the number of steps in the calibration scan, of arrays of length 2.	Set in do_calibration. It seems like for each step of the calibration, it gets two floats of data. Probably from two ROI integration ??
	detector_data_average Array of arrays of length 2.	Set in do_calibration. Averaging of detector_data if the calibration scan is travelled several times.
	calibration_scan_shape List of one int	Set in do_calibration. Used to initialize the PyTables array that stores the calibration scan.
	current_scan_path str	Path of the current scan group in the h5 tree struture.
	error float	Difference between the delay phase and the setpoint.
methods		
	__init__ arguments pid_controller instance of DAQ_PID.	Initialize some attributes. Probably useless to define widget_calib = QtWidgets.QWidget() and widget_ellipse since the contructor of Viewer1D accepts None argument. The timer is set single shot, which means that it will fire only once.
	ini_model	Defines all the actions to be performed on the initialized modules (main pid (DAQ_PID instance), actuators, detectors). Either here for specific things (ROI, ...) or within the preset of

		the current model.
	get_ellipse_fit arguments **center** (array of two floats) coordinates of the center of the ellipse **width** (float) width of the ellipse **height** (float) height of the ellipse **theta** (float) angle between the axis of the ellipse and the axis of the coordinate system (see Figure below)	Return two arrays of floats corresponding to the x and y coordinates of the points on the ellipse parametrized with the input arguments.
	update_settings arguments **param** (instance of pyqtgraph Parameter object) https://pyqtgraph.readthedocs.io/en/latest/parametertree/index.html	Get a Parameter instance whose value has been modified by a user on the UI. If the user has changed the 'do_calibration' parameter it will call **do_calibration**. If the user has changed the 'set_zero' parameter, it will set all the offsets to the current values and ??
	do_calibration	Called if the user changed the **'do_calibration'** parameter (see **update_settings**). This method performs the calibration scan and fit the ellipse obtained in an XY representation of the two signals to calibrate the parameters (center coordinates, width, height and theta). It displays the result of the fit in **viewer_ellipse**. Update the ellipse parameters in the parameters tree. Save the data (raw spectra from the oscilloscope + raw ROIs ???) in the h5 file. At the end of the scan, the actuator is sent back to its initial position (which is defined as the time zero). Then it acquires a last time the spectrum from the oscilloscope to measure the phase at time zero (set **time_zero_phase**).
	do_stabilized_scan	###
	__init_saving_stabilized_scan	Called in **do_stabilized_scan**. Prepare the saving of a stabilized scan.

		Initialize the h5 file structure,up to channel group, so it can welcome the data acquired during the stabilized scan. The PyTables arrays are initialized in **do_stabilized_scan** method. Update the **current_scan_path** attribute.
	timeout	Set **timeout_scan_flag** to True. Send the status signal *'Time out during acquisition'* and stop the timer.
	wait_for_det_done	Start the **timer**. Wait for **det_done_flag** or **timeout_scan_flag** to be True. If there is no acquisition before the timeout, the timer will emit and trigger the **timeout** method.
	det_done arguments data Ordereddict(Ndatas = 0, acq_time_s = 0, name = self.title (string), data0D = Ordereddict(CH1 = OrderedDict(), CH2 = OrderedDict(), …), data1D = Ordereddict(…), data2D = Ordereddict(…),)	Connected/disconnected in **do_calibration, which means that this method is used only during the calibration scan, which requires a direct access to the detectors ?? In that case never it will enter the if statement ??** Triggered by the **grab_done_signal** of the detector module (**DAQ_Viewer** object). It means that it is triggered each time an acquisition from the detector is ready. If the PID loop and the calibration are not running, it will call **convert_input**, else it sets **det_done_flag** to True. The argument sent to **convert_input** is of the form Ordereddict(viewer1D = data) In this PID model only one viewer is needed. But if several viewers would be needed to calculate the correction (e.g. two cameras used for beam pointing correction), then the argument sent to convert_input would be of the form Ordereddict(

		viewer1 = data_viewer1, viewer2 = data_viewer2, …)
	wait_for_move_done	Wait for **move_done_flag** or **timeout_scan_flag** to be True.
	move_done arguments **name** ?? **position** (float)	Triggered by the **move_done_signal** of the actuator module. Set **absolute_actuator_position** to **position**. Set **move_done_flag** to True. **CAUTION: this method is called ONLY while doing a calibration scan.**
	get_phi_from_xy arguments **x** (float) **y** (float) output **phi** (float) t in the Figure (in radians)	Performs a rotation of – theta on the (x,y) vector to be in the reference frame of the ellipse. Use the atan2 function to retrieve the corresponding angle (t in the Figure). Update **curr_phase**. Return the phase corresponding to the values of the two signals from the ROIs.
	interfero_from_phase arguments **phi** (float) phase correction to apply. Output from the pid module.	Called (only ?) by **convert_output**. Construct **phases** and perform the unwrap operation on it.
	convert_input arguments **measurements** OrderedDict(viewer1D = data) data = OrderedDict(name = viewer.title (string), data0D = OrderedDict(CH1 = OrderedDict(), CH2 = OrderedDict(), …), data1D = OrderedDict(…),	Called by **PIDRunner.start_PID**. Called by **det_done ???** Get the measurements from the detector and converts them to a phase measurement that corresponds to the position on the ellipsis (t in the following figure). Update **curr_input** and return its value (in microns).

	data2D =
	  OrderedDict(
	  …
	  ),
	)	

	convert_output arguments output (float) output value from the PID from which the model extract a value of the same unit as the actuator. In our case the pid module returns a phase that corresponds to the relative move to be done dt (float) ellapsed time in seconds since last call stab ??	Called by PIDRunner.start_PID. Convert the output of the PID in units to be fed into the actuator. The output of the pid module is a phase correction that has to be converted in microns for the piezo. The returned value is used to set PIDRunner.output_to_actuator. This value is sent as an absolute value order to the actuator ('move_Abs' command). This means that we have to know the current position of the actuator (absolute_actuator_position) and add the relative correction that is sent by the pid module.