Pharos Controller Documentation

Release 0.2

Aquiles Carattino

Aug 20, 2018

Contents:

The Pharos Controller can be installed as any other Python package by using the setup.py file provided. It can also be directly downloaded and run from within the folder. There are two main starting points: *start_gui* and *start_measurement*. The first starts a Window that allows to control the laser and perform scans, while the second is a simple example of how the same program can be utilized without user interfaces.

This guide will help you understand the logic behind the program and will quickly guide you on how to add new features or change the logic of the measurement. If you need help, you can always find more information on Github and on Readthedocs.

CHAPTER 1

General Structure

The program can be split into different layers. For the time being I will disregard the GUI part of the software, focusing only on the functioning logic behind it. The general principle of functioning is that a user should be able to plan the experiment ahead. He should be aware of what sensors and actuators are available, their ranges, how they are interconnected. For example the Pharos setup is very specific in the way signals are acquired; the NI card digitalizes a channel only when it receives a trigger from the scanning laser. In turn, the laser can be programmed to issue a trigger at very specific wavelengths, and thus the experiment itself consists on acquiring spectra.

The general principle of operation would be: first defining some configuration files for the devices and the experiment. These files are then read and interpreted by different Python objects. One of this objects is actually an *Experiment* object, that will hold all the logic, for example first preparing the acquisition card, then triggering the scan of the laser. Then the user can trigger different actions by calling different methods. The same structure can work for any kind of experiment and therefore it was brought as a separate package called Experimentor, that you can also find on Github.

CHAPTER 2

A quick walk-through

The first step is to create some yaml files with definitions. For example, one can start with the file *devices.yml*, in which every device is going to be specified; this means establishing to which port of the daq card a sensors is plugged, to which GPIB port of the computer the laser is connected, etc. The we define an *experiment.yml* file, in which we put every step of the experiment, from the initialization, to the scan and the finalization. Yaml files are quickly associated with dictionaries in Python and therefore they are also a useful resource when programming.

Then we need to write the experiment class. In this class we are going to load the devices and store them as a property. We are going to initialize them, i.e. we are going to start the communication with them. We are also going to load the experiment config file and we will generate methods to set up every step of a measurement. We have a *setup_scan* method as well as a *do_scan* method. I suggest you check them out to understand quickly what they are doing. The class is stored in *pharos.model.experiment*. You can also see a second example on how to build a confocal microscope with this logic.

Once everything is set up, we write our last script, the *start_measurement*. This script will call the different methods defined in the experiment class, for example, loading the devices, performing a scan, changing the parameters and performing a new scan. Something important to note is that after loading the YAML file to the experiment class, every property can be changed from outside the class. So, if we defined a laser sweep speed in the yaml file, we can easily alter it from outside:

```
experiment.laser.params['sweep_speed'] = New_Value
```

And the same is valid for all the different properties that are set through the Yaml file. The Experimentor project has several improvements in this regard, making it explicit which properties can be set and which properties can only be read.

With this logic in mind, a GUI is nothing but a way of passing values to the experiment class and a trigger of specific methods.

2.1 Adding a new feature

If you want to add a new feature, you can check this guide that walks you step by step. *Adding a new feature*

2.2 Understanding

If you want to understand different parts of the code, you can check:

2.2.1 Understanding the YAML files

The Pharos controller needs YAML files to configure some basic parameters. The YAML files are not just a whim, they make very explicit the different variables available to someone programming and therefore be used as a quick reference.

They also make possible quick changes in some default values, for example if a new photodiode is connected to the NI card, the laser changes from a GPIB connection to an USB connection, etc. Understanding the layout of these files is important but bear in mind that they are always read by a python program, that you are free to modify at your own discretion.

Devices.yml

The Pharos Controller was built upon an outdated idea of how to define devices. Basically everything in a setup is a device. The main difference is that some devices are connected to other instead of to a computer. Therefore, the program uses only one devices.yml file.

A general device such as a laser can be defined like this:

```
TSL-710:
  type: scan laser
  name: Santec Laser
  driver: controller.santec.tsl710/tsl710
  connection:
    type: GPIB
    port: 1
  defaults: config/devices_defaults.yml
```

Remember that per-se, yaml files do not do anything. They have to be interpreted later by python code. The main key, *TSL-710* identifies the device and has to be unique, however is not use in downstream code. What is used is the name, that also hase to be unique (and can be the same as the main key). The driver specifies which Python module has to be used to communicate with the device. That particular line would translate to Python code like this:

```
from controller.santec.tsl710 import tsl710
```

Todo: The code should rely on models that have standardized behavior, more than directly on drivers.

The *connection* block will be interpreted later on, and will be fed to the driver to make a connection. Knowing the type of communication the device has (GPIB, USB, Serial, etc.) and the port to which it is connected, is a responsibility of the user.

A daq card, such as an NI card can be defined as follows:

```
NI-DAQ:
  name: NI-DAQ
  type: daq
  model: ni
  number: 2
  driver: pharos.model.daq.ni/ni
  connection:
    type: daq
    port: 2
  trigger: external
  trigger_source: PFI0
```

Again, the name is the important parameter here. Moreover the last block specifies the kind of trigger we want to use for the DAQ.

Todo: Specifying the trigger when defining the DAQ is not a good idea, since in the same experiment there can be different needs. Monitoring a signal while aligning can be done without an external trigger. Being that the case, it would have made the code much more reusable if the trigger is defined elsewhere.

A photodiode connected to the ADQ can be defined like so:

```
PhotoDiode2:
  name: Photodiode 2
  port: 2
  type: analog
  mode: input
  description: Forward Intensity
  calibration:
    units: V
    slope: 1
    offset: 0
  limits:
    min: -2.5
    max: 2.5
  connection:
    type: daq
    device: NI-DAQ
```

Everything is more or less self explanatory. The calibration refers to how to convert from Volts to the units of the device. For example a piezo stage receives voltage, but it is transduced into distance. A thermocouple outputs voltage, but has a meaning of temperature, etc. The units here can be anything interpreted by the Pint module. The limits are the limits in the units given by the calibration. Limits are mandatory when dealing with NI cards, since it automatically chooses the best gain to optimize the digitalization range.

Lastly, the connection block explicitly states to which device it is connected.

Warning: This structure is cumbersome and was already superseded in the Experimentor program. If you are going to develop something new, I strongly suggest that you check that other program.

Measurement.yml

The measurement.yml file defines the basic structure of an experiment. It defines what parameters are needed, what detectors are recorded, etc. Whatever you add in here will be later available in the experiment class. Each main key of

the file should be different steps of your experiment. For example:

```
init:
  devices: 'config/devices.yml'
```

When initializing, the only important thing is to know where the file with the definition of the devices is. In the initialization step, we can use the information used in the property *devices* to initialize the appropriate drivers, etc.

Once we have our devices configured, we would like to do a scan. We call this a monitor, because in principle the scan can be repeating itself over and over:

```
monitor:
  laser:
    name: Santec Laser
    params:
      start_wavelength: 1491 nm
      stop_wavelength:  1510 nm
      wavelength_speed:  10 nm/s
      interval_trigger: 0.01 nm
      sweep_mode: ContOne
      wavelength_sweeps: 1
  detectors:
  - Photodiode Test
  - Photodiode 2
```

You see now that we define the laser we want to scan (in case there is more than one), and we refer to it by its name. We define some parameters and some detectors.

Note: Because of how the parser of the YAML file works, params is going to be a dictionary, while detectors is going to be a list.

If we want to fancy things up a bit, for example with 2-D scans instead of just 1D, we have to define which axis are we scanning. This axis should be another device and should have a range. If the device is not connected to a DAQ, we should also specify which property we would like to scan. We can do it like so:

```
scan:
  laser:
    name: Santec Laser
    params:
      start_wavelength: 1492 nm
      stop_wavelength:  1548 nm
      wavelength_speed:  50 nm/s
      interval_trigger: 0.01 nm
      sweep_mode: ContOne
      wavelength_sweeps: 1
  axis:
    device:
      name: Rotation 1
      range: [35deg, 55deg, 1deg]
      property: position
  detectors:
    - Photodiode Test
    - Photodiode 2
```

Imagine now that later on, you decide you need to have a shutter controlling a secondary laser. And you need that shutter to be closed after a scan, and to have a small delay before a new line scan starts. At least in what the configuration needs, you should only add these three lines:

```
shutter:
  port: PFI2
  delay: 100ms
```

And of course we should also include what to do once the experiment is over. For example we want to close the shutter of the laser, but not switch it off:

```
finish:
  TSL-10:
    shutter: False
```

2.2.2 Device class

In the Pharos context, everything that is part of an experiment is a device. The National Instruments card is a device, the photodiode is a device, etc. Therefore, a general class called Device is handy for defining some common behavior to all devices.

Devices are created with a dictionary of properties passed as only argument, and will be store as a property called *properties*. This shouldn't change over time, and therefore allows to user to go back to the pristine status of the device. On the other hand, the property *params* stores a dictionary of the latest values passed to a given device. In order to check whether a particular parameter was altered, it is sufficient to compare it with the properties.

Warning: The experimentor package makes the properties and the params behave in different ways, for example by

making them write-only if needed.

The most important methods available in a device are *initialize_driver* and *apply_values*. The first checks if a driver is part of the properties and initializes it according to the connection type. It is of paramount importance to note that this is heavily dependant on how *Lantz* initializes drivers. The only exception is the *daq* type of connection, that loads the specific module and passes the port as argument.

Todo: If a user needs to specify a driver that takes more arguments, or is of a different kind, the place to

look at is the *initialize_driver* method. Appropriate conditionals at this stage can take care of a great deal of different scenarios.

apply_values takes a dictionary with the structure 'property' => 'value' and directly applies it to the driver. Again this is assuming the use of Lantz drivers, where doing things like laser.wavelength = 1200*Q_('nm') makes sense.

Todo: In principle not all drivers behave in the same way. Interfacing devices with a model rather than with a

lower level driver may be a good idea.

Todo: It may be handy to expand the toolbox of devices. For example, being able to do a ramp.

class pharos.model.lib.device.**device**(*properties*)

Device object that takes as only argument a dictionary of properties. There are no required fields a-priory, except the method *initialize_driver* is used. In this case there should be a driver and a connection specified.

Parameters **properties** – Dictionary with all the properties of the device. Normally it is generated from a YAML file.

add_driver(*driver*)
Adds the driver of the device. It has to be initialized()

Parameters **driver** – driver of any class.

apply_values(*values*)
Iterates over all values of a dictionary and sets the values of the driver to it. The driver has to be set and has to be able to handle a *setattr(driver, key, value)* command. Lantz drivers in which a Feat with a setter was defined work out of the box. For other devices a workaround should be found.

Parameters **values** – a dictionary of parameters and desired values for those parameters

get_params()
Work in progress, already implemented in the Experimentor. A buffer function that allows to have read-only parameters provided that the decorator *@property* is used.

Returns Dictionary of parameters used for initializing the class.

initialize_driver()
Initializes the driver of the given device. If no driver is specified (for example when dealing with a device connected to a ADQ card, nothing happens. 'driver' and 'connection' should be available in the properties. The driver is initialized according to the Lantz way of doing it. Except if the ['connection']['type'] is 'daq', in which case the 'port' is passed as an argument to the initialization. This is how the NI class works, and can be used also for non DAQ devices, such as the rotation stage that takes the serial number as argument.

Todo: Nothing prevents a user from interacting directly with the driver without passing through the class

---

Device. This may give rise to unwanted programming patterns.

2.2.3 The Experiment Class

The experiment class holds all the logic for performing both complex and simple experiments. The Pharos Controller triggered the inception of the separation of logic from models from GUI, and therefore it possesses a slightly more primitive and down-to-earth approach. The Experimentor program has a more complete class, with specific inheritable methods.

Any user who is willing to develop new software should refer to the Experimentor and pehraps use the Pharos as an example but not as a model to follow to the line.

What follows in this page is the documentation of the Measurement class, taken directly from the source code.

2.2.4 Adding a new feature

Adding a new feature to the software is as important as being able to add a new device to the setup. Science changes, experiments change, and software should follow that behavior. In this page I will guide you step by step on how to add a shutter to the experiment. The shutter will close right after a line scan is done and will stay closed for a fixed time before a new scan starts. This prevents the sample from overheating due to the laser itself.

Identifying the hardware needs

The first step, even if redundant, is where most users fail. Identifying the need means understanding what do you want to do and how do you plan to do it. In the case of a Thorlabs Shutter, you can see that there are plenty of different

configurations. Let's assume you have the shutter SH1, and the cube for controlling it KSC101. The cube can be controlled from the computer via an USB connection; however this would imply finding drivers for it, etc. It can also be controlled via a TTL signal coming from a DAQ. For simplicity, in this example we are going to use the TTL signal.

Warning: It is **very** important that you read the manual and that you understand how the device operates. Some things are trivial (for example the cube has a button called enable), but some are not. Depending on the version of the cube you have, the behaviour of the shutter will be different. For example it can close automatically even if there is no TTL signal indicating to do so.

Identifying the software needs

The next step is to identify what has to be done. First step is adding the needed information to the pertinent YAML file. At this point is a matter of design taste. Do we prefer to add a new device or do we want to add it as a simple parameter for the experiment?

If we feel the shutter is going to become an important piece in the experiment or that we can have several shutters and want to address them in a convenient way, we should consider adding it as a device. On the other hand, if the shutter is just a step of the experiment, that we set once and forget about, we can directly include it as a step in the measurement process. In this guide we will opt for the latter.

Therefore, we add the next few lines to our YAML:

```
shutter:
  port: PFI1
  delay: 100ms
```

It is an extra entry, at the same level than the axis or the detectors. The port is where the device is plugged; the delay is how long it takes before the shutter opens and a new line scan starts.

Now we have the parameters we need, there is no more need of information. However, this information has to be interpreted and used by the experiment class. Let's pretend at this stage that there is no shutter routine added yet. If we check how the experiment works, we can quickly see that the *do_line_scan* method is the place to add our code. We want the shutter to be closed for a certain amount of time before doing a wavelength scan, and we want to close it right after.

The delay is easy to control with a *time.sleep()* command, but opening and closing the shutter has to be addressed differently. As you see from the YAML file, we just define the shutter with a port; this is assuming the shutter is connected to the NI card. Even though this gives less flexibility, if we are aware, a quick and dirty solution may be all what we are looking for. This means, we can directly control the shutter with the NI card, and we shouldn't iterate through all the DAQs to see which one is supposed to generate the TTL signal we need.

However, we see that the model used for controlling the NI card (found at *model.daq.ni*) does not possess a method for digital outputs.

Note: This entire example is based on a real world case. This means that everything that is described here was already implemented into the code.

To summarize, what we need is the following:

- Add to the YAML file the parameters we need
- Add to the NI card the control over digital outputs.
- Add to the experiment routine the delay and the open/close shutter
- Add the possibility to change the delay into the GUI

Expanding the NI toolbox

Starting with the NI class has the advantage that it is very easy to check if what we are doing makes sense. It also quickly provides insight into what can we be missing. Imagine you need to determine another parameter besides the port, etc. The outcome of this section should be that you can change the status of a given digital port from low to high or the opposite.

Relaying on pyDAQmx means that we can directly look into the documentation provided by NI. It is a vast amount of information that you should learn how to surf in order to survive. If you have a bit of experience with NI cards, you should probably know that the workflow starts by creating a task. It looks like this:

```
import PyDAQmx as nidaq
t = nidaq.Task()
```

Then we have to create a channel. The documentation gives us this information:

```
int32 DAQmxCreateDOChan (TaskHandle taskHandle, const char lines[], const char␣
↪nameToAssignToLines[], int32 lineGrouping);
```

This translates into pyDAQmx syntax like this:

```
t.CreateDOChan(channel, None, nidaq.DAQmx_Val_ChanPerLine)
```

We have removed the reference to the *taskHandle* because it is implicitly passed as the first argument of the task *t*. Second, we remove teh *DAQmx* from the name of the function. To form the *channel* string, we do the following:

```
channel = "Dev%s/%s" % (self.daq_num, port)
```

In this case the *self.daq_num* is storing the number of the DAQ; this is done at the instantiation of the NI class by the experiment class. The last variable is just how we are going to group the information. Since we are dealing with one channel at the time, we just done care.

Now we have a task that has a digital output channel available. Now we have to write to that output. Surfing through the documentation, within the Write Functions we find what we need:

```
int32 DAQmxWriteDigitalScalarU32 (TaskHandle taskHandle, bool32 autoStart, float64␣
↪timeout, uInt32 value, bool32 *reserved);
```

Again, this translates into pyDAQmx code:

```
t.WriteDigitalScalarU32(nidaq.bool32(True), 0, status, None)
```

We set it to autostart in order to skip the step of triggering the task. The rest you can understand it by reading the documentation. The only thing we need to define is what status is. If we want to use *True* as high and *False* as low, we can do it:

```
status = 0
if status:
    status = -1  # With this value, the digital output is set to High
```

The final function inside the class therefore looks like this:

```
def digital_output(self, port, status):
    """ Sets the port of the digital_output to status (either True or False)
    """
    t = nidaq.Task()
    channel = "Dev%s/%s" % (self.daq_num, port)
    t.CreateDOChan(channel, None, nidaq.DAQmx_Val_ChanPerLine)
```

```
    if status:
        status = -1  # With this value, the digital output is set to High
    else:
        status = 0
    print('Status: {}'.format(status))
    t.WriteDigitalScalarU32(nidaq.bool32(True), 0, status, None)
```

And now we have to test it by adding these lines to the end of the file (adapt to your own needs):

```
if __name__ == '__main__':
    a = ni(2)
    status = True
    while True:
        status = not status
        a.digital_output('PFI1', status)
        input()
```

Plug a multimeter to the port you want to check and run the code. Do you see it changing from high to low? Great job then!

Updating the experiment logic

Now that we know the NI can control the shutter via a TTL signal, we have to update the logic of our experiment. In the case of the Pharos, it means updating *model.experiment.measurement*. The details of this class are found in its own documentation page *The Experiment Class*

The way the 2D scan is done, is within a loop in the method *do_scan*:

```
for value in np.linspace(start, stop, num_points_dev, endpoint=True):
    [...]
    self.do_line_scan()
```

We can either open and close the shutter in this loop, or we can do it directly inside the *do_line_scan*. The second approach gives us the advantage that the behavior is going to be the same every time we trigger a line scan, not necessarily from within the *do_scan* method. Therefore that is the file we are going to edit.

The first thing we have to do is to get the shutter parameters that were loaded into the class through the YAML file. We also grab the NI DAQ device from within the devices dictionary:

```
shutter = self.scan['shutter']
ni_daq = self.devices['NI-DAQ']
```

Now we only have to close the shutter, wait a *delay* time, open the shutter and do a wavelength scan:

```
ni_daq.driver.digital_output(shutter['port'], False)
delay = shutter['delay']
time.sleep(delay)
ni_daq.driver.digital_output(shutter['port'], True)
laser.driver.execute_sweep()
ni_daq.driver.digital_output(shutter['port'], False)
```

The code is simplified for example purposes. But you see that the logic is quite clear. It really reflects what was defined as a thought experiment. Of course, there are few things we should also address. For example, the delay that we defained in the YAML has units, while the sleep function takes seconds. Fortunately, the program was built with *Quantities* all over the place. If you want to have fun, try the following code:

```
>>> from lantz import Q_
>>> dist = Q_('1nm')
>>> print(dist)
>>> dist_pm = dist.to('pm')
>>> print(dist_pm)
>>> dist_in = dist.m_as('in')
>>> print(dist_in)
```

The last couple of lines tells you that you can get the magnitude of a certain quantity in whichever units you want. So we do the same:

```
delay = Q_(shutter['delay'])
delay = delay.m_as('s')
time.sleep(delay)
```

Now it doesn't really matter if you specify the delay in seconds, milliseconds or hours. The important thing is that it should always be a time unit.

Finally, you should notice that the shutter and the digital output may not be synchronized. The shutter opens with a LOW=>HIGH edge, and closes with a HIGH=>LOW. If when you switch on, the shutter is closed and the port is High, you will be out of sync; if you set to port to high, there will be no edge and therefore nothing will happen. The way around it is to include a function that you trigger at the beginning of your program, the synchronizes the digital port and the shutter:

```
def sync_shutter(self):
    shutter = self.scan['shutter']
    ni_daq = self.devices['NI-DAQ']
    ni_daq.driver.digital_output(shutter['port'], False)
    time.sleep(0.2)
    ni_daq.driver.digital_output(shutter['port'], True)
    time.sleep(0.2)
    ni_daq.driver.digital_output(shutter['port'], False)
```

It is not an elegant solution, but it works. You do an antire cycle, finishing with the shutter closed. This guarantees that the digout and the shutter will be on the same page, when you set the digital output to High, the shutter opens, etc.

And now, you are done with everything. You just need to update your GUI to add support for this new feature.

Todo: Write the documentation on how to add it to the GUI.

Python Module Index

p

Index