# pymanip

***Release 0.3***

**Julien Salort**

**Jun 29, 2023**

# CONTENTS:

pymanip is the main package that we use for data acquisition and monitoring of our experimental systems in the Convection group at Laboratoire de physique de l'ENS de Lyon. It can be seen as an extension of the `fluidlab` module, which it heavily uses. It is available freely under the French CECILL-B license in the hope that it can be useful to others. But it is provided AS IS, without any warranty as to its commercial value, its secured, safe, innovative or relevant nature.

Unlike FluidLab, pymanip does not garantee any long term stability, and may change the API in the future without warning. However, some parts of the pymanip module may eventually be integrated into FluidLab, once they are stable enough.

The pymanip module is a set of tools for data acquisition and data management. Its goals are the following:

- management of experimental "sessions", for storing and retriving data, and useful live tools for experimental monitoring over long times, such as live plot, automated emails, and remote access of the live data, and also simple interrupt signal management;
- simplify access to FluidLab instrument classes;
- experimental implementation of asynchroneous video acquisition and DAQ acquisition;
- experimental extension of FluidLab interface and instrument classes with asynchroneous methods;
- miscellaneous CLI tools for saved session introspection, live video preview, live oscilloscope and spectrum analyser-style DAQ preview, and VISA/GPIB scanning.

CHAPTER

ONE

# INSTALLATION

## 1.1 Dependencies

pymanip requires FluidLab which it builds upon, as well as several third-party modules to communicate with instruments, either indirectly through FluidLab, or directly. Not all dependencies are declared in the *requirements.txt* file, because none of them are hard dependencies. It is possible to use pymanip or FluidLab with only a subset of these dependencies. Here is a dependence table, depending on what you want to do with the package. If a feature is available through FluidLab, the table indicates the FluidLab submodule that we are using.

| pymanip module | fluidlab module | third-pary modules |
|---|---|---|
| *pymanip.instruments* | fluidlab.instruments | gpib (linux-gpib python bindings), pymodbus, pyserial, pyvisa |
| *pymanip.daq.DAQmx* | fluidlab.daq.daqmx | PyDAQmx |
| *pymanip.daq.Scope* | | niScope |
| *pymanip.aiodaq.daqmx* | | nidaqmx |
| *pymanip.aiodaq.scope* | | niScope |
| *pymanip.aioinstruments* | fluidlab.instruments | Similar to *pymanip.instruments* |
| *pymanip.video* | | AndorNeo, pymba, opencv, pyqtgraph (optional) |
| *pymanip.asyncsession* | | aiohttp, aiohttp_jinja2, PyQt5 (optional) |
| pymanip.session | | h5py |

We also have our own bindings for some external libs, such as pymanip.video.pixelfly for PCO Library, pymanip.nisyscfg for National Instruments NISysCfg library.

## 1.2 Download and install

We recommand to install FluidLab and pymanip from the repositories, i.e. FluidLab from Heptapod and pymanip from GitHub, and to use the *-e* option of *pip install* to easily pull updates from the repositories:

```
$ hg clone https://foss.heptapod.net/fluiddyn/fluidlab
$ cd fluidlab
$ python -m pip install -e .
$ cd ..
$ git clone https://github.com/jsalort/pymanip.git
$ cd pymanip
$ python -m pip install -e .
```

However, it is also possible to install from PyPI:

```
$ python -m pip install fluidlab pymanip
```

## 1.3 Full installation with conda

Of course, it is possible to install the module, and its dependencies any way you like. For the record, I write here the procedure that we have been using in our lab for all our experimental room computers, using Anaconda. I am not advocating that it is better than another method. It installs many packages that are not dependencies of pymanip or fluidlab, but that we use regularly. We install as many packages as possible from conda, so that pip installs as little dependencies as possible. We also use black, flake8 and pre-commit hooks for the git repository.

Our base environment is setup like this:

```
$ conda create -n py37 python=3.7
$ conda activate py37
$ conda install conda
$ conda install jupyterlab jupyter_console widgetsnbextension qtconsole spyder numpy␣
↪matplotlib scipy
$ conda install h5py scikit-image opencv
$ conda install git
$ conda install cython numba aiohttp flake8 filelock flask markdown
$ python -m pip install --upgrade pip
$ python -m pip install PyHamcrest
$ python -m pip install clint pint aiohttp_jinja2
$ python -m pip install pyserial pydaqmx pyvisa pyvisa-py
$ python -m pip install pyqtgraph
$ python -m pip install llc black pre-commit
$ python -m pip install importlib_resources
```

Then fluiddyn, fluidimage, fluidlab and pymanip are installed from the repository, as indicated in the previous section. For the computer with video acquisition, the third-party library must first be installed, and then the corresponding third-party python package, as indicated in the table.

CHAPTER

TWO

# ASYNCHRONOUS SESSION

The *pymanip.asyncsession.AsyncSession* class provides tools to manage an asynchronous experimental session. It is the main tool that we use to set up monitoring of experimental systems, alongside FluidLab device management facilities. It will manage the storage for the data, as well as several asynchronous functions for use during the monitoring of the experimental system such as live plot of monitored data, regular control email, and remote HTTP access to the live data by human (connection from a web browser[1]), or by a machine using the *pymanip.asyncsession.RemoteObserver* class. It has methods to access the data for processing during the experiment, or post-processing after the experiment is finished.

Read-only access to the asyncsession data can be achieved with the *pymanip.asyncsession.SavedAsyncSession* class.

For synchronous session, one can still use the deprecated classes from `pymanip.session`, but these will no longer be updated, therefore the asynchronous session should now always be preferred.

## 2.1 Data management

AsyncSession objects can store three kinds of data:

- scalar variables monitored in time at possibly irregular time intervals. Scalar values of these variables are logged, in a "data logger" manner. They are suited to the monitoring of quantities over time when a regular measurement rate is not required. We call them "logged variables". Tasks can save values by calling the *pymanip.asyncsession.AsyncSession.add_entry()* method, and they can be later retrieved by calling the following methods *pymanip.asyncsession.AsyncSession.logged_variables()*, *pymanip.asyncsession.AsyncSession.logged_variable()*, *pymanip.asyncsession.AsyncSession.logged_data()*, *pymanip.asyncsession.AsyncSession.logged_first_values()*, *pymanip.asyncsession.AsyncSession.logged_last_values()*, *pymanip.asyncsession.AsyncSession.logged_data_fromtimestamp()*, or by using the *sesn[varname]* syntax shortcut which is equivalent to the `logged_variable()` method.
- scalar parameter defined once in the session. We call them "parameters". A program can save parameters with the *pymanip.asyncsession.AsyncSession.save_parameter()* method, and they can be later retrieved by calling the *pymanip.asyncsession.AsyncSession.parameter()*, *pymanip.asyncsession.AsyncSession.parameters()*, *pymanip.asyncsession.AsyncSession.has_parameter()* methods.
- non-scalar variables monitored in time at possibly irregular time intervals. A non-scalar value is typically a numpy array from an acquisition card or a frame from a camera. We call them "datasets". They can be saved with the *pymanip.asyncsession.AsyncSession.add_dataset()* method, and later be retrieved by the *pymanip.asyncsession.AsyncSession.dataset()*, *pymanip.asyncsession.AsyncSession.datasets()*, *pymanip.asyncsession.AsyncSession.dataset_names()*, *pymanip.asyncsession.*

[1] The default port is 6913, but it can be changed, or turned off by passing appropriate argument to *pymanip.asyncsession.AsyncSession.monitor()*.

*AsyncSession.dataset_last_data()*, *pymanip.asyncsession.AsyncSession.dataset_times()* methods.

## 2.2 Task management

The main entry point for an experimental session is the *pymanip.asyncsession.AsyncSession.monitor()* function which should be awaited from the main function of the program. The typical main structure of a program is then:

```
async def monitoring_task(sesn):
    some_value = await sesn.some_device.aget()
    sesn.add_entry(some_value=some_value)
    await sesn.sleep(10)

async def main(sesn):
    async with SomeDevice() as sesn.some_device:
        await sesn.monitor(monitoring_task)

with AsyncSession() as sesn:
    asyncio.run(main(sesn))
```

In this example, we see the use of the context managers, both for the `AsyncSession` object, and for the instrument object. The main experimental measurement lies in the `monitoring_task()` function, which should not be an explicit *for* loop. Indeed, the `sesn.monitor()` method will implement the *for* loop, with checks for the interruption signal.

Possible additionnal initialisation of devices can be added to the `main()` function. Possible additionnal initalisation of session variables can be added in the *with AsyncSession()* block. Note that all functions take the session object as its first argument. It is therefore possible to write the same code as a subclass of `AsyncSession`, but this is not strictly necessary, i.e.

```
class Monitoring(AsyncSession):

    async def monitoring_task(self):
        some_value = await self.some_device.aget()
        self.add_entry(some_value=some_value)
        await self.sleep(10)

    async def main(self):
        async with SomeDevice() as self.some_device:
            await self.monitor(self.monitoring_task)

with Monitoring() as mon:
    asyncio.run(mon.main())
```

The benefits of the asynchronous structure of the example program is clearer when plotting tasks, and email tasks are added, or when there are several concurrent monitoring tasks. The `main()` function may then become:

```
async def main(sesn):
    async with SomeDevice() as sesn.some_device:
        await sesn.monitor(monitoring_task_a,
                           monitoring_task_b,
                           sesn.plot(['var_a', 'var_b', 'var_c']),
                           sesn.send_email(from_addr='toto@titi.org',
```

(continues on next page)

(continued from previous page)

```
                                        to_addrs='tata@titi.org',
                                        delay_hours=2.0),
                            )
```

The useful pre-defined tasks are *pymanip.asyncsession.AsyncSession.send_email()*, *pymanip.asyncsession.AsyncSession.plot()* and *pymanip.asyncsession.AsyncSession.sweep()*.

## 2.3 Remote access

The *pymanip.asyncsession.AsyncSession.monitor()* method will set up a HTTP server for remote access to the live data. The server can be reached from a web browser, or from another script with an instance of the *pymanip.asyncsession.RemoteObserver* class.

The usage of the *RemoteObserver* class is straightforward:

```
observer = RemoteObserver('remote-computer.titi.org')
observer.start_recording()
# ... some time consuming task ...
data = observer.stop_recording()
```

*data* is a dictionnary which contains all the scalar variables saved on *remote-computer.titi.org* during the time consuming task.

## 2.4 Implementation

### 2.4.1 Asynchronous Session Module (pymanip.asyncsession)

This module defines two classes for live acquisition, *AsyncSession* and *RemoteObserver*. The former is used to manage an experimental session, the latter to access its live data from a remote computer. There is also one class for read-only access to previous session, *SavedAsyncSession*.

**class** pymanip.asyncsession.**AsyncSession**(*session_name=None*, *verbose=True*, *delay_save=False*, *exist_ok=True*, *readonly=False*, *database_version=-1*)

This class represents an asynchronous experiment session. It is the main tool that we use to set up monitoring of experimental systems. It will manage the storage for the data, as well as several asynchronous functions for use during the monitoring of the experimental system such as live plot of monitored data, regular control email, and remote HTTP access to the live data by human (connection from a web browser), or by a machine using the *pymanip.asyncsession.RemoteObserver* class. It has methods to access the data for processing during the experiment, or post-processing after the experiment is finished.

**Parameters**

- **session_name** (*str, optional*) – the name of the session, defaults to None. It will be used as the filename of the sqlite3 database file stored on disk. If None, then no file is created, and data will be temporarily stored in memory, but will be lost when the object is released.
- **verbose** (*bool, optional*) – sets the session verbosity, defaults to True
- **delay_save** (*bool, optional*) – if True, the data is stored in memory during the duration of the session, and is saved to disk only at the end of the session. It is not recommanded, but useful in cases where fast operation requires to avoid disk access during the session.

add_dataset(**args*, ***kwargs*)

This method adds arrays, or other pickable objects, as "datasets" into the database. They will hold a timestamp corresponding to the time at which the method has been called.

**Parameters**

- ***args** (*dict, optional*) – dictionnaries with name-value to be added in the database
- ****kwargs** (*object, optional*) – name-value to be added in the database

add_entry(**args*, ***kwargs*)

This methods adds scalar values into the database. Each entry value will hold a timestamp corresponding to the time at which this method has been called. Variables are passed in dictionnaries or as keyword-arguments. If several variables are passed, then they all hold the same timestamps.

For parameters which consists in only one scalar value, and for which timestamps are not necessary, use *pymanip.asyncsession.AsyncSession.save_parameter()* instead.

**Parameters**

- ***args** (*dict, optional*) – dictionnaries with name-value to be added in the database
- ****kwargs** (*float, optional*) – name-value to be added in the database

ask_exit(**args*, ***kwargs*)

This methods informs all tasks that the monitoring session should stop. Call this method if you wish to cleanly stop the monitoring session. It is also automatically called if the interrupt signal is received. It essentially sets the `running` attribute to False. Long user-defined tasks should check the `running` attribute, and abort if set to False. All other AsyncSession tasks check the attribute and will stop. The *sleep()* method also aborts sleeping.

dataset(*name*, *ts=None*, *n=None*)

This method returns the dataset recorded at the specified timestamp, and under the specified name.

**Parameters**

- **name** (*str*) – name of the dataset to retrieve
- **ts** (*float, optional*) – timestamp at which the dataset was stored, defaults to the timestamp of the last recorded dataset under that name
- **n** (*int, optional*) – select the nth dataset (in dataset timestamp chronological order)

**Returns**
the value of the recorded dataset

**Return type**
object

dataset_last_data(*name*)

This method returns the last recorded dataset under the specified name.

**Parameters**
**name** (*str*) – name of the dataset to retrieve

**Returns**
dataset value

**Return type**
object

**dataset_names**()

This method returns the names of the datasets currently stored in the session database.

**Returns**
names of datasets

**Return type**
set

**dataset_times**(*name*)

This method returns the timestamp of the recorded dataset under the specified name.

**Parameters**
**name** (*str*) – name of the dataset to retrieve

**Returns**
array of timestamps

**Return type**
numpy.ndarray

**datasets**(*name*)

This method returns a generator which will yield all timestamps and datasets recorded under the specified name. The rationale for returning a generator instead of a list, is that each individual dataset may be large.

**Parameters**
**name** (*str*) – name of the dataset to retrieve

**Exemple**

- To plot all the recorded datasets named 'toto'

```
>>> for timestamp, data in sesn.datasets('toto'):
>>>     plt.plot(data, label=f'ts = {timestamp-sesn.t0:.1f}')
```

- To retrieve a list of all the recorded datasets named 'toto'

```
>>> datas = [d for ts, d in sesn.datasets('toto')]
```

**async figure_gui_update**()

This method returns an asynchronous task which updates the figures created by the *pymanip.asyncsession.AsyncSession.plot()* tasks. This task is added automatically, and should not be used manually.

**get_version**()

Returns current version of the database layout.

**has_metadata**(*name*)

This method returns True if the specified metadata exists in the session database.

**has_parameter**(*name*)

This method returns True if specified parameter exists in the session database.

**Parameters**
**name** (*str*) – name of the parameter to retrieve

**Returns**
True if parameter exists, False if it does not

**Return type**
bool

**property initial_timestamp**

Session creation timestamp, identical to *pymanip.asyncsession.AsyncSession.t0*

**property last_timestamp**

Timestamp of the last recorded value

**logged_data**()

This method returns a name-value dictionnary containing all scalar variables currently stored in the session database.

**Returns**
all scalar variable values

**Return type**
dict

**logged_data_fromtimestamp**(*name*, *timestamp*)

This method returns the timestamps and values of a given scalar variable, recorded after the specified timestamp.

**Parameters**

- **name** (*str*) – name of the scalar variable to be retrieved.
- **timestamp** (*float*) – timestamp from which to recover values

**Returns**
the timestamps, and values of the specified variable

**Return type**
tuple of two numpy arrays

**logged_first_values**()

This method returns a dictionnary holding the first logged value of all scalar variables stored in the session database.

**Returns**
first values

**Return type**
dict

**logged_last_values**()

This method returns a dictionnary holding the last logged value of all scalar variables stored in the session database.

**Returns**
last logged values

**Return type**
dict

**logged_variable**(*varname*)

This method retrieve the timestamps and values of a specified scalar variable. It is possible to use the sesn[varname] syntax as a shortcut.

**Parameters**
**varname** (*str*) – name of the scalar variable to retrieve

**Returns**
timestamps and values

**Return type**
tuple (timestamps, values) of numpy arrays.

**Exemple**

```
>>> ts, val = sesn.logged_variable('T_Pt_bas')
```

`logged_variables()`

This method returns a set of the names of the scalar variables currently stored in the session database.

**Returns**
names of scalar variables

**Return type**
set

`metadata(name)`

This method retrives the value of the specified metadata.

`metadatas()`

This method returns all metadata.

`async monitor(*tasks, server_port=6913, custom_routes=None, custom_figures=None, offscreen_figures=False)`

This method runs the specified tasks, and opens a web-server for remote access and set up the tasks to run matplotlib event loops if necessary. This is the main method that the main function of user program should await for. It is also responsible for setting up the signal handling and binding it to the ask_exit method.

It defines a `running` attribute, which is finally set to False when the monitoring must stop. User can use the *ask_exit()* method to stop the monitoring. Time consuming user-defined task should check the `running` and abort if set to False.

**Parameters**

- ***tasks** (*co-routine function or awaitable*) – asynchronous tasks to run: if the task is a co-routine function, it will be called repeatedly until ask_exit is called. If task is an awaitable it is called only once. Such an awaitable is responsible to check that ask_exit has not been called. Several such awaitables are provided: *pymanip.asyncsession.AsyncSession.send_email()*, *pymanip.asyncsession.AsyncSession.plot()* and *pymanip.asyncsession.AsyncSession.sweep()*.
- **server_port** (*int, optional*) – the network port to open for remote HTTP connection, defaults to 6913. If None, no server is created.
- **custom_routes** (*co-routine function, optional*) – additionnal aiohttp routes for the web-server, defaults to None
- **custom_figures** (`matplotlib.pyplot.Figure`, optional) – additional matplotlib figure object that needs to run the matplotlib event loop
- **offscreen_figures** (*bool, optional*) – if set, figures are not shown onscreen

`async mytask(corofunc)`

This method repeatedly awaits the given co-routine function, as long as *pymanip.asyncsession.AsyncSession.ask_exit()* has not been called. Should not be called manually.

**parameter**(*name*)

This method retrieves the value of the specified parameter.

**Parameters**
**name** (*str*) – name of the parameter to retrieve

**Returns**
value of the parameter

**Return type**
float

**parameters**()

This method returns all parameter name and values.

**Returns**
parameters

**Return type**
dict

**async plot**(*varnames=None*, *maxvalues=1000*, *yscale=None*, ***, *x=None*, *y=None*, *fixed_ylim=None*, *fixed_xlim=None*)

This method returns an asynchronous task which creates and regularly updates a plot for the specified scalar variables. Such a task should be passed to *pymanip.asyncsession.AsyncSession.monitor()* or *pymanip.asyncsession.AsyncSession.run()*, and does not have to be awaited manually.

If varnames is specified, the variables are plotted against time. If x and y are specified, then one is plotted against the other.

**Parameters**

- **varnames** (*list or str, optional*) – names of the scalar variables to plot
- **maxvalues** (*int, optional*) – number of values to plot, defaults to 1000
- **yscale** (*tuple or list, optional*) – fixed yscale for temporal plot, defaults to automatic ylim
- **x** (*str, optional*) – name of the scalar variable to use on the x axis
- **y** (*str, optional*) – name of the scalar variable to use on the y axis
- **fixed_xlim** (*tuple or list, optional*) – fixed xscale for x-y plots, defaults to automatic xlim
- **fixed_ylim** (*tuple or list, optional*) – fixed yscale for x-y plots, defaults to automatic ylim

**print_description**()

Prints the list of parameters, logged variables and datasets.

**print_welcome**()

Prints informative start date/end date message. If verbose is True, this method is called by the constructor.

**run**(**tasks*, *server_port=6913*, *custom_routes=None*, *custom_figures=None*, *offscreen_figures=False*)

Synchronous call to *pymanip.asyncsession.AsyncSession.monitor()*.

**save_database**()

This method is useful only if delay_save = True. Then, the database is kept in-memory for the duration of the session. This method saves the database on the disk. A new database file will be created with the content of the current in-memory database.

This method is automatically called at the exit of the context manager.

**save_metadata**(**args*, ***kwargs*)

This method saves a text parameter into the database.

**save_parameter**(**args*, ***kwargs*)

This method saves a scalar parameter into the database. Unlike scalar values saved by the *pymanip.asyncsession.AsyncSession.add_entry()* method, such parameter can only hold one value, and does not have an associated timestamp. Parameters can be passed as dictionnaries, or keyword arguments.

**Parameters**

- ***args** (*dict, optional*) – dictionnaries with name-value to be added in the database
- ****kwargs** (*float, optional*) – name-value to be added in the database

**save_remote_data**(*data*)

This method saves the data returned by a *pymanip.asyncsession.RemoteObserver* object into the current session database, as datasets and parameters.

**Parameters**

**data** (*dict*) – data returned by the *pymanip.asyncsession.RemoteObserver* object

**async send_email**(*from_addr*, *to_addrs*, *host*, *port=None*, *subject=None*, *delay_hours=6*, *initial_delay_hours=None*, *use_ssl_submission=False*, *use_starttls=False*, *user=None*, *password=None*)

This method returns an asynchronous task which sends an email at regular intervals. Such a task should be passed to *pymanip.asyncsession.AsyncSession.monitor()* or *pymanip.asyncsession.AsyncSession.run()*, and does not have to be awaited manually.

**Parameters**

- **from_addr** (*str*) – email address of the sender
- **to_addrs** (*list of str*) – email addresses of the recipients
- **host** (*str*) – hostname of the SMTP server
- **port** (*int, optional*) – port number of the SMTP server, defaults to 25
- **delay_hours** (*float, optional*) – interval between emails, defaults to 6 hours
- **initial_delay_hours** (*float, optional*) – time interval before the first email is sent, default to None (immediately)

**async server_current_ts**(*request*)

This asynchronous method returns the HTTP response to a request for JSON with the current server time. Should not be called manually.

**async server_data_from_ts**(*request*)

This asynchronous method returns the HTTP response to a request for JSON with all data after the specified timestamp. Should not be called manually.

**async server_get_parameters**(*request*)

This asynchronous method returns the HTTP response to a request for JSON data of the session parameters. Should not be called manually.

**async server_logged_last_values**(*request*)

This asynchronous method returns the HTTP response to a request for JSON data of the last logged values. Should not be called manually.

**async server_main_page**(*request*)

This asynchronous method returns the HTTP response to a request for the main html web page. Should not be called manually.

**async server_plot_page**(*request*)

This asynchronous method returns the HTTP response to a request for the HTML plot page. Should not be called manually.

**async sleep**(*duration*, *verbose=True*)

This method returns an asynchronous task which waits the specified amount of time and prints a countdown. This should be called with verbose=True by only one of the tasks. The other tasks should call with verbose=False. This method should be preferred over asyncio.sleep because it checks that *pymanip.asyncsession.AsyncSession.ask_exit()* has not been called, and stops waiting if it has. This is useful to allow rapid abortion of the monitoring session.

**Parameters**

- **duration** (*float*) – time to wait
- **verbose** (*bool, optional*) – whether to print the countdown, defaults to True

**async sweep**(*task*, *iterable*)

This methods returns an asynchronous task which repeatedly awaits a given co-routine by iterating the specified iterable. The returned asynchronous task should be passed to *pymanip.asyncsession.AsyncSession.monitor()* or *pymanip.asyncsession.AsyncSession.run()*, and does not have to be awaited manually.

This should be used when the main task of the asynchronous session is to sweep some value. The asynchronous session will exit when all values have been iterated. This is similar to running a script which consists in a synchronous for-loop, except that other tasks, such as remote access, live-plot and emails can be run concurrently.

**Parameters**

- **task** (*function*) – the co-routine function to repeatedly call and await
- **iterable** (*list*) – values to pass when calling the function

**Example**

```
>>> async def balayage(sesn, voltage):
>>>     await sesn.generator.vdc.aset(voltage)
>>>     await asyncio.sleep(5)
>>>     response = await sesn.multimeter.aget(channel)
>>>     sesn.add_entry(voltage=voltage, response=response)
>>>
>>> async def main(sesn):
>>>     await sesn.monitor(sesn.sweep(balayage, [0.0, 1.0, 2.0]))
>>>
>>> asyncio.run(main(sesn))
```

**property t0**

Session creation timestamp

**class** pymanip.asyncsession.**SavedAsyncSession**(*session_name*, *verbose=True*)

class pymanip.asyncsession.**RemoteObserver**(*host*, *port=6913*)

This class represents remote observers of a monitoring session. It connects to the server opened on a remote computer by *pymanip.asyncsession.AsyncSession.monitor()*. The aim of an instance of RemoteObserver is to retrieve the data saved into the remote computer session database.

**Parameters**

- **host** (*str*) – hostname of the remote compute to connect to
- **port** (*int, optional*) – port number to connect to, defaults to 6913

**_get_request**(*apiname*)

Private method to send a GET request for the specified API name

**_post_request**(*apiname*, *params*)

Private method to send a POST request for the specified API name and params

**get_last_values**()

This method retrieve the last set of values from the remote monitoring session.

**Returns**
scalar variable last recorded values

**Return type**
dict

**start_recording**()

This method establishes the connection to the remote computer, and sets the start time for the current observation session.

**stop_recording**(*reduce_time=True*, *force_reduce_time=True*)

This method retrieves all scalar variable data recorded saved by the remote computer since *pymanip.asyncsession.RemoteObserver.start_recording()* established the connection.

**Parameters**

- **reduce_time** (*bool, optional*) – if True, try to collapse all timestamp arrays into a unique timestamp array. This is useful if the remote computer program only has one call to add_entry. Defaults to True.
- **force_reduce_time** (*bool, optional*) – bypass checks that all scalar values indeed have the same timestamps.

**Returns**
timestamps and values of all data saved in the remote computed database since the call to *pymanip.asyncsession.RemoteObserver.start_recording()*

**Return type**
dict

CHAPTER

THREE

# INSTRUMENT DRIVERS

The instrument drivers in `pymanip` are directly those of `fluidlab.instruments`. An instance object represents an actual physical instrument, and the features that can be read or set are represented as instance attributes. These attributes have consistant names for all our instrument drivers, summarized in the table below

| Physical measurement | Attribute name |
|---|---|
| DC Voltage | vdc |
| AC Voltage | vrms |
| DC Current | idc |
| AC Current | irms |
| 2-wire impedence | ohm |
| 4-wire impedence | ohm_4w |
| Signal phase shift | angle |
| Frequency | freq |
| On Off switch | onoff |
| Pressure | pressure |
| Temperature | temperature |
| Setpoint | setpoint |

Some device may have specific feature name in special cases, but we try to keep using similar names for similar features. Each feature then can be accessed by `get()` and `set()` methods, as appropriate. The `get()` will query the instrument for the value. The `set()` method will set the value. It is a design choice to use getters and setters, instead of python properties, to make the actual communication command more explicit.

For example to read a voltage on a multimeter:

```
Vdc = multimeter.vdc.get()
```

And to set the voltage setpoint to 1 volt on a power supply:

```
powersupply.vdc.set(1.0)
```

Unless otherwise specified, `vdc.get()` will always read an actual voltage, and not the voltage setpoint. This is a design choice because we think the users should know what setpoint they have set in the general case. In case it is necessary to actually query the instrument for its setpoint, an additionnal `setpoint` attribute may be defined.

The implementation details of the instrument drivers, and how they are mixed with the interface and feature classes is described in the `fluidlab.instruments` module documentation.

*`pymanip.instruments`* defines shortcut classes, as well as an asynchronous extension to the `fluidlab.instruments` classes.

## 3.1 Shortcuts to FluidLab Instrument classes

The instrument classes are the basic objects that we use to communicate with various scientific instruments. They are implemented in the FluidLab project in the `fluidlab.instruments` module.

The *`pymanip.instruments`* module, along with the *list_instruments cli command* are simple tools, designed to simplify access to these classes.

Indeed, each instrument class in FluidLab is defined in separate sub-modules, which can result in long and convoluted *import* statements, such as

```
from fluidlab.instruments.chiller.lauda import Lauda
from fluidlab.instruments.multiplexer.agilent_34970a import Agilent34970a
from fluidlab.instruments.motor_controller.newport_xps_rl import NewportXpsRL
```

The *`pymanip.instruments`* module simplifies these import statements and allows to write

```
from pymanip.instruments import Lauda, Agilent34970a, NewportXpsRL
```

This is of course less efficient because this means that *all* instruments classes are actually loaded, but it makes the script easier to write and read.

The names of the available instrument classes can be conveniently obtained from the command line

```
$ python -m pymanip list_instruments
```

## 3.2 Instrument classes

### 3.2.1 Instruments module (`pymanip.instruments`)

This module auto-imports all the instruments classes from `fluidlab.instruments`:

| | |
|---|---|
| StanfordSR830([interface]) | Driver for Lock-in Amplifier Stanford 830. |
| Agilent34970a([interface]) | Driver for the multiplexer Agilent 34970A. |
| Keithley2700(*args, **kwargs) | Driver for the multiplexer Keithley 2700 Series |
| Lakeshore224([interface]) | Driver for the Lakeshore Model 224 Temperature Monitor |
| Keithley705([interface]) | Driver for the multiplexer Keithley 705* |
| Cryocon24c([interface]) | |
| HP34401a([interface]) | Driver for the multimeter HP 34401a. |
| ThorlabsS110(serialPort) | |
| NeelLHLM([interface]) | |
| Keithley2400([interface]) | Driver for the sourcemeter Keithley 2400. |
| Agilent6030a([interface]) | Driver for the power supply Agilent 6030a |
| IsoTechIPS2303S([baudrate]) | Driver for the power supply IPS 2303S. |
| TdkLambda([interface]) | Driver for the power supply TDK Lambda |
| TtiCpx400dp([interface]) | Driver for the power supply TTI CPX400DP Dual 420 watt DC Power Supply |
| HP_6653A([interface]) | A driver for the power supply HP_6653A |
| XantrexXDC300([interface]) | Driver for the power supply Xantrex XDC300 |
| Agilent33220a([interface]) | A driver for the function generator Agilent 33220A |
| Agilent33500b([interface]) | Minimal implementation of a driver for the Agilent 33500B. |
| HP33120a([interface]) | Driver for the function generator Hewlett-Packard 33120A |
| TektronixAFG3022b([interface]) | A driver for the function generator Tektronix AFG 3022 B. |
| TtiTsx3510p([interface]) | A driver for the function generator Thurlby Thandar Instruments (TTI) TSX3510P. |
| StanfordDS360([interface]) | A driver for the ultra low distorsion wave generator DS360 |
| Lauda([interface]) | |
| Julabo([interface]) | |
| AgilentDSOX2014a([interface]) | Driver for the oscilloscope Agilent DSOX2014a. |
| FurnessFCO318([interface]) | |
| PfeifferMaxiGauge(serialPort[, debug]) | |
| NewportXpsRL(ip_address[, port]) | |

## 3.3 Asynchronous Extension of FluidLab Instrument Classes

FluidLab Instrument classes, which can be accessed from the *pymanip.instruments* module are written in a synchronous manner, i.e. all calls are blocking. However, most of the time, the program essentially waits for the instrument response.

While most scientific instruments are not designed for concurrent requests, and most library, such as National Instruments VISA, are not thread-safe, it is still desirable to have an asynchronous API for the instrument classes. This allows to keep other tasks running while waiting for (possibly long) delays. The other tasks include: communication with other types of devices on other types of boards, refreshing the GUI and responding to remote access requests.

This asynchronous extension is implemented in the *pymanip.aioinstruments* and *pymanip.interfaces.aiointer* modules. They define subclasses of FluidLab interface, feature are instrument classes. This is currently highly experimental, so please use it at your own risk. However, it may eventually be merged into FluidLab when it gets more mature.

From the user perspective, the usage of asynchronous instruments is relatively straightforward for those who already know how to use FluidLab Instrument classes.

The synchronous program

```
from pymanip.instruments import Agilent34970a

def main():
    with Agilent34970a('GPIB0::9::INSTR') as a:
        R1, R2 = a.ohm_4w.get((101, 102))
    return R1, R2
```

becomes

```
from pymanip.aioinstruments import AsyncAgilent34970a

async def main():
    async with AsyncAgilent34970a('GPIB0::9::INSTR') as a:
        R1, R2 = await a.ohm_4w.aget((101, 102))
    return R1, R2
```

The asynchronous instrument subclasses all have the "Async" prefix in their names. Asynchronous context manager *must* be used instead of the classical context manager because some specific initialisation may be done in the asynchronous interface `__aenter__()` method. All the features have the same name as in the synchronous class, but they have `aget()` and `aset()` co-routine methods instead of `get()` and `set()` methods.

## 3.4 Asynchronous instruments implementation

### 3.4.1 Asynchronous extension of `fluidlab.interfaces.QueryInterface` (`pymanip.interfaces.aiointer`)

This module defines *AsyncQueryInterface* as the default subclass for `fluidlab.interfaces.QueryInterface`. The default implementation simply runs the methods of `QueryInterface` into an executor, therefore in a separate thread. Because the parent class is probably not thread-safe, each call is protected by a lock to prevent concurrent calls to the instrument. However, the main thread is released and other tasks can be run on other instruments.

The methods that are defined in this way are `__aenter__()`, `__aexit__()`, `_aread()`, `_awrite()`, `_aquery()` and `await_for_srq()`. The higher lever co-routine methods `aread()`, `awrite()` and `aquery()` simply checks that the

interface is opened, and then awaits the low-level method, in a similar fashion as in the `QueryInterface` class.

Therefore, concrete subclass such as *pymanip.interfaces.aioserial.AsyncSerialInterface*, or *pymanip.interfaces.aiovisa.AsyncVISAInterface* only have to override the low-level co-routine methods (if necessary).

**class** pymanip.interfaces.aiointer.**AsyncQueryInterface**

This class represents an asynchronous Query interface. It is a subclass of the synchronous QueryInterface defined in FluidLab. The input parameters are those of QueryInterface.

Concrete subclasses may replace the `lock` attribute, and replace it by a global board lock if necessary.

**async _aquery**(*command*, *time_delay=0.1*, ***kwargs*)

Low-level co-routine to write/read a query. This method does not check whether the interface is opened. There are two cases:

- if the QueryInterface has a `_query()` method, then the interface lock is acquired and the `_query()` method is run in an executor;
- otherwise, the *awrite()* and *aread()* co-routine methods are used.

**async _aread**(**args*, ***kwargs*)

Low-level co-routine to read data from the instrument. This method does not check whether the interface is opened. In this basic class, it simply acquires the interface lock and runs the `_read()` method in an executor.

**async _awrite**(**args*, ***kwargs*)

Low-level co-routine to send data to the instrument. This method does not check whether the interface is opened. In this basic class, it simply acquires the interface lock and runs the `_write()` method in an executor.

**async aquery**(*command*, *time_delay=0.1*, ***kwargs*)

This co-routine method queries the instrument. The parameters are identical to those of the `query()` method.

**async aread**(**args*, ***kwargs*)

This co-routine method reads data from the instrument. The parameters are identical to the `read()` method.

**async awrite**(**args*, ***kwargs*)

This co-routine method writes data to the instrument. The parameters are identical to the `write()` method.

### 3.4.2 Asynchronous extension of `SerialInterface` (`pymanip.interfaces.aioserial`)

This module defines *AsyncSerialInterface* as a subclass of `fluidlab.interfaces.serial_inter.SerialInterface` and `pymanip.interface.aiointer.AsyncQueryInterface`.

**class** pymanip.interfaces.aioserial.**AsyncSerialInterface**(*port*, *baudrate=9600*, *bytesize=8*, *parity='N'*, *stopbits=1*, *timeout=1*, *xonxoff=False*, *rtscts=False*, *dsrdtr=False*, *eol=None*, *multilines=False*, *autoremove_eol=False*)

This class is an asynchronous extension of `fluidlab.interfaces.serial_inter.SerialInterface`. It inherits all its methods from the parent classes.

**async _areadlines**(**args*, ***kwargs*)

Low-level co-routine to read lines from the instrument.

**async areadlines**(**args*, ***kwargs*)

This co-routine method reads lines of data from the instrument.

### 3.4.3 Asynchronous extension of fluidlab.interfaces.VISAInterface (pymanip.interfaces.aiovisa)

This module defines *AsyncVISAInterface* as a subclass of fluidlab.interfaces.visa_inter.VISAInterface and pymanip.interface.aiointer.AsyncQueryInterface.

**class** pymanip.interfaces.aiovisa.**AsyncVISAInterface**(*resource_name*, *backend=None*)

This class is an asynchronous extension of fluidlab.interfaces.visa_inter.VISAInterface. The parameters are the same as those of the fluidlab.interfaces.visa_inter.VISAInterface class.

**async await_for_srq**(*timeout=None*)

This co-routine method acquires the interface lock and run wait_for_srq in an executor.

### 3.4.4 Asynchronous instruments module (pymanip.aioinstruments)

This module auto-imports all the asynchronous instrument classes.

### 3.4.5 Asynchronous Instrument drivers (pymanip.aioinstruments.aiodrivers)

This module defines a subclass of fluidlab.instruments.drivers.Driver where the QueryInterface attribute is replaced by the corresponding AsyncQueryInterface instance. The asynchronous context manager is bound to the interface asynchronous context manager.

pymanip.aioinstruments.aiodrivers.**interface_from_string**(*name*, *default_physical_interface=None*, ***kwargs*)

This function is similar to fluidlab.interfaces.interface_from_string() except that it returns an instance of AsyncQueryInterface instead of QueryInterface.

**class** pymanip.aioinstruments.aiodrivers.**AsyncDriver**(*interface=None*)

This class is an asynchronous extension of fluidlab.instruments.drivers.Driver.

### 3.4.6 Asynchronous Instrument features (pymanip.aioinstruments.aiofeatures)

Asynchronous extension of fluidlab instrument features. The main difference is that they define aget() and aset() co-routine methods. The original get() and set() are not overridden, and may still be used.

**class** pymanip.aioinstruments.aiofeatures.**AsyncWriteCommand**(*name*, *doc=''*, *command_str=''*)

**_build_driver_class**(*Driver*)

Add a "write function" to the driver class

**class** pymanip.aioinstruments.aiofeatures.**AsyncQueryCommand**(*name*, *doc=''*, *command_str=''*, *parse_result=None*)

**_build_driver_class**(*Driver*)

Add a "query function" to the driver class

class pymanip.aioinstruments.aiofeatures.**AsyncValue**(*name*, *doc=''*, *command_set=None*, *command_get=None*, *check_instrument_value=True*, *pause_instrument=0.0*, *channel_argument=False*)

class pymanip.aioinstruments.aiofeatures.**AsyncNumberValue**(*name*, *doc=''*, *command_set=None*, *command_get=None*, *limits=None*, *check_instrument_value=True*, *pause_instrument=0.0*, *channel_argument=False*)

class pymanip.aioinstruments.aiofeatures.**AsyncFloatValue**(*name*, *doc=''*, *command_set=None*, *command_get=None*, *limits=None*, *check_instrument_value=True*, *pause_instrument=0.0*, *channel_argument=False*)

class pymanip.aioinstruments.aiofeatures.**AsyncBoolValue**(*name*, *doc=''*, *command_set=None*, *command_get=None*, *check_instrument_value=True*, *pause_instrument=0.0*, *channel_argument=False*, *true_string='1'*, *false_string='0'*)

class pymanip.aioinstruments.aiofeatures.**AsyncIntValue**(*name*, *doc=''*, *command_set=None*, *command_get=None*, *limits=None*, *check_instrument_value=True*, *pause_instrument=0.0*, *channel_argument=False*)

class pymanip.aioinstruments.aiofeatures.**AsyncRegisterValue**(*name*, *doc=''*, *command_set=None*, *command_get=None*, *keys=None*, *default_value=0*, *check_instrument_value=True*, *pause_instrument=0.0*, *channel_argument=False*)

### 3.4.7 Asynchronous IEC60488 instrument driver (pymanip.aioinstruments.aioiec60488)

This module defines an asynchronous subclass of fluidlab.instruments.iec60488.IEC60488.

class pymanip.aioinstruments.aioiec60488.**AsyncIEC60488**(*interface=None*)

**aclear_status = <function AsyncWriteCommand._build_driver_class.<locals>.func>**

**aquery_esr = <function AsyncQueryCommand._build_driver_class.<locals>.func>**

**aquery_stb = <function AsyncQueryCommand._build_driver_class.<locals>.func>**

**aquery_identification = <function AsyncQueryCommand._build_driver_class.<locals>.func>**

areset_device = <function AsyncWriteCommand._build_driver_class.<locals>.func>

aperform_internal_test = <function
AsyncQueryCommand._build_driver_class.<locals>.func>

await_till_completion_of_operations = <function
AsyncWriteCommand._build_driver_class.<locals>.func>

aget_operation_complete_flag = <function
AsyncQueryCommand._build_driver_class.<locals>.func>

await_continue = <function AsyncQueryCommand._build_driver_class.<locals>.func>

async aclear_status()

Clears the data status structure

async aget_operation_complete_flag()

Get operation complete flag

async aperform_internal_test()

Perform internal self-test

async aquery_esr()

Query the event status register

async aquery_identification()

Identification query

async aquery_stb()

Query the status register

async areset_device()

Perform a device reset

async await_continue()

Wait to continue

async await_till_completion_of_operations()

Return "1" when all operation are completed

event_status_enable_register =
<pymanip.aioinstruments.aiofeatures.AsyncRegisterValue object>

status_enable_register = <pymanip.aioinstruments.aiofeatures.AsyncRegisterValue
object>

CHAPTER

FOUR

# VIDEO ACQUISITION

The *pymanip.video* module provides tools to help with camera acquisition. We use the third-party `pymba` module bindings to the AVT Vimba SDK for AVT cameras, the third-party `pyAndorNeo` module bindings to the Andor SDK3 library for the Andor Camera, the third-party `pyueye` module bindings to the IDS ueye library, the Python module provided by Ximea for the Ximea cameras, and PyVCAM wrapper for Photometrics camera.

We wrote our own bindings to the Pixelfly library for the PCO camera. Beware that the code works for us, but there is no garantee that it will work with your camera models.

The idea was for us to be able to switch cameras, without having to change much of the acquisition code. So we define an abstract *pymanip.video.Camera* base class, and all concrete sub-classes follow the exact same user API. The methods allow to start video acquisition, in a manner consistent with our needs, and also provides a unified live preview API. It also makes it relatively straightforward to do simultaneous acquisition on several cameras, even if they are totally different models and brands and use different underlying libraries.

The useful concrete classes are given in this table:

| Camera type | Concrete class |
|---|---|
| AVT | *pymanip.video.avt.AVT_Camera* |
| PCO | `pymanip.video.pco.PCO_Camera` |
| Andor | `pymanip.video.andor.Andor_Camera` |
| IDS | *pymanip.video.ids.IDS_Camera* |
| Ximea | `pymanip.video.ximea.Ximea_Camera` |
| Photometrics | `pymanip.video.photometrics.Photometrics_Camera` |

They all are sub-classes of the *pymanip.video.Camera* abstract base class. Most of the user-level useful documentation lies in the base class. Indeed, all the concrete implementation share the same API, so their internal methods are implementation details.

In addition, a high-level class, *pymanip.video.session* is provided to build simple video acquisition scripts, with possible concurrent cameras trigged by a function generator.

## 4.1 Simple usage

### 4.1.1 Context manager

First of all, the *Camera* object uses context manager to ensure proper opening and closing of the camera connection. This is true for all the methods, synchronous or asynchronous. Therefore, all our example will have a block like this:

```
from pymanip.video.avt import AVT_Camera as Camera

with Camera() as cam:

    # ... do something with the camera ...
```

And in all cases, switching to another camera, for example to a PCO camera, only requires to change the import statement, e.g.

```
from pymanip.video.pco import PCO_Camera as Camera

with Camera() as cam:

    # ... do something with the camera ...
```

### 4.1.2 Simple high-level acquisition function

The easiest way to do an image acquisition with `pymanip` is to use the high-level *`acquire_to_files()`* method. It is a one-liner that will start the camera, acquire the desired number of frames, and save them on the disk. It is enough for very simple acquisition programs. Parameters of the acquisition can be set with dedicated methods beforehands, such as,

- `set_exposure_time()`
- `set_trigger_mode()`
- `set_roi()`
- `set_frame_rate()`

The advantage over direct calls to modules like `pymba` or `AndorNeo` is that it is straightforward to switch camera, without changing the user-level acquisition code.

A simple acquisition script, for use with an external GBF clock, would be:

```
import numpy as np
from pymanip.video.avt import AVT_Camera as Camera

acquisition_name = "essai_1"
nframes = 3000

with Camera() as cam:
    cam.set_trigger_mode(True) # set external trigger
    count, dt = cam.acquire_to_files(
        nframes,
        f"{acquisition_name:}/img",
        dryrun=False,
        file_format="png",
        compression_level=9,
        delay_save=True,
    )

dt_avg = np.mean(t[1:] - t[:-1])
print("Average:", 1.0 / dt_avg, "fps")
```

The returned image is an instance of *MetadataArray*, which is an extension of `numpy.ndarray` with an additionnal *metadata* attribute. When possible, the *Camera* concrete subclasses set this metadata attribute with two key-value pairs:

- "timestamp";
- "counter".

The "timestamp" key is the frame timestamp in camera clock time. The "counter" key is the frame number.

### 4.1.3 Generator method

It is sometimes desirable to have more control over what to do with the frames. In this case, we can use the *acquisition()* generator method. The parameters are similar to the *acquire_to_files()* method, except that the frame will be yielded by the generator, and the user is responsible to do the processing and saving.

The previous example can be rewritten like this:

```
import numpy as np
import cv2
from pymanip.video.avt import AVT_Camera as Camera

acquisition_name = "essai_1"
nframes = 3000
compression_level = 9
params = (cv2.IMWRITE_PNG_COMPRESSION, compression_level)
t = np.zeros((nframes,))

with Camera() as cam:

    for i, frame in enumerate(cam.acquisition(nframes)):
        filename = f"{acquisition_name:}/img-{i:04d}.png"
        cv2.imwrite(filename, frame, params)
        t[i] = frame.metadata["timestamp"].timestamp()

dt_avg = np.mean(t[1:] - t[:-1])
print("Average:", 1.0 / dt_avg, "fps")
```

Of course, the advantage of the generator method is more apparent when you want to do more than what the *acquire_to_files()* does.

## 4.2 Asynchronous acquisition

### 4.2.1 Simple asynchronous acquisition

The video recording may be used as part of a larger experimental program. In particular, you may want to keep monitoring the experiment conditions while recording the video, and possibly save the experiment parameters next to the video frames. The simplest way to achieve that is to implement the monitoring task and the video recording task as asynchronous functions.

The very simple *acquire_to_files_async()* method is sufficient for very basic cases. The usage is strictly similar to the synchronous *acquire_to_files()* method described in the previous section. In fact, the synchronous method is only a wrapper around the asynchronous method.

The simple example of the previous section can be rewritten like this:

```
import asyncio
import numpy as np
from pymanip.video.avt import AVT_Camera as Camera

async def some_monitoring():

    # ... do some monitoring of the experiment conditions ...

async def video_recording():

    acquisition_name = "essai_1"
    nframes = 3000

    with Camera() as cam:
        cam.set_trigger_mode(True)
        count, dt = await cam.acquire_to_files_async(
            nframes,
            f"{acquisition_name:}/img",
            dryrun=False,
            file_format="png",
            compression_level=9,
            delay_save=True,
        )

    dt_avg = np.mean(t[1:] - t[:-1])
    print("Average:", 1.0 / dt_avg, "fps")

async def main():
    await asyncio.gather(some_monitoring(),
                         video_recording(),
                        )

asyncio.run(main())
```

### 4.2.2 Multi-camera acquisition

One application of this simple method is to extent to simultaneous acquisition on several cameras (possibly of different brands). To ensure simultaneous frame grabbing, it is necessary to use an external function generator for external triggering. In the following example, we use an Agilent 33220a function generator which we configure for a burst with a software trigger. In our case, we use USB to communicate with the generator. Once the two cameras are ready for frame grabbing, the software trigger is sent, and the frames from both cameras are acquired.

Some manual tickering may be necessary in real cases. For example, in our example we use two Firewire AVT cameras connected on the same FireWire board. So we must adjust the packet size.

To know when both cameras are ready to grab frames, we use the `initialising_cams` parameter. Each camera object removes itself from this set when it is ready to grab frames. All we need is therefore to implement a task which will send the software trigger to the function generator, once the set is empty.

Because the function generator is programmed for a burst of N peaks, the cameras will only be triggered N times. Therefore, it is a way to make sure that the N obtained frames were indeed simultaneous. If one camera skips one frame, the total number of frame will no longer be N.

The code is as follow:

```
import asyncio
from datetime import datetime
from pymanip.instruments import Agilent33220a
from pymanip.video.avt import AVT_Camera

basename = "multi"
fps = 10.0
nframes = 100

with Agilent33220a("USB0::2391::1031::MY44052515::INSTR") as gbf:

    gbf.configure_burst(fps, nframes)

    async def start_clock(cams):
        """This asynchronous function sends the software trigger to
           the gbf when all cams are ready.

           :param cams: initialising cams
           :type cams: set
           :return: timestamp of the time when the software trigger was sent
           :rtype: float
        """
        while len(cams) > 0:
            await asyncio.sleep(1e-3)
        gbf.trigger()
        return datetime.now().timestamp()

    with AVT_Camera(0) as cam0, \
         AVT_Camera(1) as cam1:

        cam0.set_trigger_mode(True)  # external trigger
        cam1.set_trigger_mode(True)
        cam0.set_exposure_time(10e-3)
        cam1.set_exposure_time(10e-3)
        cam0.camera.IIDCPacketSizeAuto = "Off"
        cam0.camera.IIDCPacketSize = 5720
        cam1.camera.IIDCPacketSizeAuto = "Off"
        cam1.camera.IIDCPacketSize = 8192 // 2

        initialing_cams = {cam0, cam1}

        task0 = cam0.acquire_to_files_async(
            nframes,
            basename + "/cam0",
            zerofill=4,
            file_format="png",
            delay_save=True,
            progressbar=True,
            initialising_cams=initialing_cams,
        )
        task1 = cam1.acquire_to_files_async(
```

(continues on next page)

(continued from previous page)

```
            nframes,
            basename + "/cam1",
            zerofill=4,
            file_format="png",
            delay_save=True,
            progressbar=False,
            initialising_cams=initialing_cams,
        )
        task2 = start_clock(initialing_cams)

        tasks = asyncio.gather(task0, task1, task2)
        loop = asyncio.get_event_loop()
        (countA, dtA), (countB, dtB), t0 = loop.run_until_complete(tasks)
```

Note that we use the `progressbar` parameter to avoid printing two progress bars. The `acquire_to_files_async()` methods are passed the number of expected frames. If one frame is skipped, a Timeout exception will be raised.

## 4.3 Advanced usage

In this section, we illustrate a more advanced usage from one of our own use case. We need simultaneous acquisition on two cameras. The framerate is too fast to wait for each frame to be saved before grabbing the next one. But we don't want to delay until the end of the acquisition (which might still be long) to start saving, because we don't want to loose all the data in case something bad happens, and we wish to be able to have a look at the picture before the acquisition ends.

So, in this example, we implement simple queues in which the frames are stored, and there is a fourth task which gets the frames from this queue and saves them, (at a lower rate than the acquisition rate). When the acquisition is stopped, this last task finishes the saving. In addition, we want to save acquisition parameters with an *AsyncSession* object. To summarize the four tasks:

- task0: acquisition on cam 0
- task1: acquisition on cam 1
- task2: software trigger when cams are ready
- task3: background saving of images

The first 3 tasks are similar to those in the previous section. We use Python standard library `SimpleQueue` class to implement the frame queue.

We comment the various parts of this script in the following subsections.

### 4.3.1 Preambule

First the import statements, and definition of some global parameters (video parameter, as well as names for output files).

```
from queue import SimpleQueue
import asyncio
import os
import cv2
from datetime import datetime
```

(continues on next page)

(continued from previous page)

```
from pymanip.instruments import Agilent33220a
from pymanip.video.avt import AVT_Camera
from pymanip.asyncsession import AsyncSession
from progressbar import ProgressBar

# User inputs
compression_level = 3
exposure_time = 10e-3
cam_type = "avt"
fps = 2
num_imgs = 10

# Paths to save data
current_date = datetime.today().strftime("%Y%m%d")
current_dir = os.path.dirname(os.path.realpath(__file__))
saving_dir_date = f"{current_dir}\\data\\{current_date}\\"
if not os.path.isdir(saving_dir_date):
    os.makedirs(saving_dir_date)
num_dir = len(os.listdir(saving_dir_date))
saving_dir_run = f"{saving_dir_date}run{num_dir+1:02.0f}\\"
if not os.path.isdir(saving_dir_run):
    os.makedirs(saving_dir_run)
basename = f"{saving_dir_run}session"
```

### 4.3.2 Acquisition task (task 0 and task 1)

This task is responsible for grabbing frames for the camera, and putting it in the queue. Note that we must *copy* the image because the numpy array yielded by the *acquisition_async()* generator uses shared memory (and would no longer hold this particular frame on subsequent iterations of the generator). The queues are created in the main function, and are called `im_buffer0` for camera 0, and `im_buffer1` for camera 1.

In addition, we want to be able to abort the script. We use the mechanisms defined in the *pymanip.asyncsession.AsyncSession* class which set ups signal handling for interrupt signal. It basically defines a `running` attribute that is set to `False` when the program should stop. The acquisition task must check this variable to cleanup stop grabbing the frames if the user has sent the interrupt signal.

```
async def acquire_images(sesn, cam, num_cam, initialing_cams):
    global num_imgs

    kk = 0
    bar = ProgressBar(min_value=0, max_value=num_imgs, initial_value=kk)
    gen = cam.acquisition_async(num_imgs, initialising_cams=initialing_cams)

    async for im in gen:
        if num_cam == 0:
            sesn.im_buffer0.put(im.copy())
        elif num_cam == 1:
            sesn.im_buffer1.put(im.copy())

        kk += 1
```

(continues on next page)

(continued from previous page)

```
        bar.update(kk)
        if not sesn.running:
            num_imgs = kk
            success = await gen.asend(True)
            if not success:
                print("Unable to stop camera acquisition")
            break
    bar.finish()

    print(f"Camera acquisition stopped ({kk:d} images recorded).")
    sesn.running = False
```

### 4.3.3 Software trigger task (task 2)

This task monitor the set of initialising cams, which gets empty when all the cameras are ready to grab frames. Then, it triggers the generator function.

```
async def start_clock(cams):
    # Start clocks once all camera are done initializing
    while len(cams) > 0:
        await asyncio.sleep(1e-3)
    gbf.trigger()
    return datetime.now().timestamp()
```

### 4.3.4 Background saving of images (task 3)

This task looks for images in the `im_buffer0` and `im_buffer1` queues, as long as the acquisition is still running or that the queues are not empty. The images are saved using OpenCV `imwrite()` function that we run in an executor (i.e. in a separate thread), so as not to block the acquisition tasks.

```
async def save_images(sesn):
    params = (cv2.IMWRITE_PNG_COMPRESSION, compression_level)
    loop = asyncio.get_event_loop()
    i = 0
    bar = None

    while sesn.running or not sesn.im_buffer0.empty() or not sesn.im_buffer1.empty():
        if sesn.im_buffer0.empty() and sesn.im_buffer1.empty():
            await asyncio.sleep(1.0)
        else:
            if not im_buffer0.empty():
                im0 = sesn.im_buffer0.get()
                filename0 = f"{saving_dir_run}\\cam0_{i:04d}.png"
                await loop.run_in_executor(None, cv2.imwrite, filename0, im0, params)
                i += 1

            if not im_buffer1.empty():
                im1 = sesn.im_buffer1.get()
                filename1 = f"{saving_dir_run}\\cam1_{i:04d}.png"
                await loop.run_in_executor(None, cv2.imwrite, filename1, im1, params)
```

(continues on next page)

(continued from previous page)

```
        if not sesn.running:
            if bar is None:
                print("Saving is terminating...")
                bar = ProgressBar(
                    min_value=0, max_value=2 * num_imgs, initial_value=2 * i
                )
            else:
                bar.update(2 * i)
    if bar is not None:
        bar.finish()
    print(f"{2*i:d} images saved.")
```

One important point is that this task is the only task which access disk storage. The acquisition tasks work solely on memory, so they are not slowed down by the saving task.

### 4.3.5 Main function and setup

The main function sets up the function generator and the cameras, and start the tasks. It must also create the queues.

```
async def main():

    with AsyncSession(basename) as sesn, \
         Agilent33220a("USB0::2391::1031::MY44052515::INSTR") as sesn.gbf:

        # Configure function generator
        sesn.gbf.configure_burst(fps, num_imgs)
        sesn.save_parameter(fps=fps, num_imgs=num_imgs)

        # Prepare buffer queues
        sesn.im_buffer0 = SimpleQueue()
        sesn.im_buffer1 = SimpleQueue()

        # Prepare camera and start tasks
        with AVT_Camera(0) as cam0, \
             AVT_Camera(1) as cam1:

            # External trigger and camera properties
            cam0.set_trigger_mode(True)
            cam1.set_trigger_mode(True)

            cam0.set_exposure_time(exposure_time)
            cam1.set_exposure_time(exposure_time)
            sesn.save_parameter(exposure_time=exposure_time)

            cam0.camera.IIDCPacketSizeAuto = "Off"
            cam0.camera.IIDCPacketSize = 5720
            cam1.camera.IIDCPacketSizeAuto = "Off"
            cam1.camera.IIDCPacketSize = 8192 // 2

            # Set up tasks
```

(continues on next page)

(continued from previous page)

```
            initialing_cams = {cam0, cam1}
            task0 = acquire_images(sesn, cam0, 0, initialing_cams)
            task1 = acquire_images(sesn, cam1, 1, initialing_cams)
            task2 = start_clock(initialing_cams)
            task3 = save_images(sesn)

            # We use AsyncSession monitor co-routine which set ups the signal
            # handling. We don't need remote access, so server_port=None.
            # Alternative:
            # await asyncio.gather(task0, task1, task2, task3)
            await sesn.monitor(task0, task1, task2, task3,
                               server_port=None)

asyncio.run(main())
```

## 4.4 Asynchronous acquisition session (pymanip.video.session)

This module provides a high-level class, *VideoSession*, to be used in acquisition scripts. Users should subclass *VideoSession*, and add two methods: `prepare_camera(self, cam)` in which additionnal camera setup may be done, and `process_image(self, img)` in which possible image post-processing can be done. The `prepare_camera` method, if defined, is called before starting camera acquisition. The `process_image` method, if defined, is called before saving the image.

The implementation allows simultaneous acquisition of several cameras, trigged by a function generator. Concurrent tasks are set up to grab images from the cameras to RAM memory, while a background task saves the images to the disk. The cameras, and the trigger, are released as soon as acquisition is finished, even if the images are still being written to the disk, which allows several scripts to be executed concurrently (if the host computer has enough RAM). The `trigger_gbf` object must implement `configure_square()` and `configure_burst()` methods to configure square waves and bursts, as well as `trigger()` method for software trigger. An exemple is `fluidlab.instruments.funcgen.agilent_33220a.Agilent33220a`.

A context manager must be used to ensure proper saving of the metadata to the database.

**Example**

```
import numpy as np
import cv2

from pymanip.video.session import VideoSession
from pymanip.video.ximea import Ximea_Camera
from pymanip.instruments import Agilent33220a


class TLCSession(VideoSession):

    def prepare_camera(self, cam):
        # Set up camera (keep values as custom instance attributes)
        self.exposure_time = 50e-3
        self.decimation_factor = 2

        cam.set_exposure_time(self.exposure_time)
```

(continues on next page)

(continued from previous page)

```
        cam.set_auto_white_balance(False)
        cam.set_limit_bandwidth(False)
        cam.set_vertical_skipping(self.decimation_factor)
        cam.set_roi(1298, 1833, 2961, 2304)

        # Save some metadata to the AsyncSession underlying database
        self.save_parameter(
            exposure_time=self.exposure_time,
            decimation_factor=self.decimation_factor,
        )

    def process_image(self, img):
        # On decimate manuellement dans la direction horizontale
        img = img[:, ::self.decimation_factor, :]

        # On redivise encore tout par 2
        # image_size = (img.shape[0]//2, img.shape[1]//2)
        # img = cv2.resize(img, image_size)

        # Correction (fixe) de la balance des blancs
        kR = 1.75
        kG = 1.0
        kB = 2.25
        b, g, r = cv2.split(img)
        img = np.array(
            cv2.merge([kB*b, kG*g, kR*r]),
            dtype=img.dtype,
        )

        # Rotation de 180°
        img = cv2.rotate(img, cv2.ROTATE_180)

        return img


with TLCSession(
    Ximea_Camera(),
    trigger_gbf=Agilent33220a("USB0::0x0957::0x0407::SG43000299::INSTR"),
    framerate=24,
    nframes=1000,
    output_format="png",
    ) as sesn:

    # ROI helper (remove cam.set_roi in prepare_camera for correct usage)
    # sesn.roi_finder()

    # Single picture test
    # sesn.show_one_image()

    # Live preview
    # ts, count = sesn.live()
```

(continues on next page)

(continued from previous page)

```
    # Run actual acquisition
    ts, count = sesn.run(additionnal_trig=1)
```

class pymanip.video.session.**VideoSession**(*camera_or_camera_list*, *trigger_gbf*, *framerate*, *nframes*, *output_format*, *output_format_params=None*, *output_path=None*, *exist_ok=False*, *timeout=None*, *burst_mode=True*)

Bases: *AsyncSession*

This class represents a video acquisition session.

**Parameters**

- **camera_or_camera_list** (*Camera* or list of *Camera*) – Camera(s) to be acquired
- **trigger_gbf** (Driver) – function generator to be used as trigger
- **framerate** (*float*) – desired framerate
- **nframes** (*int*) – desired number of frames
- **output_format** (*str*) – desired output image format, “bmp”, “png”, tif”, or video format “mp4”
- **output_format_params** (*list*) – additionnal params to be passed to `cv2.imwrite()`
- **exist_ok** (*bool*) – allows to override existing output folder

**async _acquire_images**(*cam_no*, *live=False*)

Private instance method: image acquisition task. This task asynchronously iterates over the given camera frames, and puts the obtained images in a simple FIFO queue.

**Parameters**

- **cam_no** (*int*) – camera index
- **live** (*bool, optional*) – if True, images are converted to numpy array even if self.output_format = ‘dng’

**_convert_for_ffmpeg**(*cam_no*, *img*, *fmin*, *fmax*, *gain*)

Private instance method: image conversion for ffmpeg process. This method prepares the input image to bytes to be sent to the ffmpeg pipe.

**Parameters**

- **cam_no** (*int*) – camera index
- **img** (numpy.ndarray) – image to process
- **fmin** (*int*) – minimum level
- **fmax** (*int*) – maximum level
- **gain** (*float*) – gain

**async _fast_acquisition_to_ram**(*cam_no*, *total_timeout_s*)

Private instance method: fast acquisition to ram task

**async _live_preview**(*unprocessed=False*)

Private instance method: live preview task. This task checks the FIFO queue, drains it and shows the last frame of each camera using cv2. If more than one frame is in the queues, the older frames are dropped.

**Parameters**
**unprocessed** (*bool*) – do not call `process_image()` method.

**async _save_images**(*keep_in_RAM=False*, *unprocessed=False*, *no_save=False*)

Private instance method: image saving task. This task checks the image FIFO queue. If an image is available, it is taken out of the queue and saved to the disk.

**Parameters**

- **keep_in_RAM** (*bool*) – if set, images are not saved and kept in a list.
- **unprocessed** (*bool*) – if set, the `process_image` method is not called.
- **no_save** (*bool*) – do not actually save (dry run), for testing purposes

**async _save_images_finished**()

Awaits all images in queue have been saved.

**async _save_video**(*cam_no*, *gain=1.0*, *unprocessed=False*)

Private instance method: video saving task. This task waits for images in the FIFO queue, and sends them to ffmpeg via a pipe.

**Parameters**

- **cam_no** (*int*) – camera index
- **gain** (*float*) – gain
- **unprocessed** (*bool*) – if set, `process_image()` method is not called

**async _start_clock**()

Private instance method: clock starting task. This task waits for all the cameras to be ready for trigger, and then sends a software trig to the function generator.

**get_one_image**(*additionnal_trig=0*, *unprocessed=False*, *unpack_solo_cam=True*)

Get one image from the camera(s).

**Parameters**

- **additionnal_trig** (*int*) – additionnal number of pulses sent to the camera
- **unprocessed** (*bool*) – do not call `process_image()` method.
- **unpack_solo_cam** (*bool*) – if set, keep list on return, even if there is only one camera

**Returns**
image(s) from the camera(s)

**Return type**
`numpy.ndarray` or list of `numpy.ndarray` (if multiple cameras, or `unpack_solo_cam=False`).

**live**()

Starts live preview.

**async main**(*keep_in_RAM=False*, *additionnal_trig=0*, *live=False*, *unprocessed=False*, *delay_save=False*, *no_save=False*)

Main entry point for acquisition tasks. This asynchronous task can be called with `asyncio.run()`, or combined with other user-defined tasks.

**Parameters**

- **keep_in_RAM** (*bool*) – do not save to disk, but keep images in a list

- **additionnal_trig** (*int*) – additionnal number of pulses sent to the camera
- **live** (*bool*) – toggle live preview
- **unprocessed** (*bool*) – do not call `process_image()` method.

**Returns**
camera_timestamps, camera_counter

**Return type**
`numpy.ndarray`, `numpy.ndarray`

**roi_finder**(*additionnal_trig=0*)

Helper to determine the ROI. This method grabs one unprocessed image from the camera(s), and allows interactive selection of the region of interest. Attention: it is assumed that the `prepare_camera()` method did not already set the ROI.

**Parameters**
**additionnal_trig** (*int*) – additionnal number of pulses sent to the camera

**Returns**
region of interest coordinates, X0, Y0, X1, Y1

**Return type**
list of floats

**run**(*additionnal_trig=0*, *delay_save=False*, *no_save=False*)

Run the acquisition.

**Parameters**
**additionnal_trig** (*int*) – additionnal number of pulses sent to the camera

**Returns**
camera_timestamps, camera_counter

**Return type**
`numpy.ndarray`, `numpy.ndarray`

**show_one_image**(*additionnal_trig=0*)

Get one image from the camera(s), and plot them with matplotlib.

**Parameters**
**additionnal_trig** (*int*) – additionnal number of pulses sent to the camera

## 4.5 Implementation

### 4.5.1 Video acquisition module (`pymanip.video`)

This module defines the *Camera* abstract base class, which implements common methods such as the live video preview, and higher level simple methods to quickly set up a video recording. It also defines common useful functions, and a simple extension of Numpy arrays to hold metadata (such as frame timestamp).

**class** pymanip.video.**Camera**

This class is the abstract base class for all other concrete camera classes. The concrete sub-classes *must* implement the following methods:

- *acquisition_oneshot()* method
- *acquisition()* and *acquisition_async()* generator methods

- `resolution`, `name` and `bitdepth` properties

The concrete sub-classes will also probably have to override the constructor method, and the enter/exit context manager method, as well as common property getters and setters:

- `set_exposure_time()`
- `set_trigger_mode()`
- `set_roi()`
- `set_frame_rate()`

It may also define specialized getters for the camera which support them:

- `set_adc_operating_mode()`: ADC operating mode
- `set_pixel_rate()`: pixel rate sensor readout (in Hz)
- `set_delay_exposuretime()`

**acquire_signalHandler**(**args*, ***kwargs*)

This method sends a stop signal to the *acquire_to_files_async()* method.

**acquire_to_files**(**args*, ***kwargs*)

This method starts the camera, acquires images and saves them to the disk. It is a simple wrapper around the *pymanip.video.Camera.acquire_to_files_async()* asynchronous method. The parameters are identical.

**async acquire_to_files_async**(*num*, *basename*, *zerofill=4*, *dryrun=False*, *file_format='png'*, *compression=None*, *compression_level=3*, *verbose=True*, *delay_save=False*, *progressbar=True*, *initialising_cams=None*, ***kwargs*)

This asynchronous method starts the camera, acquires **num** images and saves them to the disk. It is a simple quick way to perform camera acquisition (one-liner in the user code).

**Parameters**

- **num** (*int*) – number of frames to acquire
- **basename** (*str*) – basename for image filenames to be saved on disk
- **zerofill** (*int, optional*) – number of digits for the framenumber for image filename, defaults to 4
- **dryrun** (*bool, optional*) – do the acquisition, but saves nothing (testing purposes), defaults to False
- **file_format** (*str, optional*) – format for the image files, defaults to "png". Possible values are "raw", "npy", "npy.gz", "hdf5", "png" or any other extension supported by OpenCV imwrite.
- **compression** (*str, optional*) – compression option for HDF5 format ("gzip", "lzf"), defaults to None.
- **compression_level** (*int, optional*) – png compression level for PNG format, defaults to 3.
- **verbose** (*bool, optional*) – prints information message, defaults to True.
- **delay_save** (*bool, optional*) – records all the frame in RAM, and saves at the end. This is useful for fast framerates when saving time is too slow. Defaults to False.
- **progressbar** (*bool, optional*) – use `progressbar` module to show a progress bar. Defaults to True.

- **initialising_cams** (*set, optional*) – None, or set of camera objects. This camera object will remove itself from this set, once it is ready to grab frames. Useful in the case of multi camera acquisitions, to determine when all cameras are ready to grab frames. Defaults to None.

**Returns**
image_counter, frame_datetime

**Return type**
list, list

The details of the file format are given in this table:

| file_format | description |
|---|---|
| raw | native 16 bits integers, i.e. li16 (little-endian) on Intel CPUs |
| npy | numpy npy file (warning: depends on pickle format) |
| npy.gz | gzip compressed numpy file |
| hdf5 | hdf5, with optional compression |
| png, jpg, tif | image format with opencv imwrite with optional compression level for PNG |

Typical usage of the function for one camera:

```
async def main():

    with Camera() as cam:
        counts, times = await cam.acquire_to_files_async(num=20, basename='img-
↪')

asyncio.run(main())
```

**acquisition**(*num=inf*, *timeout=1000*, *raw=False*, *initialising_cams=None*, *raise_on_timeout=True*)

This generator method is the main method that sub-classes must implement, along with the asynchronous variant. It is used by all the other higher-level methods, and can also be used directly in user code.

**Parameters**

- **num** (*int, or float("inf"), optional*) – number of frames to acquire, defaults to float("inf").
- **timout** – timeout for frame acquisition (in milliseconds)
- **raw** (*bool, optional*) – if True, returns bytes from the camera without any conversion. Defaults to False.
- **initialising_cams** (*set, optional*) – None, or set of camera objects. This camera object will remove itself from this set, once it is ready to grab frames. Useful in the case of multi camera acquisitions, to determine when all cameras are ready to grab frames. Defaults to None.
- **raise_on_timeout** (*bool, optional*) – boolean indicating whether to actually raise an exception when timeout occurs

It starts the camera, yields `num` images, and closes the camera. It can be aborted when sent a true-truth value object. It then cleanly stops the camera and finally yields True as a confirmation that the stop_signal has been caught before returning. Sub-classes must therefore reads the possible stop_signal when yielding the frame, and act accordingly.

The *MetadataArray* objects yielded by this generator use a shared memory buffer which may be overriden for the next frame, and which is no longer defined when the generator object is cleaned up. The users are responsible for copying the array, if they want a persistant copy.

User-level code will use the generator in this manner:

```
gen = cam.acquire()
for frame in gen:

    # .. do something with frame ..

    if I_want_to_stop:
        clean = gen.send(True)
        if not clean:
            print('Warning generator not cleaned')
        # no need to break here because the gen will be automatically exhausted
```

**async acquisition_async**(*num=inf*, *timeout=1000*, *raw=False*, *initialising_cams=None*, *raise_on_timeout=True*)

This asynchronous generator method is similar to the *acquisition()* generator method, except asynchronous. So much so, that in the general case, the latter can be defined simply by yielding from this asynchronous generator (so that the code is written once for both use cases), i.e.

```
from pymanip.asynctools import synchronize_generator

def acquisition(
    self,
    num=np.inf,
    timeout=1000,
    raw=False,
    initialising_cams=None,
    raise_on_timeout=True,
):
    yield from synchronize_generator(
        self.acquisition_async,
        num,
        timeout,
        raw,
        initialising_cams,
        raise_on_timeout,
    )
```

It starts the camera, yields `num` images, and closes the camera. It can stop yielding images by sending the generator object a true-truth value object. It then cleanly stops the camera and finally yields True as a confirmation that the stop_signal has been caught before returning. Sub-classes must therefore reads the possible stop_signal when yielding the frame, and act accordingly.

The *MetadataArray* objects yielded by this generator use a shared memory buffer which may be overriden for the next frame, and which is no longer defined when the generator object is cleaned up. The users are responsible for copying the array, if they want a persistant copy.

The user API is similar, except with asynchronous calls, i.e.

```
gen = cam.acquire_async()
async for frame in gen:
```

(continues on next page)

(continued from previous page)

```
    # .. do something with frame ..

    if I_want_to_stop:
        clean = await gen.asend(True)
        if not clean:
            print('Warning generator not cleaned')
        # no need to break here because the gen will be automatically exhausted
```

**acquisition_oneshot**()

This method must be implemented in the sub-classes. It starts the camera, grab one frame, stops the camera, and returns the frame. It is useful for testing purposes, or in cases where only one frame is desired between very long time delays. It takes no input parameters. Returns an "autonomous" array (the buffer is independant of the camera object).

**Returns**
frame

**Return type**
*MetadataArray*

**display_crosshair**()

This method adds a centered crosshair for self-reflection to the live-preview window (qt backend only)

**preview**(*backend='cv'*, *slice_=None*, *zoom=0.5*, *rotate=0*)

This methods starts and synchronously runs the live-preview GUI.

**Parameters**

- **backend** (*str*) – GUI library to use. Possible values: "cv" for OpenCV GUI, "qt" for PyQtGraph GUI.
- **slice** (*Iterable[int], optional*) – coordinate of the region of interest to show, defaults to None
- **zoom** (*float, optional*) – zoom factor, defaults to 0.5
- **rotate** (*float, optional*) – image rotation angle, defaults to 0

**async preview_async_cv**(*slice_*, *zoom*, *name*, *rotate=0*)

This method starts and asynchronously runs the live-preview with OpenCV GUI. The params are identical to the *preview()* method.

**preview_cv**(*slice_*, *zoom*, *rotate=0*)

This method starts and synchronously runs the live-preview with OpenCV GUI. It is a wrapper around the *pymanip.video.Camera.preview_async_cv()* method. The params are identical to the *preview()* method.

**preview_exitHandler**()

This method sends a stop signal to the camera acquisition generator of the live-preview GUI.

**preview_qt**(*slice*, *zoom*, *app=None*, *rotate=0*)

This methods starts and synchronously runs the live-preview with Qt GUI. The params are identical to the *preview()* method.

**class** pymanip.video.**CameraTimeout**

This class defines a CameraTimeout exception.

class pymanip.video.**MetadataArray**(*input_array*, *metadata=None*)

Bases: ndarray

This class extends Numpy array to allow for an additionnal metadata attribute.

**metadata**

dictionnary attribute containing user-defined key-value pairs

pymanip.video.**save_image**(*im*, *ii*, *basename*, *zerofill*, *file_format*, *compression*, *compression_level*, *color_order=None*)

This function is a simple general function to save an input image from the camera to disk.

**Parameters**

- **im** (*MetadataArray*) – input image
- **ii** (*int*) – frame number
- **basename** (*str*) – file basename
- **zerofill** (*int*) – number of digits for the frame number
- **file_format** (*str*) – image file format on disk. Possible values are: "raw", "npy", "npy.gz", "hdf5", "png", or a file extension that OpenCV imwrite supports
- **compression** (*str*) – the compression argument "gzip" or "lzf" to pass to h5py.create_dataset() if file_format is "hdf5"
- **compression_level** (*int*) – the png compression level passed to opencv for the "png" file format

## 4.6 Concrete implementation for Andor camera

### 4.6.1 Andor module (pymanip.video.andor)

This module is a shortcut for the *pymanip.video.andor.camera.Andor_Camera* class.

### 4.6.2 Andor Camera module (pymanip.video.andor.camera)

This module implement the *Andor_Camera* class, as a subclass of the *pymanip.video.Camera* base class. It uses the third-party pyAndorNeo module.

class pymanip.video.andor.camera.**Andor_Camera**(*camNum=0*)

Concrete *pymanip.video.Camera* class for Andor camera.

**Parameters**

**camNum** (*int, optional*) – camera number, defaults to 0.

**async acquisition_async**(*num=inf*, *timeout=None*, *raw=False*, *initialising_cams=None*, *raise_on_timeout=True*)

Concrete implementation of *pymanip.video.Camera.acquisition_async()* for the Andor camera.

**acquisition_oneshot**(*timeout=1.0*)

Concrete implementation of *pymanip.video.Camera.acquisition_oneshot()* for the Andor camera.

**close**()

This method closes the Camera handle.

**get_exposure_time**()

This method returns the exposure time in seconds

**set_exposure_time**(*seconds*)

This method sets the camera exposure time

**Parameters**
**seconds** (*float*) – exposure in seconds

### 4.6.3 Andor reader module (pymanip.video.andor.reader)

This module implements simple pure-python reader for Andor DAT and SIF files.

**class** pymanip.video.andor.reader.**AndorAcquisitionReader**(*acquisition_folder*)

This class is a simple pure-python reader for Andor DAT spool files in a directory.

**Parameters**
**acquisition_folder** (*str*) – the folder in which to read the DAT spool files

**images**()

This generator method yields the images found in the folder.

**class** pymanip.video.andor.reader.**SIFFile**(*filename*)

This class implements a pure-python reader for Andor SIF files.

**Parameters**
**filename** (*str*) – the SIF filename

**close**()

This method closes the file.

**images**()

This generator yields all the frames in the file.

**open**()

This method opens the file.

**read_frame**()

The methods reads the next frame in file.

**Returns**
frame

**Return type**
numpy.ndarray

**read_header**()

This method reads the header in the SIF file.

**read_nth_frame**(*n*)

This method reads and returns the nth frame in the file.

**Parameters**
**n** (*int*) – frame number to read

**Returns**
frame

**Return type**
numpy.ndarray

## 4.7 Concrete implementation for AVT camera

### 4.7.1 AVT Camera module (pymanip.video.avt)

This module implements the *pymanip.video.avt.AVT_Camera* class using the third-party pymba module.

**class** pymanip.video.avt.**AVT_Camera**(*cam_num=0*, *pixelFormat='Mono16'*)

Bases: *Camera*

Concrete *pymanip.video.Camera* class for AVT camera.

**Parameters**

- **cam_num** (*int, optional*) – camera number, defaults to 0
- **pixelFormat** (*str, optional*) – pixel format to use, e.g. "Mono8", "Mono16". Defaults to "Mono16"

**acquisition**(*num=inf*, *timeout=1000*, *raw=False*, *framerate=None*, *external_trigger=False*, *initialising_cams=None*, *raise_on_timeout=True*)

Concrete implementation of *pymanip.video.Camera.acquisition()* for the AVT camera.

**async acquisition_async**(*num=inf*, *timeout=1000*, *raw=False*, *framerate=None*, *external_trigger=False*, *initialising_cams=None*, *raise_on_timeout=True*)

Concrete implementation of *pymanip.video.Camera.acquisition_async()* for the AVT camera.

**acquisition_oneshot**()

Concrete implementation of *pymanip.video.Camera.acquisition_oneshot()* for the AVT camera.

**camera_feature_info**(*featureName*)

This method queries the camera for the specified feature.

**Parameters**
**featureName** (*str*) – one of the features returned by *camera_features()*

**Returns**
values associated with the specified feature

**Return type**
dict

**camera_features**()

This methods returns the list of possible camera features.

**Returns**
camera feature

**Return type**
list

**close**()

This method closes the connection to the camera.

**classmethod get_camera_list**()

This methods returns the list of camera numbers connected to the computer.

**Returns**
camera ids

**Return type**
list

**set_exposure_time**(*seconds*)

This method sets the exposure time for the camera.

**Parameters**
**seconds** (*float*) – exposure in seconds. Possible values range from 33.0 µs to 67108895.0 µs.

**set_roi**(*roiX0=0*, *roiY0=0*, *roiX1=0*, *roiY1=0*)

This mothods sets the position of the upper left corner (X0, Y0) and the lower right (X1, Y1) corner of the ROI (region of interest) in pixels.

**set_trigger_mode**(*mode=False*)

This method sets the trigger mode for the camera. For external trigger, the method also sets IIDCPacketSizeAuto to 'Maximize'.

**Parameters**
**mode** (*bool*) – True if external trigger. False if internal trigger.

## 4.8 Concrete implementation for PCO camera

### 4.8.1 PCO module (pymanip.video.pco)

This module is a shortcut for the *pymanip.video.pco.camera.PCO_Camera* class. It also defines utility functions for PCO camera.

pymanip.video.pco.**PCO_read_binary_file**(*f*)

This functions reads PCO binary image file.

pymanip.video.pco.**print_available_pco_cameras**()

This functions queries the Pixelfly library for available cameras, and prints the result.

### 4.8.2 PixelFly library bindings (pymanip.video.pco.pixelfly)

This module implements bindings to the PCO PixelFly library using ctypes. Please not that these bindings are not official, and that not all PixelFly functions are wrapped. Please refer to the official PCO PixelFly documentation for accurate description of the functions.

**class** pymanip.video.pco.pixelfly.**PCO_Image**

This class is a binding to the PCO_Image C Structure.

pymanip.video.pco.pixelfly.**PCO_manage_error**(*ret_code*)

This function raises an error exception or a runtime warning if ret_code is non-zero.

**Parameters**
**ret_code** (*int*) – PCO library function return code

pymanip.video.pco.pixelfly.**bcd_byte_to_str**(*input_*)

This function converts a one-byte bcd value into two digit string.

**Parameters**
**input** (*int*) – bcd value to convert

**Returns**
converted bcd value

**Return type**
str

pymanip.video.pco.pixelfly.**bcd_to_int**(*input_*, *endianess='little'*)

This function converts decimal-coded value (bcd) into int.

**Parameters**
**input** (*byte or bytearray*) – input bcd value

**Returns**
integer value

**Return type**
int

Decimal-encoded value format is given in this table:

| Decimal digit | Bits |
|---|---|
| 0 | 0000 |
| 1 | 0001 |
| 2 | 0010 |
| 3 | 0011 |
| 4 | 0100 |
| 5 | 0101 |
| 6 | 0110 |
| 7 | 0111 |
| 8 | 1000 |
| 9 | 1001 |

pymanip.video.pco.pixelfly.**PCO_OpenCamera**()

This function opens a camera device and attach it to a handle, which will be returned by the parameter ph. This function scans for the next available camera. If you want to access a distinct camera please use PCO_OpenCameraEx. Due to historical reasons the wCamNum parameter is a don't care.

pymanip.video.pco.pixelfly.**PCO_OpenCameraEx**(*interface_type*, *camera_number*)

This function opens a distinct camera, e.g. a camera which is connected to a specific interface port.

**Parameters**

- **interface_type** (*int*) – Interface type to look for the camera
- **camera_number** (*int*) – ID of the camera

The interface_type values are given in this table:

| Interface | interface_type |
|---|---|
| FireWire | 1 |
| Camera Link Matrox | 2 |
| Camera Link Silicon Software mE III | 3 |
| Camera Link National Instruments | 4 |
| GigE | 5 |
| USB 2.0 | 6 |
| Camera Link Silicon Software mE IV | 7 |
| USB 3.0 | 8 |
| Reserved for WLAN | 9 |
| Camera Link serial port only | 10 |
| Camera Link HS | 11 |
| all | 0xFFFF |

pymanip.video.pco.pixelfly.**PCO_CloseCamera**(*handle*)

This function closes a camera device.

**Parameters**
**handle** (*HANDLE*) – handle of the camera

pymanip.video.pco.pixelfly.**PCO_GetInfoString**(*handle*)

This function reads information about the camera, e.g. firmware versions.

**Parameters**
**handle** (*HANDLE*) – camera handle

pymanip.video.pco.pixelfly.**PCO_GetROI**(*handle: int*) → Tuple[int, int, int, int]

This function returns the current ROI (region of interest) setting in pixels. (X0,Y0) is the upper left corner and (X1,Y1) the lower right one.

**Parameters**
**handle** (*HANDLE*) – handle of the camera

**Returns**
X0, Y0, X1, Y1

**Return type**
int, int, int, int

pymanip.video.pco.pixelfly.**PCO_SetROI**(*handle: int*, *RoiX0: c_ushort*, *RoiY0: c_ushort*, *RoiX1: c_ushort*, *RoiY1: c_ushort*)

This function does set a ROI (region of interest) area on the sensor of the camera.

**Parameters**

- **handle** (*HANDLE*) – handle of the camera
- **RoiX0** (*int*) – left border
- **RoiY0** (*int*) – upper border
- **RoiX1** (*int*) – right border
- **RoiY1** (*int*) – lower border

pymanip.video.pco.pixelfly.**PCO_GetFrameRate**(*handle*)

This function returns the current frame rate and exposure time settings of the camera.Returned values are only valid if last timing command was PCO_SetFrameRate.

pymanip.video.pco.pixelfly.**PCO_SetFrameRate**(*handle: int*, *FrameRateMode: c_ushort*, *FrameRate: c_ulong*, *FrameRateExposure: c_ulong*)

This function sets Frame rate (mHz) and exposure time (ns) Frame rate status gives the limiting factors if the condition are not met.

pymanip.video.pco.pixelfly.**PCO_GetCameraName**(*handle*)

This function retrieves the name of the camera.

pymanip.video.pco.pixelfly.**PCO_GetGeneral**(*handle*)

This function requests all info contained in the following descriptions, especially:

- camera type, hardware/firmware version, serial number, etc.
- Request the current camera and power supply temperatures

pymanip.video.pco.pixelfly.**PCO_GetSensorStruct**(*handle*)

Get the complete set of the sensor functions settings

pymanip.video.pco.pixelfly.**PCO_GetCameraDescription**(*handle*)

Sensor and camera specific description is queried. In the returned PCO_Description structure margins for all sensor related settings and bitfields for available options of the camera are given.

pymanip.video.pco.pixelfly.**PCO_GetCameraHealthStatus**(*handle*)

This function retrieves information about the current camera status.

pymanip.video.pco.pixelfly.**PCO_GetRecordingStruct**(*handle*)

Get the complete set of the recording function settings. Please fill in all wSize parameters, even in embedded structures.

pymanip.video.pco.pixelfly.**PCO_GetSizes**(*handle: int*) → Tuple[int, int, int, int]

This function returns the current armed image size of the camera.

pymanip.video.pco.pixelfly.**PCO_AllocateBuffer**(*handle: int*, *bufNr: int*, *size: int*, *bufPtr: LP_c_ushort*, *hEvent: c_void_p = 0*) → Tuple[int, LP_c_ushort, int]

This function sets up a buffer context to receive the transferred images. A buffer index is returned, which must be used for the image transfer functions. There is a maximum of 16 buffers per camera.

Attention: This function cannot be used, if the connection to the camera is established through the serial connection of a Camera Link grabber. In this case, the SDK of the grabber must be used to do any buffer management.

pymanip.video.pco.pixelfly.**PCO_FreeBuffer**(*handle*, *bufNr*)

This function frees a previously allocated buffer context with a given index. If internal memory was allocated for this buffer context, it will be freed. If an internal event handle was created, it will be closed.

pymanip.video.pco.pixelfly.**PCO_GetBufferStatus**(*handle*, *sBufNr*)

This function queries the status of the buffer context with the given index:

- **StatusDll describes the state of the buffer context:**

| StatusDll | description |
|---|---|
| 0x80000000 | Buffer is allocated |
| 0x40000000 | Buffer event created inside the SDK DLL |
| 0x20000000 | Buffer is allocated externally |
| 0x00008000 | Buffer event is set |

- **StatusDrv describes the state of the last image transfer into this buffer.**
  - PCO_NOERROR: Image transfer succeeded

  - others: See error codes

pymanip.video.pco.pixelfly.**PCO_ArmCamera**(*handle*)

Arms, i.e. prepares the camera for a consecutive set recording status = [run] command. All configurations and settings made up to this moment are accepted and the internal settings of the camera is prepared. Thus the camera is able to start immediately when the set recording status = [run] command is performed.

pymanip.video.pco.pixelfly.**PCO_GetRecordingState**(*handle*)

Returns the current Recording state of the camera: - 0x0000: camera is stopped, recording state [stop]

- 0x0001: camera is running, recording state [run]

pymanip.video.pco.pixelfly.**PCO_SetRecordingState**(*handle*, *state*)

Sets the current recording status and waits till the status is valid. If the state can't be set the function will return an error.

---

**Note:**

- it is necessary to arm camera before every set recording status command in order to ensure that all settings are accepted correctly.
- During the recording session, it is possible to change the timing by calling *PCO_SetDelayExposureTime()*.

---

pymanip.video.pco.pixelfly.**PCO_GetBitAlignment**(*handle*)

This function returns the current bit alignment of the transferred image data. The data can be either MSB (Big Endian) or LSB (Little Endian) aligned. Returns: - 0x0000 [MSB]

- 0x0001 [LSB]

pymanip.video.pco.pixelfly.**PCO_SetBitAlignment**(*handle*, *littleEndian*)

This functions sets the bit alignment of the transferred image data. littleEndian can be 0 or 1.

pymanip.video.pco.pixelfly.**PCO_GetImageStruct**(*handle*)

Information about previously recorded images is queried from the camera and the variables of the PCO_Image structure are filled with this information.

pymanip.video.pco.pixelfly.**PCO_GetMetaData**(*handle*, *bufNr*)

Cameras: pco.dimax and pco.edge

Query additionnal image information, which the camera has attached to the transferred image, if Meta Data mode is enabled.

pymanip.video.pco.pixelfly.**PCO_SetMetaDataMode**(*handle*, *MetaDataMode*)

Cameras: pco.dimax and pco.edge

Sets the mode for Meta Data and returns information about size and version of the Meta Data block. When Meta Data mode is set to [on], a Meta Data block with additional information is added at the end of each image. The internal buffers allocated with PCO_AllocateBuffer are adapted automatically.

pymanip.video.pco.pixelfly.**PCO_GetMetaDataMode**(*handle*)

Returns the current Meta Data mode of the camera and information about size and version of the Meta Data block.

pymanip.video.pco.pixelfly.**PCO_SetTimestampMode**(*handle*, *mode*)

Sets the timestamp mode of the camera:

| mode | short description | long description |
|---|---|---|
| 0x0000 | [off] | |
| 0x0001 | [binary] | BCD coded timestamp in the first 14 pixels |
| 0x0002 | [binary+ASCII] | BCD coded timestamp in the first 14 pixels + ASCII text |
| 0x0003 | [ASCII] | ASCII text only (see camera descriptor for availability) |

pymanip.video.pco.pixelfly.**PCO_AddBufferEx**(*handle*, *dw1stImage*, *dwLastImage*, *sBufNr*, *wXRes*, *wYRes*, *wBitPerPixel*)

This function sets up a request for a single transfer from the camera and returns immediately.

pymanip.video.pco.pixelfly.**PCO_CancelImages**(*handle*)

This function does remove all remaining buffers from the internal queue, reset the internal queue and also reset the transfer state machine in the camera. It is mandatory to call PCO_CancelImages after all image transfers are done. This function can be called before or after setting PCO_SetRecordingState to [stop].

pymanip.video.pco.pixelfly.**PCO_SetImageParameters**(*handle*, *XRes*, *YRes*, *flags*)

This function sets the image parameters for internal allocated resources. This function must be called before an image transfer is started. If next image will be transfered from a recording camera, flag IMAGEPARAMETERS_READ_WHILE_RECORDING must be set. If next action is to readout images from the camera internal memory, flag IMAGEPARAMETERS_READ_FROM_SEGMENTS must be set.

pymanip.video.pco.pixelfly.**PCO_GetImageEx**(*handle*, *segment*, *firstImage*, *lastImage*, *bufNr*, *xRes*, *yRes*, *bitsPerPixel*)

This function can be used to get a single image from the camera. The function does not return until the image is transferred to the buffer or an error occured. The timeout value for the transfer can be set with function PCO_SetTimeouts, the default value is 6 seconds. On return the image stored in the memory area of the buffer, which is addressed through parameter sBufNr.

pymanip.video.pco.pixelfly.**PCO_SetDelayExposureTime**(*handle*, *dwDelay*, *dwExposure*, *wTimeBaseDelay*, *wTimeBaseExposure*)

This function sets the delay and exposure time and the associated time base values. Restrictions for the parameter values are defined in the PCO_Description structure: - dwMinDelayDESC

- dwMaxDelayDESC
- dwMinDelayStepDESC
- dwMinExposDESC
- dwMaxExposDESC
- dwMinExposStepDESC

Possible values for wTimeBaseDelay and wTimeBaseExposure:

| Value | Unit |
|---|---|
| 0x0000 | ns |
| 0x0001 | µs |
| 0x0002 | ms |

pymanip.video.pco.pixelfly.**PCO_GetDelayExposureTime**(*handle*)

Returns the current setting of delay and exposure time

pymanip.video.pco.pixelfly.**PCO_GetTriggerMode**(*handle*)

Returns the current trigger mode setting of the camera

pymanip.video.pco.pixelfly.**PCO_SetTriggerMode**(*handle*, *mode*)

Sets the trigger mode of the camera.

pymanip.video.pco.pixelfly.**PCO_SetADCOperation**(*handle*, *operation*)

Sets the ADC (analog-digital-converter) operating mode. If sensor data is read out using single ADC operation, linearity of image data is enhanced. Using dual ADC, operation readout is faster and allows higher frame rates. If dual ADC operating mode is set, horizontal ROI must be adapted to symmetrical values.

Possible values:

| Value | Mode |
|---|---|
| 0x0001 | [single ADC] |
| 0x0002 | [dual ADC] |

pymanip.video.pco.pixelfly.**PCO_GetADCOperation**(*handle*)

Returns the ADC operation mode (single / dual)

pymanip.video.pco.pixelfly.**PCO_SetPixelRate**(*handle*, *rate*)

This functions sets the pixel rate for the sensor readout.

pymanip.video.pco.pixelfly.**PCO_GetPixelRate**(*handle*)

Returns the current pixel rate of the camera in Hz. The pixel rate determines the sensor readout speed.

pymanip.video.pco.pixelfly.**PCO_GetNoiseFilterMode**(*handle*)

This function returns the current operating mode of the image correction in the camera.

The noise filter mode:

| Value | Mode |
|---|---|
| 0x0000 | [off] |
| 0x0001 | [on] |
| 0x0101 | [on + hot pixel correction] |

pymanip.video.pco.pixelfly.**PCO_SetNoiseFilterMode**(*handle*, *mode*)

This function does set the image correction operating mode of the camera. Image correction can either be switched to totally off, noise filter only mode or noise filter plus hot pixel correction mode. The command will be rejected, if Recording State is [run], see PCO_GetRecordingState.

The noise filter mode:

| Value | Mode |
|---|---|
| 0x0000 | [off] |
| 0x0001 | [on] |
| 0x0101 | [on + hot pixel correction] |

pymanip.video.pco.pixelfly.**PCO_TriggerModeDescription**

dictionnary of trigger modes

### 4.8.3 PCO Camera module (pymanip.video.pco.camera)

This module implement the pymanip.video.pco.PCO_Camera class using bindings to the Pixelfly library from *pymanip.video.pco.pixelfly*.

**class** pymanip.video.pco.camera.**PCO_Camera**(*interface='all'*, *camera_num=0*, ***, *metadata_mode=False*, *timestamp_mode=True*)

Concrete *pymanip.video.Camera* class for PCO camera.

**Parameters**

- **interface** (*str, optional*) – interface where to look for the camera, defaults to "all"
- **camera_num** (*int, optional*) – camera number to look for, defaults to 0.
- **metadata_mode** (*bool, optional*) – enable PCO Metadata mode, defaults to False.
- **timestamp_mode** (*bool, optional*) – enable Timestamp mode (supported by all cameras), defaults to True.

**async acquisition_async**(*num=inf*, *timeout=None*, *raw=False*, *initialising_cams=None*, *raise_on_timeout=True*)

Concrete implementation of *pymanip.video.Camera.acquisition_async()* for the PCO camera.

**acquisition_oneshot**()

Concrete implementation of *pymanip.video.Camera.acquisition_oneshot()* for the PCO camera.

**property bitdepth**

Camera sensor bit depth

**close**()

This method closes the connection to the camera.

**current_adc_operation**()

This method returns the current ADC operation mode.

**Returns**
Current ADC operation mode (0x0001 for "single", 0x0002 for "dual")

**Return type**
int

**current_delay_exposure_time**()

This method returns current delay and exposure time in seconds.

**Returns**
delay, exposure

**Return type**
float, float

**current_frame_rate**()

This method returns the current frame rate.

**Returns**
Current frame rate

**Return type**
int

`current_noise_filter_mode()`

This methods queries the current noise filter mode.

**Returns**
the noise filter mode

**Return type**
int

The noise filter mode:

| Value | Mode |
|---|---|
| 0x0000 | [off] |
| 0x0001 | [on] |
| 0x0101 | [on + hot pixel correction] |

`current_pixel_rate()`

This method returns the current pixel rate.

**Returns**
Current pixel rate (e.g. 10 MHz or 40 MHz for the PCO.1600

**Return type**
int

`current_trigger_mode_description()`

This method returns the current trigger mode description.

**Returns**
description of current trigger mode

**Return type**
str

`health_status()`

This method queries the camera for its health status.

**Returns**
warn, err, status

`property name`

Camera name

`property resolution`

Camera maximum resolution

`set_adc_operating_mode(mode)`

This function selects single or dual ADC operating mode:

**Parameters**
**mode** (*int or str*) – "single" (or 0x0001) or "dual" (or 0x0002)

- Single mode increases linearity;
- Dual mode allows higher frame rates.

**set_delay_exposuretime**(*delay=None*, *exposuretime=None*)

This method sets both the delay and the exposure time.

**Parameters**

- **delay** (*float*) – delay in seconds
- **exposuretime** (*float*) – exposure time in seconds

**set_frame_rate**(*Frameratemode*, *Framerate*, *Framerateexposure*)

This method sets Frame rate (mHz) and exposure time (ns).

**Parameters**

- **Frameratemode** (*int*) – one of the possible framerate modes (0x0000, 0x0001, 0x0002, 0x0003)
- **Framerate** (*int*) – framerate in mHz
- **Framerateexposure** (*int*) – Exposure time in ns

**Returns**
message, framerate, exposure time

**Return type**
int, int, int

The meaning of the framerate mode is given in this table

| Framerate mode | Meaning |
|---|---|
| 0x0000 | Auto mode (camera decides which parameter will be trimmed) |
| 0x0001 | Frame rate has priority, (exposure time will be trimmed) |
| 0x0002 | Exposure time has priority, (frame rate will be trimmed) |
| 0x0003 | Strict, function shall return with error if values are not possible. |

The message value in return gives the limiting factors when the condition are not fulfilled. The meaning is given in this table

| Message | Meaning |
|---|---|
| 0x0000 | Settings consistent, all conditions met |
| 0x0001 | Frame rate trimmed, frame rate was limited by readout time |
| 0x0002 | Frame rate trimmed, frame rate was limited by exposure time |
| 0x0004 | Exposure time trimmed, exposure time cut to frame time |
| 0x8000 | Return values dwFrameRate and dwFrameRateExposure are not yet validated. |

In the case where message 0x8000 is returned, the other values returned are simply the parameter values passed to the function.

**set_noise_filter_mode**(*mode*)

This method does set the image correction operating mode of the camera. Image correction can either be switched totally off, noise filter only mode or noise filter plus hot pixel correction mode. The command will be rejected, if Recording State is [run], see PCO_GetRecordingState.

**Parameters**
**mode** (*int*) – the noise filter mode

The noise filter mode:

| Value | Mode |
|---|---|
| 0x0000 | [off] |
| 0x0001 | [on] |
| 0x0101 | [on + hot pixel correction] |

**set_pixel_rate**(*rate*)

This function selects the pixel rate for sensor readout.

**Parameters**
**rate** (*float*) – readout rate (in Hz)

For PCO.1600: 10 Mhz or 40 MHz

**set_roi**(*roiX0=0*, *roiY0=0*, *roiX1=0*, *roiY1=0*)

This method sets the positions of the upper left corner (X0,Y0) and lower right (X1,Y1) corner of the ROI (region of interest) in pixels.

**Parameters**

- **roiX0** (*int*) – left border in pixels, must be $1 + 32n, n \in \mathbb{N}$
- **roiY0** – top border in pixels, must be $1 + 8n, n \in \mathbb{N}$
- **roiX1** (*int*) – right border in pixels, must be $32m, m \in \mathbb{N}$
- **roiY1** – bottom border in pixels, must be $8m, m \in \mathbb{N}$

The minimum ROI is $64 \times 16$ pixels, and it is required that $roiX1 \geq roix0$ and $roiY1 \geq roiY0$.

**set_trigger_mode**(*mode*)

This method sets the trigger mode for the camera.

**Parameters**
**mode** (*str*) – one of PCO_TriggerModeDescription

Possible values are:

| mode | description |
|---|---|
| 0x0000 | auto sequence |
| 0x0001 | software trigger |
| 0x0002 | external exposure start & software trigger |
| 0x0003 | external exposure control |
| 0x0004 | external synchronized |
| 0x0005 | fast external exposure control |
| 0x0006 | external CDS control |
| 0x0007 | slow external exposure control |
| 0x0102 | external synchronized HDSDI |

pymanip.video.pco.camera.**PCO_get_binary_timestamp**(*image*)

This functions reads the BCD coded timestamp in the first 14 pixels of an image from a PCO camera.

**Parameters**
**image** (*array*) – the PCO camera image buffer

**Returns**
counter, timestamp

**Return type**
int, datetime

We assume the following format (per PCO documentation):

| Pixel | Description | Range |
|---|---|---|
| Pixel 1 | Image counter (MSB) | 00..99 |
| Pixel 2 | Image counter | 00..99 |
| Pixel 3 | Image counter | 00..99 |
| Pixel 4 | Image counter (LSB) | 00..99 |
| Pixel 5 | Year (MSB) | 20 |
| Pixel 6 | Year (LSB) | 03..99 |
| Pixel 7 | Month | 01..12 |
| Pixel 8 | Day | 01..31 |
| Pixel 9 | Hour | 00..23 |
| Pixel 10 | Minutes | 00..59 |
| Pixel 11 | Seconds | 00..59 |
| Pixel 12 | µs * 10000 | 00..99 |
| Pixel 13 | µs * 100 | 00..99 |
| Pixel 14 | µs | 00..90 |

**class** pymanip.video.pco.camera.**PCO_Buffer**(*cam_handle*, *XResAct*, *YResAct*)

This class represents an allocated buffer for the PCO camera. It implements context manager, as well as utility function to convert to bytes and numpy array. The buffer is allocated in the constructor method, and freed either by the context manager exit method, or manually calling the *free()* method.

**Parameters**

- **cam_handle** (*HANDLE*) – camera handle
- **XResAct** (*int*) – resolution in x direction
- **YResAct** (*int*) – resolution in y direction

**as_array**()

This methods returns the buffer as a numpy array. No data is copied, the memory is still bound to this buffer. The user must copy the data if necessary.

**Returns**
image array

**Return type**
numpy.ndarray

**bytes**()

This methods returns the data as a bytearray.

**Returns**
image data

**Return type**
bytearray

**free**()

This methods frees the buffer.

## 4.9 Concrete implementation for IDS camera

### 4.9.1 IDS Camera module (pymanip.video.ids)

This module implements the *pymanip.video.ids.IDS_Camera* class using the third-party pyueye module.

**class** pymanip.video.ids.**IDS_Camera**(*cam_num=0*)

Bases: *Camera*

Concrete implementation for IDS Camera.

**async acquisition_async**(*num=inf*, *timeout=1000*, *raw=False*, *initialising_cams=None*, *raise_on_timeout=True*)

Concrete implementation

**acquisition_oneshot**(*timeout_ms=1000*)

This method must be implemented in the sub-classes. It starts the camera, grab one frame, stops the camera, and returns the frame. It is useful for testing purposes, or in cases where only one frame is desired between very long time delays. It takes no input parameters. Returns an "autonomous" array (the buffer is independant of the camera object).

**Returns**
frame

**Return type**
*MetadataArray*

**current_exposure_time**()

Query the current exposure time in ms

**current_frame_rate**()

Queries the current framerate

**current_pixelclock**()

Queries the current pixelclock (in MHz)

**possible_exposure_time**()

Query the min, max and increment in ms

**possible_pixelclock**()

Query the possible values for pixelclock (in MHz)

**set_exposure_time**(*exposure_ms*)

Sets exposure time in ms. If 0 is passed, the exposure time is set to the maximum value of 1/frame rate.

**set_frame_rate**(*framerate_fps*)

Sets the framerate in frames per seconds

**set_pixelclock**(*pixelclock*)

Set the pixelclock (in MHz)

## 4.10 Concrete implementation for Ximea camera

### 4.10.1 Xiseq file parser (pymanip.video.ximea.xiseq)

**class** pymanip.video.ximea.xiseq.**XiseqFile**(*filepath*)

Simple parser for Ximea Xiseq file.

**Parameters**
**filepath** (Path or str) – path to file

**files**()

Iterate through files in sequence. Yields a dictionnary with keys `timestamp`, `frame` and `filename`.

### 4.10.2 Ximea Camera module (pymanip.video.ximea.camera)

This module implements the `pymanip.video.ximea.Ximea_Camera` using the Python module provided by Ximea.

**class** pymanip.video.ximea.camera.**Ximea_Camera**(*serial_number=None*, *pixelFormat=None*)

Bases: *Camera*

Concrete *pymanip.video.Camera* class for Ximea camera.

**async acquisition_async**(*num=inf*, *timeout=None*, *raw=False*, *initialising_cams=None*, *raise_on_timeout=True*, *wait_before_stopping=None*)

Concrete implementation of *pymanip.video.Camera.acquisition_async()* for the Ximea camera.

timeout in milliseconds.

**Parameters**

- **raw** (*bool (optional)*) – if True, returns the XI_IMG object instead of a numpy array.
- **wait_before_stopping** (*async function (optional)*) – async function to be awaited before stopping the acquisition_async

**close**()

Close connection to the camera

**get_exposure_time**()

This method gets the exposure time in seconds for the camera.

**async get_image**(*loop*, *image*, *timeout=5000*)

Asynchronous version of xiapi.Camera.get_image method This function awaits for next image to be available in transport buffer. Color images are in BGR order (similar to OpenCV default). Attention: matplotlib expects RGB order.

**Parameters**

- **image** (`xiapi.Image`) – Image instance to copy image to
- **timeout** (*int*) – timeout in milliseconds

**get_roi**()

Get ROI

**get_trigger_mode**()

Returns True if external, False if software

**property name**

Camera name

**set_auto_white_balance**(*toggle*)

Enable/Disable auto white balance

**Parameters**
**toggle** (*bool*) – True if auto white balance

**set_downsampling**(*binning*)

Change image resolution by binning

**set_exposure_time**(*seconds*)

This method sets the exposure time for the camera.

**Parameters**
**seconds** (*float*) – exposure in seconds.

**set_frame_rate**(*framerate_fps*)

Sets the framerate in frames per seconds

**set_limit_bandwidth**(*limit*)

Enable limit bandwidth (useful if there are several cameras acquiring simultaneously).

**set_roi**(*roiX0=0*, *roiY0=0*, *roiX1=0*, *roiY1=0*)

This method sets the positions of the upper left corner (X0,Y0) and lower right (X1,Y1) corner of the ROI (region of interest) in pixels.

**set_trigger_mode**(*external*)

Set external trigger (edge rising).

**set_vertical_skipping**(*factor*)

Set vertical skipping

**xidngFillMetadataFromCameraParams**()

Fills entire metadata structure with current camera parameter values obtained from xiGetParam calls.

**xidngStore**(*filename*, *img*, *camera_metadata*)

Saves the image into the file in DNG image file format. Does not support images in planar RGB and Transport formats.

## 4.11 Concrete implementation for Photometrics camera

### 4.11.1 Photometrics Camera module (pymanip.video.photometrics.camera)

This module implements the pymanip.video.photometrics.Photometrics_Camera using the Python wrapper provided by Photometrics. The documentation for the PVCAM SDK is available online.

**class** pymanip.video.photometrics.camera.**Photometrics_Camera**(*cam_num=0*, *readout_port=0*)

Bases: *Camera*

Concrete *pymanip.video.Camera* class for Photometrics camera.

**async acquisition_async**(*num=inf*, *timeout=None*, *raw=False*, *initialising_cams=None*, *raise_on_timeout=True*)

Concrete implementation of *pymanip.video.Camera.acquisition_async()* for the Photometrics camera.

timeout in milliseconds.

**close**()

Close connection to the camera

**async fast_acquisition_to_ram**(*num*, *total_timeout_s=300*, *initialising_cams=None*, *raise_on_timeout=True*)

Fast method (without the overhead of run_in_executor and asynchronous generator), for acquisitions where concurrent saving is not an option (because the framerate is so much faster than writting time), so all frames are saved in RAM anyway.

**get_exposure_time**()

This method gets the exposure time in seconds for the camera.

**set_exposure_time**(*seconds*)

This method sets the exposure time for the camera.

**Parameters**

**seconds** (*float*) – exposure in seconds.

**set_roi**(*roiX0=0*, *roiY0=0*, *roiX1=0*, *roiY1=0*)

This method sets the positions of the upper left corner (X0,Y0) and lower right (X1,Y1) corner of the ROI (region of interest) in pixels.

**set_trigger_mode**(*external*)

Set external trigger (edge rising). Possible modes are available in self.cam.exp_modes.

CHAPTER

FIVE

# ACQUISITION CARDS

## 5.1 Synchronous acquisition functions

This section describes the synchronous functions implemented in `pymanip` for signal acquisition on DAQmx and Scope boards. New scripts should preferably use the newer *pymanip.aiodaq* instead.

### 5.1.1 DAQmx acquisition module (`pymanip.daq.DAQmx`)

The `fluidlab.daq.daqmx` module is a simple functional front-end to the third-party `PyDAQmx` module. It mainly provides two simple one-liner functions:

- `read_analog()`
- `write_analog()`

The *pymanip.daq.DAQmx* module is essentially based on its fluidlab counterpart, with other choices for the default values of arguments, and an additionnal *autoset* feature for the *read_analog()* function. It also adds a convenience function for printing the list of DAQmx devices, used by the pymanip CLI interface.

A discovery function is also added, `print_connected_devices()`, based on the dedicated `DAQDevice` class, which is used by the *list_daq* sub-command on pymanip command line.

**class** pymanip.daq.DAQmx.**DAQDevice**(*device_name*)

This class is represents a DAQmx device.

**Parameters**
**device_name** (*str*) – name of the DAQmx device, e.g. “Dev1”

It mostly implement a number of property getters, which are wrappers to the `PyDAQmx` low-level functions.

In addition, it has a static method, *list_connected_devices()* to discover currently connected devices.

**property ai_chans**

List of the analog input channels on the device

**property ao_chans**

List of the analog output channels on the device

**property bus_type**

Bus type connection to the device

**property di_lines**

List of digital input lines on the device

**property di_ports**

List of digital input ports on the device

**property do_lines**

List of digital output lines on the device

**property do_ports**

List of digital output ports on the device

**static list_connected_devices()**

This static method discovers the connected devices.

**Returns**
connected devices

**Return type**
list of *pymanip.daq.DAQmx.DAQDevice* objects

**property location**

Description of the location (PCI bus and number, or PXI chassis and slot)

**property pci_busnum**

PCI Bus number

**property pci_devnum**

PCI Device number

**property product_category**

Device product category (str)

**property product_num**

Device product num

**property product_type**

Device product type

**property pxi_chassisnum**

PXI Chassis number

**property pxi_slotnum**

PXI Slot number

pymanip.daq.DAQmx.**print_connected_devices()**

This function prints the list of connected DAQmx devices.

pymanip.daq.DAQmx.**read_analog**(*resource_names*, *terminal_config*, *volt_min=None*, *volt_max=None*, *samples_per_chan=1*, *sample_rate=1*, *coupling_types='DC'*, *output_filename=None*, *verbose=True*)

This function reads signal from analog input.

**Parameters**

- **resources_names** – names from MAX (Dev1/ai0)
- **terminal_config** (*str, or list*) – "Diff", "RSE", "NRSE"
- **volt_min** (*float, or list, optional*) – minimum voltage
- **volt_max** (*float, or list, optional*) – maximum voltage
- **samples_per_chan** (*int*) – Number of samples to be read per channel

- **sample_rate** (*float*) – Clock frequency
- **coupling_type** (*str, or list*) – Coupling of the channels (“DC”, “AC”, “GND”)
- **output_filename** (*str, optional*) – If not None, file to write the acquired data
- **verbose** (*bool, optional*) – Verbosity level. Defaults to True (unlike in Fluidlab)

If the channel range is not specified, a 5.0 seconds samples will first be acquired to determine appropriate channel range (autoset feature).

### 5.1.2 Scope acquisition module (pymanip.daq.Scope)

This module implements a *read_analog()* similar to that of the *DAQmx* module, but for Scope devices. It uses the niScope module from National Instruments.

pymanip.daq.Scope.**read_analog**(*scope_name*, *channelList='0'*, *volt_range=10.0*, *samples_per_chan=100*, *sample_rate=1000.0*, *coupling_type='DC'*)

This function reads signal from a digital oscillosope.

**Parameters**

- **scope_name** – name of the NI-Scope device (e.g. ‘Dev3’)
- **channelList** (*str*) – comma-separated string of channel number (e.g. “0”)
- **volt_range** (*float*) – voltage range
- **samples_per_chan** (*int*) – number of samples to read per channel
- **sample_rate** (*float*) – for 5922 60e6/n avec n entre 4 et 1200
- **coupling_type** (*str*) – ‘DC’, ‘AC’, ‘GND’

## 5.2 Asynchronous acquisition

The *pymanip.aiodaq* implements acquisition cards in a similar manner as *pymanip.video* for cameras. The acquisition system (currently DAQmx or Scope) are represented with a single object-oriented interface, to allow to easily switch between different cards, and possibly acquire on several systems concurrently.

In addition, it provides a full GUI with both oscilloscope and signal analyser tools. This oscilloscope GUI can be invoked directly from the command line (see *Oscilloscope*):

```
$ python -m pymanip oscillo
```

Like the other sub-modules in pymanip, it is built with python standard asyncio module, so it can be easily mixed up with *pymanip.asyncsession*, *pymanip.aioinstruments* and *pymanip.video*.

### 5.2.1 Usage

To use the module, simply instantiate one of the concrete class, `DAQSystem` or *ScopeSystem*, and use their context manager. Then, the configuration is done with the methods such as *add_channel()* and *configure_clock()*, and reading is initiated and performed with the *start()*, *read()* and *stop()* methods.

Example with a Scope device:

```
import asyncio
from pymanip.aiodaq import TerminalConfig
from pymanip.aiodaq.scope import ScopeSystem, possible_sample_rates

async def main():
    with ScopeSystem('Dev3') as scope:
        scope.add_channel('0', TerminalConfig.RSE,
                          voltage_range=10.0)
        scope.configure_clock(sample_rate=min(possible_sample_rates),
                              samples_per_chan=1024)
        scope.start()
        d = await scope.read(tmo=1.0)
        await scope.stop()
    return d

asyncio.run(main())
```

## 5.3 Implementation of asynchronous acquisition

### 5.3.1 Asynchronous Acquisition Card (pymanip.aiodaq)

This module defines an abstract base class for asynchronous communication with acquisition cards. This is used by the live oscilloscope command line tool.

Concrete implementations are:

- *pymanip.aiodaq.daqmx.DAQmxSystem* for NI DAQmx cards;
- *pymanip.aiodaq.scope.ScopeSystem* for NI Scope cards.

In principle, other library bindings could be implemented.

**class** pymanip.aiodaq.**TerminalConfig**(*value*)

An enumeration.

**class** pymanip.aiodaq.**TriggerConfig**(*value*)

An enumeration.

**class** pymanip.aiodaq.**TimeoutException**

**class** pymanip.aiodaq.**AcquisitionCard**

Base class for all acquisition cards. The constructor takes no argument. Channels are added using the *add_channel()* method, and the clock is configured with the *configure_clock()* method.

**add_channel**(*channel_name*, *terminal_config*, *voltage_range*)

This method adds a channel for acquisition.

**Parameters**

- **channel_name** (*str*) – the channel to add, e.g. "Dev1/ai0"
- **terminal_config** (*TerminalConfig*) – the configuration of the terminal, i.e. RSE, NRSE, DIFFERENTIAL or PSEUDODIFFERENTIAL
- **voltage_range** (*float*) – the voltage range for the channel (actual value may differ)

**close**()

This method closes the connection to the acquisition card.

**configure_clock**(*sample_rate*, *samples_per_chan*)

This method configures the board clock for the acquisition.

**Parameters**

- **sample_rate** (*float*) – the clock frequency in Hz
- **samples_per_chan** (*int*) – number of samples to be read on each channel

**configure_trigger**(*trigger_source=None*, *trigger_level=0*, *trigger_config=TriggerConfig.EdgeRising*)

This method configures the trigger for the acquisition, i.e. internal trigger or triggered on one of the possible channels. The list of possible channels can be obtained from the *possible_trigger_channels()* method.

**Parameters**

- **trigger_source** (*str*) – the channel to use for triggering, or None to disable external trigger (switch to Immeditate trigger). Defaults to None.
- **trigger_level** (*float*) – the voltage threshold for triggering
- **trigger_config** (*pymanip.aiodaq.TriggerConfig*, optional) – the kind of triggering, e.g. EdgeRising. Defaults to EdgeRising.

**possible_trigger_channels**()

This method returns the list of channels that can be used as trigger.

**async read**(*tmo=None*)

This asynchronous method reads data from the acquisition card.

**async read_analog**(*resource_names*, *terminal_config*, *volt_min=None*, *volt_max=None*, *samples_per_chan=1*, *sample_rate=1*, *coupling_types='DC'*, *output_filename=None*, *verbose=True*)

This asynchronous method is a high-level method for simple case. It configures all the given channels, as well as the clock, then starts the acquisition, read the data, and stops the acquisition.

It is essentially similar to *pymanip.daq.DAQmx.read_analog()*, except asynchronous and functionnal for other cards than DAQmx cards.

**Parameters**

- **resource_names** (*list or str*) – list of resources to read, e.g. ["Dev1/ai1", "Dev1/ai2"] for DAQmx cards, or name of the resource if only one channel is to be read.
- **terminal_config** (*list*) – list of terminal configs for the channels
- **volt_min** (*float*) – minimum voltage expected on the channel
- **volt_max** (*float*) – maximum voltage expected on the channel
- **samples_per_chan** (*int*) – number of samples to read on each channel
- **sample_rate** (*float*) – frequency of the clock

- **coupling_type** (*str*) – coupling for the channel (e.g. AC or DC)
- **output_filename** (*str, optional*) – filename for direct writting to the disk
- **verbose** (*bool, optional*) – verbosity level

**read_analog_sync**(**args*, ***kwargs*)

Synchronous wrapper around *pymanip.aiodaq.AcquisitionCard.read_analog()*.

**read_sync**(*tmo=None*)

This method is a synchronous wrapper around *start_read_stop()* method. It is a convenience facility for simple usage.

**property samp_clk_max_rate**

Maximum sample clock rate

**start**()

This method starts the acquisition

**async start_read_stop**(*tmo=None*)

This asynchronous method starts the acquisition, reads the data, and stops the acquisition.

**Parameters**
**tmo** (*float*) – timeout for reading, defaults to None

**async stop**()

This asynchronous method aborts the acquisition

### 5.3.2 Concrete implementation with nidaqmx-python (pymanip.aiodaq.daqmx)

This module implements a concrete implementation of the *AcquisitionCard* class using the nidaqmx module.

**class** pymanip.aiodaq.daqmx.**DAQmxSystem**

This class is the concrete implementation for NI DAQmx board using the nidaqmx module.

**add_channel**(*channel_name*, *terminal_config*, *voltage_range*)

Concrete implementation of *pymanip.aiodaq.AcquisitionCard.add_channel()*.

**close**()

This method closes the active task, if there is one.

**configure_clock**(*sample_rate*, *samples_per_chan*)

Concrete implementation of *pymanip.aiodaq.AcquisitionCard.configure_clock()*

**configure_trigger**(*trigger_source=None*, *trigger_level=0*, *trigger_config=TriggerConfig.EdgeRising*)

Concrete implementation of *pymanip.aiodaq.AcquisitionCard.configure_trigger()*

**possible_trigger_channels**()

This method returns the list of channels that can be used as trigger.

**async read**(*tmo=None*)

This asynchronous method reads data from the task.

**property samp_clk_max_rate**

Maximum sample clock rate

**start**()

This method starts the task.

**async stop**()

This asynchronous method aborts the current task.

pymanip.aiodaq.daqmx.**get_device_list**()

This function returns the list of devices that the NI DAQmx library can discover.

**Returns**
dictionnary with board description as key and channels as value

**Return type**
dict

### 5.3.3 Concrete implementation with niscope (pymanip.aiodaq.scope)

This module is a concrete implementation of the *AcquisitionCard* class using the niscope module.

**Note:** We have tested this module only for with a PXI-5922 card.

**class** pymanip.aiodaq.scope.**ScopeSystem**(*scope_name=None*)

This class is the concrete implentation for NI Scope cards.

**Parameters**
**scope_name** (*str*) – the name of the scope device, e.g. "Dev1"

**add_channel**(*channel_name*, *terminal_config*, *voltage_range*)

Concrete implementation of *pymanip.aiodaq.AcquisitionCard.add_channel()*.

**close**()

This method closes the connection to the board.

**configure_clock**(*sample_rate*, *samples_per_chan*)

Concrete implementation for *pymanip.aiodaq.AcquisitionCard.configure_clock()*

**configure_trigger**(*trigger_source=None*, *trigger_level=0*, *trigger_config=TriggerConfig.EdgeRising*)

Concrete implementation for *pymanip.aiodaq.AcquisitionCard.configure_trigger()*

**possible_trigger_channels**()

This method returns the list of possible channels for external triggering.

**async read**(*tmo=None*)

Concrete implementation for *pymanip.aiodaq.AcquisitionCard.read()*

**property samp_clk_max_rate**

Maximum rate for the board clock.

**start**()

Concrete implementation for *pymanip.aiodaq.AcquisitionCard.start()*

**async stop**()

Concrete implementation for *pymanip.aiodaq.AcquisitionCard.stop()*

pymanip.aiodaq.scope.**get_device_list**(*daqmx_devices=None*, *verbose=False*)

This function gets the list of Scope device in the system. If NI System Configuration is available, the list is grabbed from this library. Otherwise, the function attempts to use the *nilsdev* command line tool.

Because *nilsdev* returns both DAQmx and Scope devices, the list of DAQmx devices is queried to remove them from the returned list. If the user code has already queried them, it is possible to pass them to avoid unnecessary double query.

**Parameters**

- **daqmx_devices** (*list, optional*) – the list of DAQmx devices.
- **verbose** (*bool, optional*) – sets verbosity level

CHAPTER

SIX

# COMMAND LINE TOOLS

The pymanip package provides several command line tools, for saved experimental session introspection and management, and for simple tests of some instruments. These tools can be invoked from the command line using the *-m pymanip*. For example for the inline manual,

```
$ python -m pymanip -h
```

## 6.1 Session introspection and management

The sessions created by pymanip use standard formats for storage:

- the synchronous sessions use both HDF5 and raw ascii files. For a session named *toto*, three files are created: *toto.dat*, *toto.hdf5* and *toto.log*;
- the asynchronous sessions use a SQLite3 database. For a session name *toto*, one file *toto.db* is created.

The rationale for having three files in the first case is that HDF5 is not designed for repeated access, and it happens sometimes that the file gets corrupted, for example if the program is interrupted while the program was writting to the disk. The ascii files however are opened in *append* mode, so that no data can ever be lost.

We switched to a database format for the newer asynchronous session, because such a format is designed for multiple concurrent access, and each transaction is atomic and can be safely rolled back if an unexpected error occurs. The risk of corruption is much less, and we thought it was no longer necessary to keep the ascii files.

Because all those file formats are simple and documented, it is possible to read and inspect them with standard tools, such as *h5dump* for the synchronous session files, or the *sqlite3* command line tool, but it is impractical to use these tools to quickly check the content of some session file. That is why a simple command line tool is provided.

The main command for introspecting saved session is

```
$ python -m pymanip info session_name
```

It can read *asynchronous sessions*, as well as synchronous sessions and old-style OctMI sessions. It is not necessary to happen the filename extensions, e.g. *.db* for asynchronous sessions. The command will output the start and end dates of the session, as well as a list of saved variables.

Two other commands are provided, but they are specific to synchronous sessions which are stored in HDF5 file format:

- the *check_hdf* sub-command checks that the ascii and HDF5 files are identical;
- the *rebuild_hdf* sub-command rebuilds the hdf file from the ascii file. This is useful if the HDF5 file has been corrupted.

## 6.2 Instrument informations and live-preview

### 6.2.1 Instrument classnames

The names of the available instrument classes available in the *pymanip.instruments* module can be conveniently obtained from the command line:

```
$ python -m pymanip list_instruments
```

### 6.2.2 Scanning for instruments

Two command line tools are provided to scan for instruments:

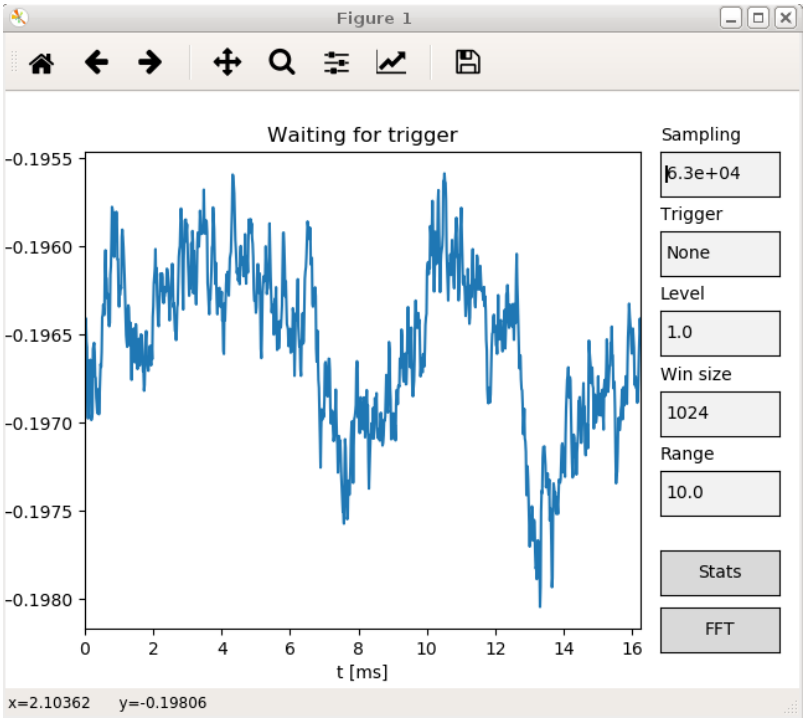
- the *list_daq* sub-command searches for acquisition cards using the NI-DAQmx library;
- the *scan_gpib* sub-command searches for connected GPIB devices using the linux-gpib free library (linux only). On Windows and Macs, this is not useful because one can simply use the GUI provided by National Instruments (NI MAX).

## 6.3 Oscilloscope

The *oscillo* sub-commands implements a simple matplotlib GUI for the *pymanip.aiodaq* module, and allows use any of these channels as oscilloscope and live signal analyser. It is simply invoked with this command

```
$ python -m pymanip oscillo
```

then the user is prompted for the channels that can be viewed (on the connected DAQmx and Scope cards).



## 6.4 Live preview

The *video* sub-command implements live video preview with the *pymanip.video* classes. Two GUI toolkits are possible: pyqt or opencv. The desired camera and acquisition parameters must be passed as argument on the command line.

## 6.5 CLI reference

pymanip CLI interface

```
usage: pymanip [-h]
               {info,list_instruments,list_daq,check_hdf,rebuild_hdf,scan_gpib,oscillo,
↪video}
               ...
```

### 6.5.1 command

**command** Possible choices: info, list_instruments, list_daq, check_hdf, rebuild_hdf, scan_gpib, oscillo, video

pymanip command

### 6.5.2 Sub-commands

#### info

shows the content of a saved pymanip session

```
pymanip info [-h] [-q] [-l line] [-p varname] session_name
```

#### Positional Arguments

**session_name** name of the saved session to inspect

#### Named Arguments

**-q, --quiet** do not list content.

Default: False

**-l, --line** print specified line of logged data.

**-p, --plot** plot the specified variable.

### list_instruments

List supported instruments

```
pymanip list_instruments [-h]
```

### list_daq

List available acquisition cards

```
pymanip list_daq [-h]
```

### check_hdf

checks dat and hdf files are identical

```
pymanip check_hdf [-h] [-p varname] session_name
```

#### Positional Arguments

**session_name** Name of the pymanip acquisition to inspect

#### Named Arguments

**-p, --plot** Plot the specified variable

### rebuild_hdf

Rebuilds a pymanip HDF5 file from the ASCII dat file

```
pymanip rebuild_hdf [-h] input_file output_name
```

#### Positional Arguments

**input_file** Input ASCII file

**output_name** Output MI session name

### scan_gpib

Scans for connected instruments on the specified GPIB board (linux-gpib only)

```
pymanip scan_gpib [-h] [board_number]
```

### Positional Arguments

**board_number** GPIB board to scan for connected instruments

Default: 0

### oscillo

Use NI-DAQmx and NI-Scope cards as oscilloscope and signal analyser

```
pymanip oscillo [-h] [-s sampling_freq] [-r volt_range] [-t level] [-T 0]
                [-b daqmx] [-p port]
                [channel_name [channel_name ...]]
```

### Positional Arguments

**channel_name** DAQmx channel names

### Named Arguments

**-s, --sampling** Sampling frequency

Default: 5000.0

**-r, --range** Channel volt range

Default: 10.0

**-t, --trigger** Trigger level

**-T, --trigsource** Trigger source index

Default: 0

**-b, --backend** Choose daqmx or scope backend

Default: “daqmx”

**-p, --serialport** Arduino Serial port

### video

Display video preview for specified camera

```
pymanip video [-h] [-l] [-i interface] [-b board [board ...]] [-t toolkit]
              [-s slice slice slice slice] [-z zoom] [-T trigger] [-w]
              [-e exposure_ms] [-d bitdepth] [-f framerate] [-r angle]
              [-R roi roi roi roi]
              camera_type
```

### Positional Arguments

**camera_type** Camera type: PCO, AVT, Andor, Ximea, IDS, Photometrics

### Named Arguments

**-l, --list** List available cameras

Default: False

**-i, --interface** Specify interface

Default: “”

**-b, --board** Camera board address

Default: 0

**-t, --toolkit** Graphical toolkit to use: cv or qt

Default: “qt”

**-s, --slice** Slice image x0, x1, y0, y1 in pixels

Default: []

**-z, --zoom** Zoom factor

Default: 0.5

**-T, --Trigger** Trigger mode

Default: -1

**-w, --whitebalance** Enable auto white balance (for color cameras)

Default: False

**-e, --exposure** Exposure time (ms)

Default: 20

**-d, --bitdepth** Bit depth

Default: 12

**-f, --framerate** Acquisition framerate in Herz

Default: 10.0

**-r, --rotate** Rotate image

Default: 0.0

**-R, --ROI** Set Region of Interest xmin, ymin, xmax, ymax

CHAPTER

SEVEN

# MISCELLANEOUS

## 7.1 Time utilities (pymanip.mytime)

This modules contains very simple functions for handling dates and times. In particular, the *datestr2epoch()* is useful to read fluidlab's session timestamps which are roughly in RFC 3339 format.

pymanip.mytime.**sleep**(*duration*)

Prints a timer for specified duration. This is mostly similar to *pymanip.asyncsession.AsyncSession.sleep()*, except that it is meant to be used outside a session.

**Parameters**
**duration** (*float*) – the duration for which to sleep

pymanip.mytime.**datestr2epoch**(*string*)

Convert datestr into epoch. Correct string is: '2016-02-25T17:36+0100' or '2016-02-25T17:36+01:00'.

UTC+1 or UTC+0100 is also accepted by this function Accepts a single string or a list of string

pymanip.mytime.**epoch2datestr**(*epoch*, *tz=None*)

Convert epoch into datestr. If no tz is given, the output is given in Coordinated Universal Time

pymanip.mytime.**tic**()

Simple timer start.

pymanip.mytime.**toc**(*comment=None*)

Simpler timer stop.

**Parameters**
**comment** (*str, optional*) – comment for print

CHAPTER

EIGHT

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## p

# INDEX

## Symbols



## A

## F



## G



## H



## I



## L



## M

## N



## O



## P

## R



## S

## T



## V



## X