Autolab

Release 1.1.12

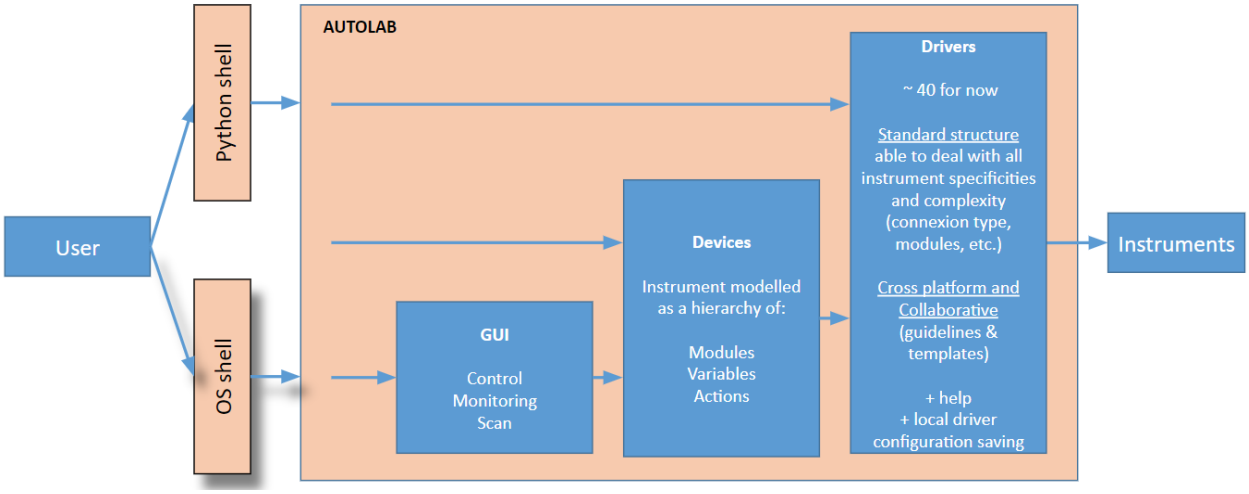
Quentin Chateiller & Bruno Garbin

Apr 15, 2023

CONTENTS

"Forget your instruments, focus on your experiment!"

Autolab is a Python package dedicated to control remotely any laboratory instruments and automate scientific experiments in the most user-friendly way. This package provides a set of standardized drivers for about 40 instruments (for now) which are ready to use, and is open to inputs from the community (new drivers or upgrades of existing ones). The configuration required to communicate with a given instrument (connection type, address, ...) can be saved locally to avoid providing it each time. Autolab can also be used either through a Python shell, an OS shell, or a graphical interface.



In this package, the interaction with a scientific instrument can be done through two different objects : the **Drivers**, or the **Devices**.

- The *Drivers (Low-level interface)* provides a raw access to the package's drivers functions.

```
>>> import autolab

>>> laserSource = autolab.get_driver('yenista_TUNICS',connection='VISA',
↪address='GPIB0::12::INSTR')
>>> laserSource.set_wavelength(1550)
>>> laserSource.get_wavelength()
1550

>>> powerMeter = autolab.get_driver('newport_1918C', connection='DLL')
>>> powerMeter.get_power()
156.89e-6

>>> stage = autolab.get_driver('newport_XPS', connection='SOCKET')
>>> stage.go_home()
```

- The *Devices (High-level interface)*, are an abstraction layer of the low-level interface that provide a simple and straightforward way to communicate with an instrument, through a hierarchy of Modules, Variables and Actions objects.

```
>>> import autolab

# Create the Device 'my_tunics' defined in 'devices_config.ini'
>>> laserSource = autolab.get_device('my_tunics')
>>> laserSource.wavelength(1550)                    # Set the Variable
```

(continues on next page)

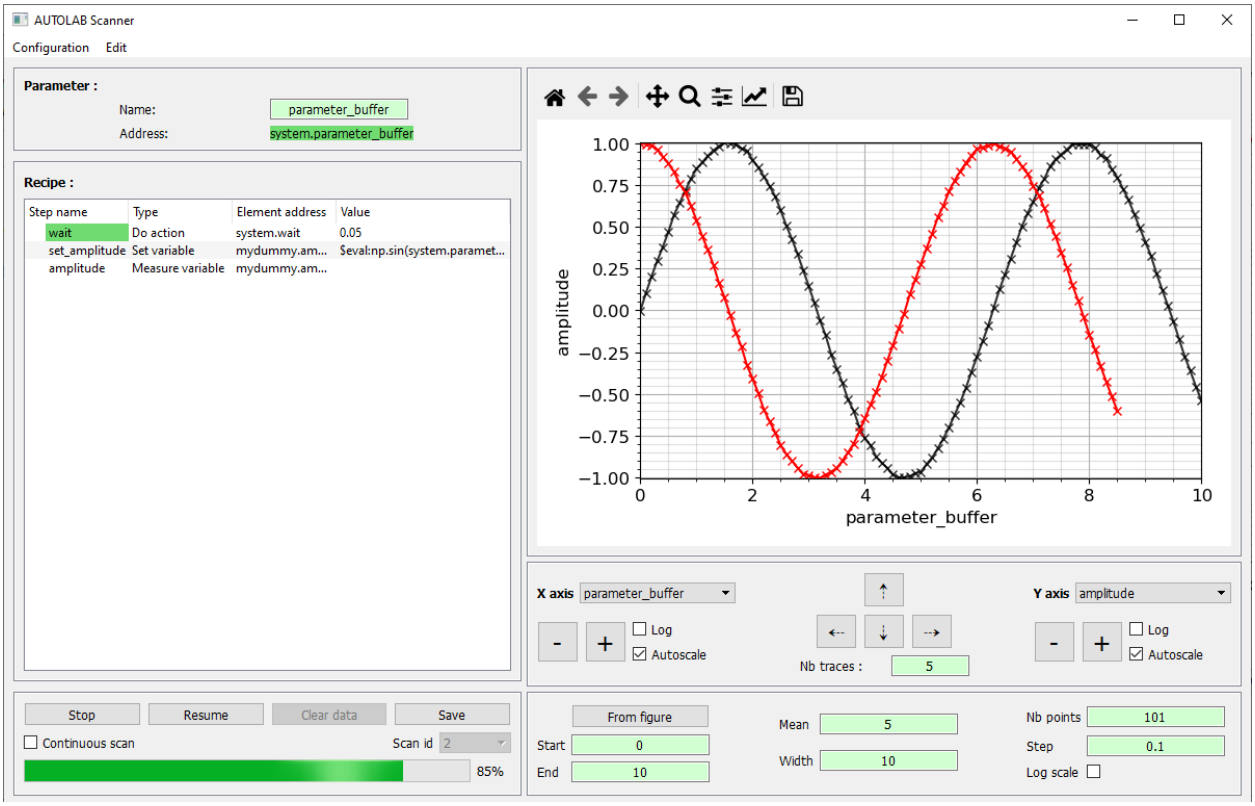
(continued from previous page)

```
↪'wavelength'
>>> laserSource.wavelength()                    # Read the Variable
↪'wavelength'
1550

>>> powerMeter = autolab.get_device('my_powermeter')    # Create the Device 'my_
↪powermeter'
>>> powerMeter.power()                          # Read the Variable
↪'power'
156.89e-6

>>> stage = autolab.get_device('my_stage')      # Create the Device 'my_
↪stage'
>>> stage.home()                                # Execute the Action
↪'home'
```

The user can also interact even more easily with this high-level interface through a user-friendly *Graphical User Interface (GUI)* which contains three panels: A Control Panel (graphical equivalent of the high-level interface), a Monitor (to monitor the value of a Variable in time) and a Scanner (to scan a Parameter and execute a custom Recipe).



All the Autolab's features are also available through an *OS shell*. interface (Windows and Linux) that can be used to perform for instance a quick single-shot operation without opening explicitely a Python shell.

```
>>> autolab driver -D yenista_TUNICS -C VISA -A GPIB0::12::INSTR -w 1551
>>> autolab device -D my_tunics -e wavelength -v 1551
```

Note: Useful links:

- Slides of our Autolab seminar (March 2020)
- Github project: github.com/qcha41/autolab
- PyPi project: pypi.org/project/autolab/
- Online documentation: autolab.readthedocs.io/

Table of contents:

CHAPTER

ONE

INSTALLATION

1.1 Python

This package is working on Python version 3.6+.

- On Windows, we recommend to install Python through the distribution Anaconda: https://www.anaconda.com/
- On older versions of Windows (before Windows 7), we recommend to install Python manually: https://www.python.org/
- On Linux, we recommend to install Python through the apt-get command.

Additional required packages (installed automatically with Autolab):

- numpy
- pandas
- pyvisa
- python-vxi11

1.2 Autolab package

This project is hosted in the global python repository PyPi at the following address : https://pypi.org/project/autolab/
To install the Autolab python package on your computer, we then advice you to use the Python package manager `pip` in a Python environnement:

```
pip install autolab
```

If the package is already installed, you can check the current version installed and upgrade it to the last official version with the following commands:

```
pip show autolab
pip install autolab --upgrade
```

Import the Autolab package in a Python shell to check that the installation is correct.

```
>>> import autolab
```

1.3 PyQt5 for the GUI

The GUI requires the package PyQt5. But depending if you are using Anaconda or not, the installation is different:

With Anaconda:

```
conda install pyqt
```

Without:

```
pip install pyqt5
```

1.4 Development version

You can install the latest development version (at your own risk) directly form GitHub:

```
pip install http://github.com/qcha41/autolab/zipball/master
```

CHAPTER

TWO

DRIVERS (LOW-LEVEL INTERFACE)

In Autolab, a **Driver** refers to a Python class dedicated to communicate with one particular instrument. This class contains functions that perform particular operations, and may also contain subclasses in case some modules or channels are present in the instrument. Autolab comes with a set of about 40 different **Drivers**, which are ready to use.

The first part of this section explains how to configure and open a **Driver**, and how to use it to communicate with your instrument. The, we present the guidelines to follow for the creation of new driver files, to contribute to the Autolab Python package.

Table of contents:

2.1 Load and use a Driver

The low-level interface provides a raw access to the drivers implemented in Autolab, through a **Driver** object, which contains functions that perform particular operations in your instrument.

Attention: The Autolab drivers may contains internal functions, that are not dedicated to be called by the user, and some functions requires particular types of inputs. **The authors declines any responsibility for the consequences of an incorrect use of the drivers**. To avoid any problems, make sure you have a real understanding of what you are doing, or prefer the use of the *Devices (High-level interface)*.

To see the list of available drivers in Autolab, call the `list_drivers` function.

```
>>> import autolab
>>> autolab.list_drivers()
```

Note: The driver of your instrument is missing ? Please contribute to Autolab by creating yourself a new driver, following the provided guidelines : *Write your own Driver*

2.1.1 Load and close a Driver

The instantiation of a *Driver* object is done through the function `get_driver` of Autolab, and requires a particular configuration:

- The name of the driver: one of the name appearing in the `list_drivers` function (ex: 'yenista_TUNICS').
- The connection parameters as keywords arguments: the connection type to use to communicate with the instrument ('VISA', 'TELNET', ...), the address, the port, the slots, ...

```
>>> laserSource = autolab.get_driver('yenista_TUNICS','VISA',address='GPIB0::12::INSTR')
```

To know what is the required configuration to interact with a given instrument, call the function `config_help` with the name of the driver.

```
>>> autolab.config_help('yenista_TUNICS')
```

To close properly the connection to the instrument, simply call its the function `close` of the **Driver**.

```
>>> lightSource.close()
```

2.1.2 Use a Driver

You are now ready to use the functions implemented in the **Driver**:

```
>>> laserSource.set_wavelength(1550)
>>> laserSource.get_wavelength()
1550
```

You can get the list of the available functions by calling the function `autolab.explore_driver` with the instance of your **Driver**. Once again, note that some of these functions are note supposed to be used directly, some of them may be internal functions.

```
>>> autolab.explore_driver(laserSource)
```

2.1.3 Script example

With all these commands, you can now create your own Python script. Here is an example of a script that sweep the wavelength of a light source, and measure a power of a power meter:

```
# Import the package
import autolab
import pandas as pd

# Open the Devices
myTunics = autolab.get_driver('yenista_TUNICS',connection='VISA',address=
↪'GPIB0::12::INSTR')
myPowerMeter = autolab.get_driver('powermeter_driver', connection='DLL')

# Turn on the light source
myTunics.set_output(True)

# Sweep its wavelength and measure a power with a power meter
```

(continues on next page)

(continued from previous page)

```
df = pd.DataFrame()
step = 0.01
start = 1550
stop = 1560
points = int(1+(stop-start)/step)
for wl in np.linspace(start, stop, points):

    # Set the parameter
    myTunics.set_wavelength(wl)

    # Measures the values
    wl_measured = myTunics.get_wavelength()
    power = myPowerMeter.line1.set_power()

    # Store the values in a list
    df = df.append({'wl_measured':wl_measured, 'power':power},ignore_index=True)

# Turn off the light source
myTunics.set_output(False)

# Close the Devices
myTunics.close()
myPowerMeter.close()

# Save data
df.to_csv('data.csv')
```

2.2 Write your own Driver

The goal of this tutorial is to present the general structure of the drivers of this package, in order for you to create simply your own drivers, and make them available to the community within this collaborative project. We notably provide a fairly understandable driver structure that can handle the highest degree of instruments complexity (including: single and multi-channels function generators, oscilloscopes, Electrical/Optical frames with associated interchangeable submodules, etc.). This provides reliable ways to add other types of connection to your driver (e.g. GPIB to Ethenet) or other functions (e.g. get_amplitude, set_frequency, etc.).

Note: To help you with writting your own drivers a few templates are provided on the GitHub page of the project.

We will first discuss the generalities to create a new driver or modify an existing one and share it with the community in **getting started: create a new driver**, that will particularly describe the required convention (location, files and namings) as well as the actual way to share it with the community (addition to the main package), and finally we will detail the typical **driver structure** as well as the required homogeneities. Those last will ensure that all the features of the drivers you would add are best used by autolab's utilities (helps, gui, parser, etc.).

2.2.1 Getting started: create a new driver

To develop your own drivers, autolab provide you with a directory named local_drivers (located at ~/autolab/local_drivers, where ~ represents the user root) created when the package is installed. This directory is inspected by autolab to search for locally defined drivers. This way you may modify existing drivers (addition of new functions, etc.) or create new drivers to drive new instruments not yet supported by autolab.

Note: Each driver name should be unique: do not define new drivers (in your local folders) with a name that already exists in the main package.

In the local_drivers directory, as in the main package, each instrument has/should have its own directory organized and named as follow. The name of this folder take the form *<manufacturer>_<MODEL>*. The driver associated to this instrument is a python script taking the same name as the folder: *<manufacturer>_<MODEL>.py*. A second python script, allowing the parser to work properly, should be named *<manufacturer>_<MODEL>_utilities.py* (find a minimal template here). Additional python scripts may be present in this folder (devices's modules, etc.). Please see the existing drivers of the autolab package for extensive examples.

For addition to the main package: Once you tested your driver and it is ready to be used by others, you can send the appropriate directory to the contacts (*About*).

Warning: General note

- The imports of additional modules (numpy, pandas, time, etc.) should be made in the class they are needed so that the imports are done only if needed (e.g. import visa within the Driver_VISA class).

2.2.2 Driver structure (*<manufacturer>_<MODEL>.py* file)

The Driver is organized in several python class with a structure as follow. The numbers represent the way sections appear from the top to the bottom of an actual driver file. We chose to present the sections in a different way:

1 - import modules (optionnal)

To import possible additional modules, e.g.:

```
import time
from numpy import zeros,ones,linspace
```

3 - class Driver_CONNECTION

The class Driver_CONNECTION: **establish the connection with the instrument** and **define the communication functions**.

As a reminder, a communication with an instruments occurs in general with strings that are set by the manufacturer and instrument and model dependent. To receive and send strings from and to the instrument we first need to establish a connection. This will be done using dedicated python package such as *pyvisa*, *pyserial*, *socket* and physical connections such as Ethernet, GPIB, or USB. See bellow for an example help with using a VISA type of connection.

Caution: The connection types are refered to with capital characters in the classes names, e.g.:

```
class Driver_SOCKET():
class Driver_TELNET():
```

When using the driver module (.py) the Driver_CONNECTION class is imported as the top layer, it inherits all the attributes of the Driver class and run its `__init__` function. It is the class that is used. Note that the connection classes are located, within a driver module, bellow the Driver class, because they use it before reaching their own `__init__` function.

Here is a commented example of the Driver_CONNECTION class, further explained bellow:

```
################################################################################
##
############################## Connections classes ##############################
##
class Driver_VISA(Driver):           # Inherits all the attributes of the␣
↪class Driver
    def __init__(self, address='GPIB0::2::INSTR',**kwargs):  # 0) Definition␣
↪of the ``__init__`` function
        import pyvisa as visa                # 1) Connection library to use

        rm = visa.ResourceManager()  # Use of visa's ressource manager
        self.inst = rm.get_instrument(address) # 2) Establish the␣
↪communication with the instrument

        Driver.__init__(self)        # 3) Run what is define in the Driver.__
↪init__ function

    # Communication functions
    def write(self,command):         # 4) Defines a write function
        self.inst.write(command)     # Sends a string 'command' to the instrument
    def read(self):                  # 5) Defines a read function
        rep = self.inst.read()       # Receives a string 'rep' from the␣
↪instrument and return it
        return rep
    def query(self,query):           # 6) Defines a query function: combine␣
↪your own write and read functions to send a string and ask for an answer
        self.write(query)
        return self.read()
    def close(self):                 # 7) Closes the communication
        self.inst.close()
############################## Connections classes ##############################
##
################################################################################
##
```

In this case the Driver_CONNECTION class is called `Driver_VISA`. To use a driver we usually create an instance of the Driver_CONNECTION class (cf. *Load and use a Driver*):

```
>>> Instance = Driver_VISA(address='GPIB0::3::INSTR')   # Use the given `visa`␣
↪address (i.e., GPIB address 3 and board_index 0)
```

This execute the `__init__` function that (following this example labels):

1) import the connection type library

2) load the instrument (using its address and eventual other arguments)

3) run the Driver.__init__ (for everything not related with the connection to the instrument, detailed in the Driver class section)

In general, the `__init__` function should establish the connection and store the instrument Instance in a class attribute (here: `self.inst`). (The communication functions that follow will use this attribute.)

Importantly, the communication functions are (re-)defined in this class including write [4)], read [5)], query [6)] and close [7)] functions that are the bare minimum. They are the ones that must be used in all the other classes (Driver, Module_, etc.). They must take **a string as argument** and **return a string**, **without any termination character** (e.g. `\n`, `\r`, etc.). This way several connection classes can coexist and use the same other classes allowing different possible physical connections and in general more flexibility.

Caution: Several points are worth noting:

- 0) The `__init__` function definition should explicitely contain all the arguments that are necessary to establish the communication (in this exemple `address`) along with a default value (for example the one that works for you), in order for the automatic autolab help to behave properly. The `__init__` function definition should also have an extra argument `**kwargs` allowing to accept and possibly pass any extra argument provided.
- 3) For more complicated instruments an additional argument `**kwargs` would be provided, giving:

  ```
  Driver.__init__(self,**kwargs)
  ```

  This enables passing extra arguments (e.g. slot configuration, etc.) to the Driver class, that will instantiate the instrument configuration, in the form of a dictionnary.
- 7) The close function is mandatory, even though you do not use it in any of the other classes of the *<manufacturer>_<MODEL>.py* file.

Further instrument complexity:

With further instrument and/or connection type complexity you will need to add other arguments to the `__init__` function of Driver_CONNECTION class. As an example to add an argument board_index for a GPIB connection type, you would need to modify the example line 0) to:

```
def __init__(self, address=19,board_index=0,**kwargs):
```

You may also need to pass arguments to the class Driver (see next section), that may come from e.g. the number of channels of an oscilloscope or the consideration of an instrument with *slots*, you would need to modify line 3) of the example:

```
Driver.__init__(self,**kwargs)
```

Please check out autolab existing drivers for more examples and/or to re-use existing connection classes (those would most likely need small adjustments to fit your instruments).

Note: Help for VISA addresses

For *visa* module to work properly, you need to provide an address for communication, that you may be able to get types the few next lines:

```
import pyvisa as visa
rm = visa.ResourceManager()
rm.list_resources()
```

Just execute them before and after plugging in your instrument to see which address appears. For ethernet connections, you should know the IP address (set it to be part of your local network) and the port (instrument documentation) of your instrument.

Examples of visa addresses may be find here online :

```
TCPIP::192.168.0.5::INSTR
GPIB0::3::INSTR
```

---

2 - class Driver

The class Driver: **establish the connection with internal modules or channels** (optionnal as dependant on the instrument, see next section) and **define instrument-related functions**.

After the communication with your instrument is established, we need to send commands or receive answers (to get the results of a query or a requested command). The communication part being manage by the class Driver_CONNECTION, any time we want to send a (instrument-specific) command to the instrument from the class Driver, we need to use the communication functions defined in the class Driver_CONNECTION.

The class Driver_CONNECTION inherits all the attributes of the class Driver. The function `__init__` of the class Driver is run by the class Driver_CONNECTION. The Driver class will act as your main instrument.

Here is a commented example of the class Driver, further explained bellow:

```
class Driver():
    def __init__(self):                    # 1) Definition of the ``__init__``␣
↪function
        import time                        # 2) Additional imports and/or␣
↪setup additional attributes

        self.write('VUNIT MV')             # 3) Run additional commands to␣
↪instantiate the instrument (e.g. set the vertical unit to be used)

    def set_amplitude(self,amplitude):     # 4) Defines a function to set a␣
↪value to the instrument
        self.write(f'VOLT {amplitude}')    # 5) Sets the amplitude, instrument␣
↪specific
    def get_amplitude(self):               # 6) Defines a function to query a␣
↪value to the instrument
        return float(self.query(f'VOLT?')) # 7) Returns the amplitude,␣
↪instrument specific
    def single_burst(self):                # 8) Defines a function to perform␣
↪an action
        self.write('BRST SINGLE')          # 9) Triggers a single burst,␣
↪instrument specific

    def idn(self):                         # 10) This function should work␣
```

(continues on next page)

(continued from previous page)

```
↪with all instruments
        self.write('*IDN?')                 # 11) '*IDN?' should be understood by␣
↪all instruments
        return self.read()                  # 12) Returns the identification of␣
↪an instrument
```

When the class Driver_CONNECTION is is instantiated, the `__init__` function is executed. It does the following (following this example labels):

1) import additional libraries

2) run additional commands to instantiate the instrument (e.g. set the vertical unit to be used)

Caution: For further instrument complexity, including multi-channels instruments (generators, oscilloscopes, etc.) or instruments with *slots*, the instantiation of additional classes must be done here. See the following examples.

In general, the `__init__` function should run instrument-related initializations. If nothing in particular needs to be done then, one can just:

```
def __init__(self,nb_channels=2):       # 1)
    pass
```

Importantly, the class Driver defines all the functions that are related to the main instrument: to set [4)]/query [6)] some values (e.g. the output amplitude of a function generator) or perform actions (e.g. trigger a single burst event).

Caution: Several points are worth noting:

1) Favor python f strings (`f''`) that are more, especially when an argument has to be passed to the function, that are more robust to different types [5)].

2) You should explicitely convert the string returned by Driver_CONNEXION.query() (or Driver_CONNEXION.read) to the expected *variable* type [7)].

3) For more complex instruments (i.e. with additional classes), please refer to the next section. In general, only the functions associated with the **main** instrument should be found here.

Further instrument complexity:

Here is a way to modify the `__init__` function of the class Driver to deal with the case of a **multi-channel instrument**. (Note: some of the lines have been removed from the previous example for clarity.) It is further explained bellow:

```
def __init__(self,nb_channels=2):       # 1) Definition of the ``__init_
↪_`` function

    self.nb_channels = int(nb_channels) # 2) Set arguments given to␣
↪the class as class attributes to be re-used elsewhere (within the␣
↪class)

    for i in range(1,self.nb_channels+1):
        setattr(self,f'channel{i}',Channel(self,i)) # 3) Set␣
```

(continues on next page)

(continued from previous page)

```
↪additional Module\_MODEL classes (called Channel here) as classes␣
↪attibutes
```

Here, the number of channels is provided as argument to the `__init__` function [1)], and for each channel [3)] an attribute of the class Driver is created by instantiating an additional class called **Channel**. The line 3) is formally equivalent to (considering: i=1):

```
self.channel1 = Channel(self,1)
```

All the channels are thus equivalent in this example as they use the same additional class (**Channel**). The arguments provided to the class **Channel** are: all the attributes of the actual class (**Driver**) and the number of the instantiated channel; both will be used in the additional class (e.g. the connection functions, etc.)

The previous structure should be used only if the physical slot configuration is naturally fixed by the manufacturer (a power meter with two channels for instance). In the particular case of an **instrument with `slots`**, all the *channels* are not equivalent. They rely on different physical modules that may be disposed differently and in different numbers for different users. Then one class for each different module (that are inserted in a main frame) should be defined (**Module_MODEL**). Here is a way to modify the `__init__` function of the class Driver to deal with the case of an instrument with *slots*:

```
def __init__(self, **kwargs):

    ### Submodules loading
    self.slot_names = {}
    prefix = 'slot'
    for key in kwargs.keys():
        if key.startswith(prefix) and not '_name' in key :
            slot_num = key[len(prefix):]
            module_name = kwargs[key].strip()
            module_class = globals()[f'Module_{module_name}']
            if f'{key}_name' in kwargs.keys() : name = kwargs[f'{key}_
↪name']
            else : name = f'{key}_{module_name}'
            setattr(self,name,module_class(self,slot_num))
            self.slot_names[slot_num] = name
```

This will parse the arguments received by the `__init__` function (of the class **Driver**) in the `**kwargs` appropriately to instantiate the right combination Modules/Slots providing the Modules (additional classes) follow some naming conventions (explained in the next section).

Note: For the particular case of instruments that one usually gets 1 dimensionnal traces from (e.g. oscilloscope, spectrum annalyser, etc.), it is useful to add to the class Driver some user utilities such as procedure for channel acquisitions:

```
### User utilities
def get_data_channels(self,channels=[],single=False):
    """Get all channels or the ones specified"""
    previous_trigger_state = self.get_previous_trigger_state()          ␣
↪          # 1)
    self.stop()                                                         ␣
↪          # 2)
```

(continues on next page)

(continued from previous page)

```
    if single: self.single()                                                    ␣
↪           # 3)
    while not self.is_stopped(): time.sleep(0.05)                               ␣
↪           # 4)
    if channels == []: channels = list(range(1,self.nb_channels+1))
    for i in channels:
        if not(getattr(self,f'channel{i}').is_active()): continue
        getattr(self,f'channel{i}').get_data_raw()                              ␣
↪           # 5)
        getattr(self,f'channel{i}').get_log_data()                              ␣
↪           # 6)
    self.set_previous_trigger_state(previous_trigger_state)                     ␣
↪           # 7)

def save_data_channels(self,filename,channels=[],FORCE=False):
    if channels == []: channels = list(range(1,self.nb_channels+1))
    for i in channels:
        getattr(self,f'channel{i}').save_data_raw(filename=filename,
↪FORCE=FORCE) # 8)
        getattr(self,f'channel{i}').save_log_data(filename=filename,
↪FORCE=FORCE) # 9)
```

These functions rely on some other functions that should be implemented by the user (`single`, `get_previous_trigger_state`, etc.). The reader may find a find a full template example here.

Overall, the function get_data_channels:

1) Store the previous trigger state
2) Stop the instrument
3) Trigger a single trigger event (if requested)
4) Wait for the scope to be stopped
5) Acquire the channels provided (all if no channel is provided)
6) Acquire the logs of the channels provided (all if no channel is provided)
7) Set the previous trigger state back

Overall, the function save_data_channels:

8) Save the channels provided (all if no channel is provided)
9) Save the logs of the channels provided (all if no channel is provided)

4 - Additional class (optionnal)

> **Caution: Additional classes namings**
>
> The additional classes should be named **Module_MODEL**. Exceptions do occur for some oscilloscopes (**Channel**), spectrum annalyser (**Trace**) or some multi-channel instruments (**Output**), in which case we stick to the way it is refered to as in the Programmer Manual of the associated instrument.

In the particular case of an **instrument with `slots`**, all the *channels* are not equivalent. They rely on different physical modules that may be disposed differently and in different numbers for different users. Then one class for each different module (that are inserted in a main frame) should be defined (**Module_MODEL**). The `__init__` function of the class **Driver** will deal with which class **Module_MODEL** to instantiate with which *slot* depending on the actual configuration of the user. Thus the class **Module_MODEL** (or **Channel**, etc.) have all a similar structure, structure that is similar to the one of the class Driver. In other words the class **Driver** deal with the *main* instruments while the additional classes deal with the sub-modules.

Here is an example of the class Channel of a double channel function generator:

```
class Channel():
    def __init__(self,dev,channel):
        self.channel = int(channel)
        self.dev     = dev

    def amplitude(self,amplitude):
        self.dev.write(f':VOLT{self.channel} {amplitude}')
    def offset(self,offset):
        self.dev.write(f':VOLT{self.channel}:OFFS {offset}')
    def frequency(self,frequency):
        self.dev.write(f':FREQ{self.channel} {frequency}')
```

Here is an example of the two class Module_MODEL of a instrument with *slot* for which slots are non-equivalent (strings needed to perform the same actions are different):

```
class Module_TEST111() :
    def __init__(self,driver,slot):
        self.driver = driver
        self.slot   = slot

    def set_power(self,value):
        self.dev.write(f'POWER={value}')
    def get_power(self):
        return float(self.dev.query('POWER?'))

class Module_TEST222() :
    def __init__(self,driver,slot):
        self.driver = driver
        self.slot   = slot

    def set_power(self,value):
        self.dev.write(f'POWER={value}')
    def get_power(self):
        return float(self.dev.query('POWER?'))
```

One can note (for both cases):

1) In the `__init__` function both the driver `self` and the channel/slot naming are passed to an attribute of the actual class (**Channel**, **Module_TEST111**, **Module_TEST222**).

2) The connection functions used are the one coming from the class **Driver**, thus one now call them `self.dev.connection_function` (for connection_function defined in the class **Driver_CONNECTION** in: write, read, query, etc.).

3) Finally there is a collection of functions that are *channel*/*slot*-dependant.

Note: For the particular case of instruments that one usually gets 1 dimensionnal traces from (e.g. oscilloscope, spectrum annalyser, etc.), it is useful to define functions to get and save the data. See the folliwing instrument dependant example:

```
def get_data_raw(self):
    if self.autoscale:
        self.do_autoscale()
    self.dev.write(f'C{self.channel}:WF? DAT1')
    self.data_raw = self.dev.read_raw()
    self.data_raw = self.data_raw[self.data_raw.find(b'#')+11:-1]
    return self.data_raw
def get_data(self):
    return frombuffer(self.get_data_raw(),int8)
def get_log_data(self):
    self.log_data = self.dev.query(f"C{self.channel}:INSP? 'WAVEDESC'")
    return self.log_data

def save_data_raw(self,filename,FORCE=False):
    temp_filename = f'{filename}_WAVEMASTERCH{self.channel}'
    if os.path.exists(os.path.join(os.getcwd(),temp_filename)) and not(FORCE):
        print('\nFile ', temp_filename, ' already exists, change filename or␣
↪remove old file\n')
        return
    f = open(temp_filename,'wb')# Save data
    f.write(self.data_raw)
    f.close()
def save_log_data(self,filename,FORCE=False):
    temp_filename = f'{filename}_WAVEMASTERCH{self.channel}.log'
    if os.path.exists(os.path.join(os.getcwd(),temp_filename)) and not(FORCE):
        print('\nFile ', temp_filename, ' already exists, change filename or␣
↪remove old file\n')
        return
    f = open(temp_filename,'w')
    f.write(self.log_data)
    f.close()
```

Those will then be attributes of the class **Channel** and may be called from the class **Driver** (depending on the channel's instance name in this class):

```
self.channel1.get_data()
```

2.2.3 Additional necessary functions/files

Function get_driver_model (in each class but Driver_CONNECTION)

The function `get_driver_model` should be present in each of the classes of the *<manufacturer>_<MODEL>.py* but the class **Driver_CONNECTION** (including the class Driver and any optionnal class **Module_MODEL**), in order for many features of the package to work properly. It simply consists in a list of predefined elements that will indicate to the package the structure of the driver and predefined variable and actions. There are three possible elements in the function `get_driver_model`: *Module*, *Variable* and *Action*.

Shared by the three elements (*Module*, *Variable*, *Action*):

- 'name': nickname for your element (argument type: string)
- 'element': element type, exclusively in: 'module', 'variable', 'action' (argument type: string)
- 'help': quick help, optionnal (argument type: string)

***Module*:**

- 'object' : attribute of the class (argument type: Instance)

***Variable*:**

- 'read': class attribute (argument type: function)
- 'write': class attribute (argument type: function)
- 'type': python type, exclusively in: int, float, bool, str, bytes, np.ndarray, pd.DataFrame
- 'unit': unit of the variable, optionnal (argument type: string)

> **Caution:** Either 'read' or 'write' key, or both of them, must be provided.

***Action*:**

- 'do' : class attribute

Example code:

```
def get_driver_model(self):
    model = []
    model .append({'name':'line1', 'element':'module','object':self.slot1,'help':'Simple␣
↪help for line1 module'})
    model .append({'name':'amplitude', 'element':'variable', 'type':float, 'read':self.
↪get_amplitude, 'write':self.set_amplitude, 'unit':'V', 'help':'Simple help for␣
↪amplitude variable'}
    model.append({'name':'go_home', 'element':'action', 'read':self.home, 'help':'Simple␣
↪help for go_home action'})
return model
```

Driver utilities structure (*<manufacturer>_<MODEL>_utilities.py* file)

This file should be present in the driver directory (*<manufacturer>_<MODEL>.py*).

Here is a commented example of the file *<manufacturer>_<MODEL>_utilities.py*, further explained bellow:

```
category = 'Optical source'                                     #

class Driver_parser():                                          #
    def __init__(self, Instance, name, **kwargs):               #
        self.name     = name                                    #
        self.Instance = Instance                                #


    def add_parser_usage(self,message):                         #
        """Usage to be used by the parser"""                    #
        usage = f"""                                            #
{message}                                                       #
                                                                #
----------------  Examples:  ----------------                   #
                                                                #
usage:    autolab driver [options] args                         #
                                                                #
    autolab driver -D {self.name} -A GPIB0::2::INSTR -C VISA -a 0.2
    load {self.name} driver using VISA communication protocol with address GPIB... and␣
↪set the laser pump current to 200mA.
            """                                                 #
        return usage                                            #

    def add_parser_arguments(self,parser):                      #
        """Add arguments to the parser passed as input"""       #
        parser.add_argument("-a", "--amplitude", type=str, dest="amplitude",␣
↪default=None, help="Set the pump current value in Ampere." )

        return parser                                           #

    def do_something(self,args):                                #
        if args.amplitude:                                      #
            # next line equivalent to: self.Instance.amplitude = args.amplitude
            getattr(self.Instance,'amplitude')(args.amplitude)

    def exit(self):                                             #
        self.Instance.close()                                   #
```

It contains:

- The category of the instrument (see `autolab.infos` (from python shell) or `autolab infos` for (OS shell) for examples of identified categories).
- A class **Driver_parser** with 5 functions:

  1) `__init__`: defines class attributes

  2) `add_parser_usage`: adds help to the parser in order to help the user

  3) `add_parser_arguments`: configures options to be used from the OS shell (e.g. `autolab driver -D nickname -a 2`). See *Command driver* for full usage.

4) `do_something`: configures action to perform/variable to set (here: modify the amplitude to the the provided argument value), and link them to the values of the argument added with **3**).

5) `exit`: closes properly the connection

Note: Please do consider, keeping each line ending with a # character in the example as is.This way you would need to modify 3 main parts to configure options, associated actions and help: **3**), **4**) and **2**) (respectively).

CHAPTER

THREE

DEVICES (HIGH-LEVEL INTERFACE)

3.1 What is a Device ?

The high-level interface of Autolab is an abstraction layer of its low-level interface, which allows to communicate easily and safely with laboratory instruments without knowing the structure of its associated **Driver**.

In this approach, an instrument is fully described with a hierarchy of three particular **Elements**: the **Modules**, the **Variables** and the **Actions**.

- A **Module** is an **Element** that consists in a group of **Variables**, **Actions**, and sub-**Modules**. The top-level **Module** of an instrument is called a **Device**.
- A **Variable** is an **Element** that refers to a physical quantity, whose the value can be either set and/or read from an instrument (wavelength of an optical source, position of a linear stage, optical power measured with a power meter, spectrum measured with a spectrometer...). Depending on the nature of the physical quantity, it may have a unit.
- An **Action** is an **Element** that refers to a particular operation that can be performed by an instrument. (homing of a linear stage, the zeroing of a power meter, the acquisition of a spectrum with a spectrometer...). An **Action** may have a parameter.

The **Device** of a simple instrument is usually represented by only one **Module**, and a few **Variables** and **Actions** attached to it.

```
-- Tunics (Module/Device)
        |-- Wavelength (Variable)
        |-- Output state (Variable)
```

Some instruments are a bit more complex, in the sense that they can host several different modules. Their representation in this interface generally consists in one top level **Module** (the frame) and several others sub-**Modules** containing the **Variables** and **Actions** of each associated modules.

```
-- XPS Controller (Module/Device)
        |-- ND Filter (Module)
                |-- Angle (Variable)
                |-- Transmission (Variable)
                |-- Homing (Action)
        |-- Linear stage (Module)
                |-- Position (Variable)
                |-- Homing (Action)
```

This hierarchy of **Elements** is implemented for each instrument in its drivers files, and is thus ready to use.

3.2 Load and close a Device

The procedure to load a **Device** is almost the same as for the **Driver**, but with the function `get_device`. You need to provide the nickname of a driver defined in the `devices_config.ini` (see *Local configuration*).

```
>>> lightSource = autolab.get_device('my_tunics')
```

Note: You can overwrite temporarily some of the parameters values of a configuration by simply providing them as keywords arguments in the `get_device` function:

```
>>> laserSource = autolab.get_device('my_tunics',address='GPIB::9::INSTR')
```

To close properly the connection to the instrument, simply call its the function `close` of the **Device**. This object will not be usable anymore.

```
>>> lightSource.close()
```

To close all devices connection (not drivers) at once you can use the Autolab close function.

```
>>> autolab.close()
```

3.3 Navigation and help in a Device

The navigation in the hierarchy of **Elements** of a given **Device** is based on relative attributes. For instance, to access the **Variable** `wavelength` of the **Module** (**Device**) `my_tunics`, simply execute the following command:

```
>>> lightSource.wavelength
```

In the case of a more complex **Device**, for instance a power meter named `my_power_meter` that has several channels, you can access the **Variable** `power` of the first channel `channel1` with the following command:

```
>>> powerMeter = autolab.get_device('my_power_meter')
>>> powerMeter.channel1.power
```

Every **Element** in Autolab is provided with a function `help` that can be called to obtain some information about it, but also to know which further **Elements** can be accessed through it, in the case of a **Module**. For a **Variable**, it will display its read and/or write functions (from the driver), its python type, and its unit if provided in the driver. For an **Action**, il will display the associated function in the driver, and its parameter (python type and unit) if it has one. You can also `print()` the object to display this help.

```
>>> lightSource.help()
>>> print(lightSource.wavelength)
>>> powerMeter.help()
>>> print(powerMeter.channel1)
>>> powerMeter.channel1.power.help()
```

3.4 Use a Variable

If a **Variable** is readable (read function provided in the driver), its current value can be read by calling its attribute:

```
>>> lightSource.wavelength()
1550.55
>>> lightSource.output()
False
```

If a **Variable** is writable (write function provided in the driver), its current value can be set by calling its attribute with the desired value:

```
>>> lightSource.wavelength(1549)
>>> lightSource.output(True)
```

To save locally the value of a readable **Variable**, use its function *save* with the path of the desired output directory (default filename), or file:

```
>>> lightSource.wavelength.save('.\mesures\')
>>> lightSource.wavelength.save('.\mesures\power.txt')
```

3.5 Use an Action

You can execute an **Action** simply by calling its attribute:

```
>>> linearStage = autolab.get_device('my_linear_stage')
>>> linearStage.goHome()
```

3.6 Script example

With all these commands, you can now create your own Python script. Here is an example of a script that sweep the wavelength of a light source, and measure a power of a power meter:

```
# Import the package
import autolab
import pandas as pd

# Open the Devices
myTunics = autolab.get_device('my_tunics')
myPowerMeter = autolab.get_device('my_power_meter')

# Turn on the light source
myTunics.output(True)

# Sweep its wavelength and measure a power with a power meter
df = pd.DataFrame()
step = 0.01
start = 1550
stop = 1560
```

(continues on next page)

(continued from previous page)

```
points = int(1+(stop-start)/step)
for wl in np.linspace(start, stop, points):

    # Set the parameter
    myTunics.wavelength(wl)

    # Measures the values
    wl_measured = myTunics.wavelength()
    power = myPowerMeter.line1.power()

    # Store the values in a list
        df = df.append({'wl_measured':wl_measured, 'power':power},ignore_index=True)

# Turn off the light source
myTunics.output(False)

# Close the Devices
myTunics.close()
myPowerMeter.close()
# Or use autolab.close()

# Save data
df.to_csv('data.csv')
```

CHAPTER

FOUR

LOCAL CONFIGURATION

To avoid having to provide each time the full configuration of an instrument (connection type, address, port, slots, ...) to load a **Device**, Autolab proposes to store it locally for further use.

More precisely, this configuration is stored in a local configuration file named `devices_config.ini`, which is located in the local directory of Autolab. Both this directory and this file are created automatically in your home directory the first time you use the package (the following messages will be displayed, indicating their exact paths).

```
INFORMATION: The local directory of AUTOLAB has been created: C:\Users\<USER>\autolab
INFORMATION: The devices configuration file devices_config.ini has been created: C:\
↪Users\<USER>\autolab\devices_config.ini
```

Warning: Do not move or rename the local directory nor the configuration file.

A device configuration is composed of several parameters:

- The name of the device, which is usually the nickname of your instrument in Autolab.
- The name of the associated Autolab **driver**.
- All the connection parameters (connection, address, port, slots, ...)

To see the list of the available devices configurations, call the function `list_devices`.

```
>>> autolab.list_devices()
```

To know what parameters have to be provided for a particular **Device**, use the function *config_help* with the name of corresponding driver.

```
>>> autolab.config_help('yenista_TUNICS')
```

4.1 Edit the configuration file

You can manually edit the devices configuration file `devices_config.ini`.

This file is structured in blocks, each of them containing the configuration of an instrument. Each block contains a header (the configuration name / nickname of the instrument in square brackets `[ ]`). The parameters and values are then listed below line by line, separated by an equal sign =.

```
[<NICKNAME_OF_YOUR_DEVICE>]
driver = <DRIVER_NAME>
connection = <CONNECTION_TYPE>
address = <ADDRESS>
slot1 = <MODULE_NAME>
slot1_name = <MY_MODULE_NAME>
```

To see a concrete example of the block you have to append in the configuration file for a given driver, call the function `config_help` with the name of the driver. You can then directly copy and paste this exemple into the configuration file, and customize the value of the parameters to suit those of your instrument. Here is an example for the Yenista Tunics light source:

```
[my_tunics]
driver = yenista_TUNICS
connection = VISA
address = GPIB0::2::INSTR
```

Save the configuration file, and go back to Autolab. You don't need to restart Autolab, the configuration file will be read automatically at the next request.

```
>>> laserSource = autolab.get_device('my_tunics')
```

CHAPTER

FIVE

GRAPHICAL USER INTERFACE (GUI)

Autolab is provided with a user-friendly graphical interface based on the **Device** interface, that allows the user to interact even more easily with its instruments. It can be used only for local configurations (see *Local configuration*).

The GUI has four panels : a **Control Panel** that allows to see visually the architecture of a **Device**, and to interact with an instrument through the *Variables* and *Actions*. The **Monitoring Panel** allows the user to monitor a *Variable* in time. The **Scanning Panel** allows the user to configure the scan of a parameter and the execution of a custom recipe for each value of the parameter. The **Plotting Panel** allows the user to plot data.

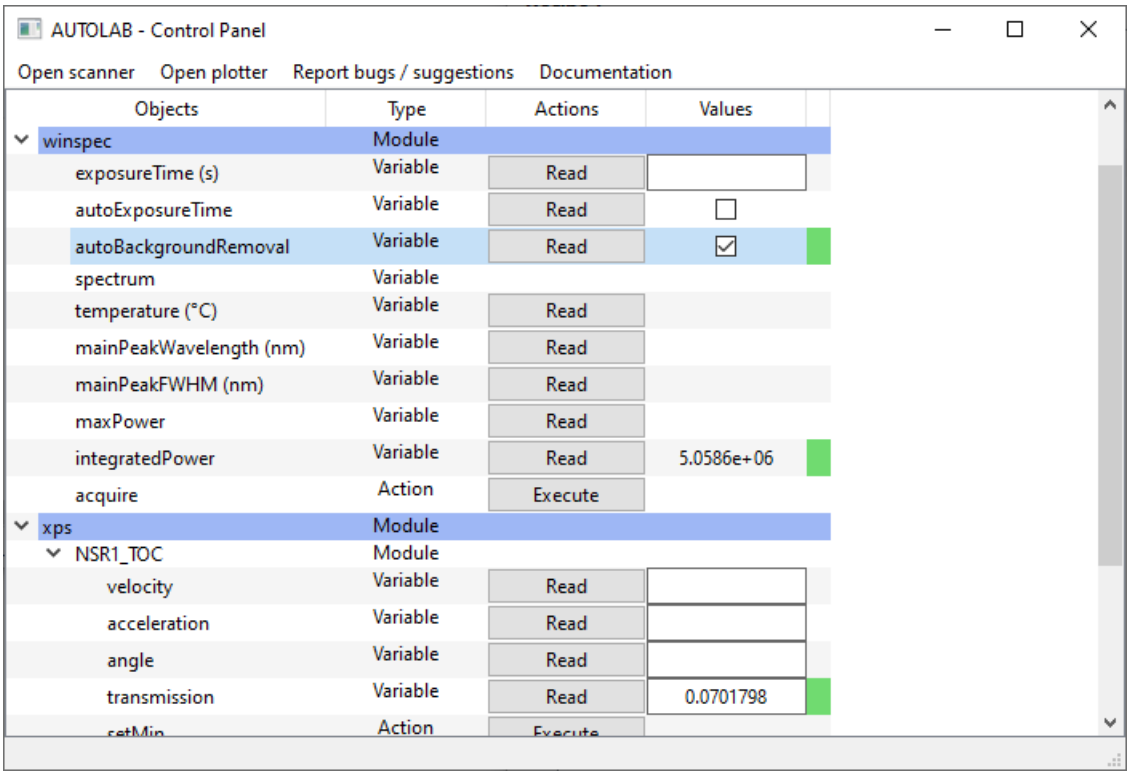


Fig. 1: Control panel

To start the GUI from a Python shell, call the function `gui` of the package:

```
>>> import autolab
>>> autolab.gui()
```

To start the GUI from an OS shell, call:

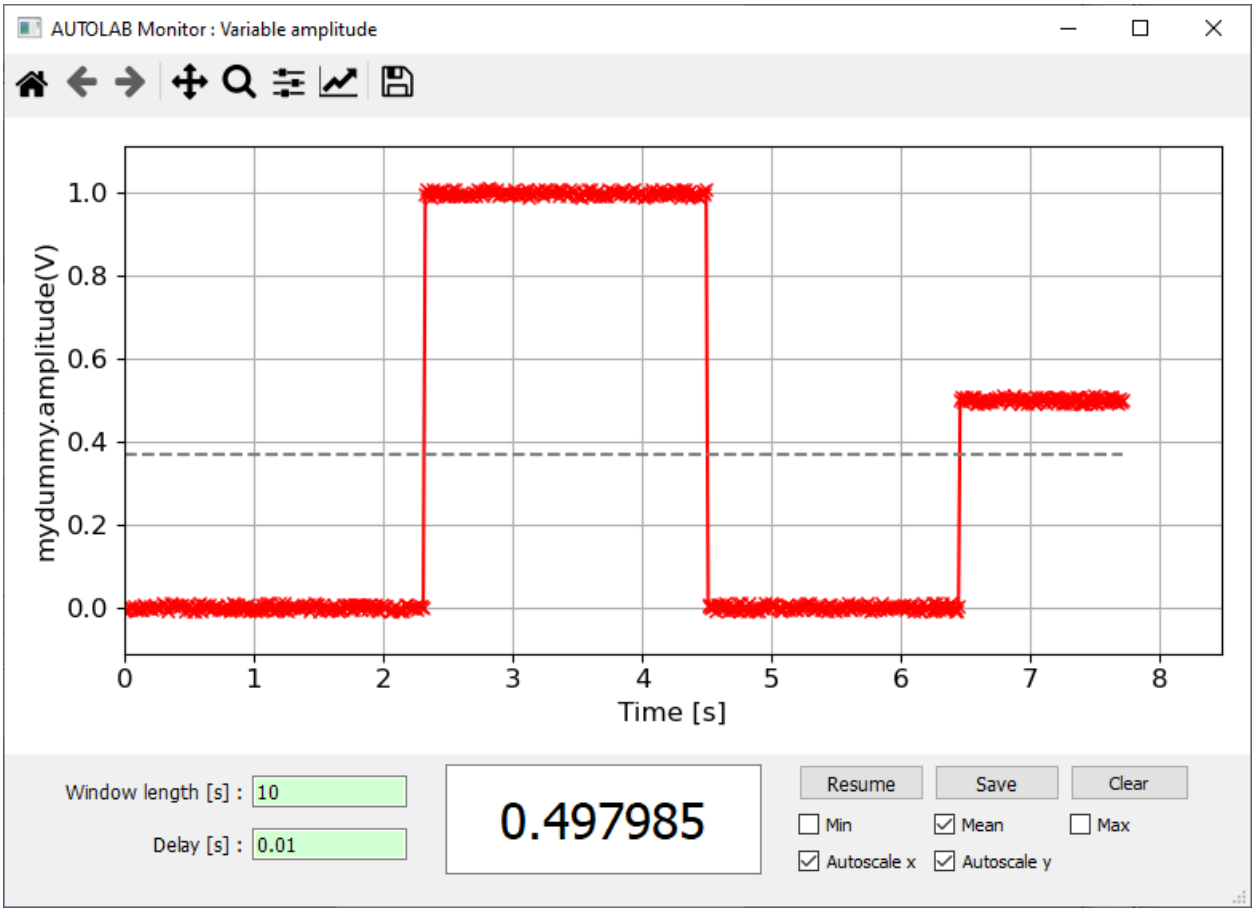


Fig. 2: Monitoring panel

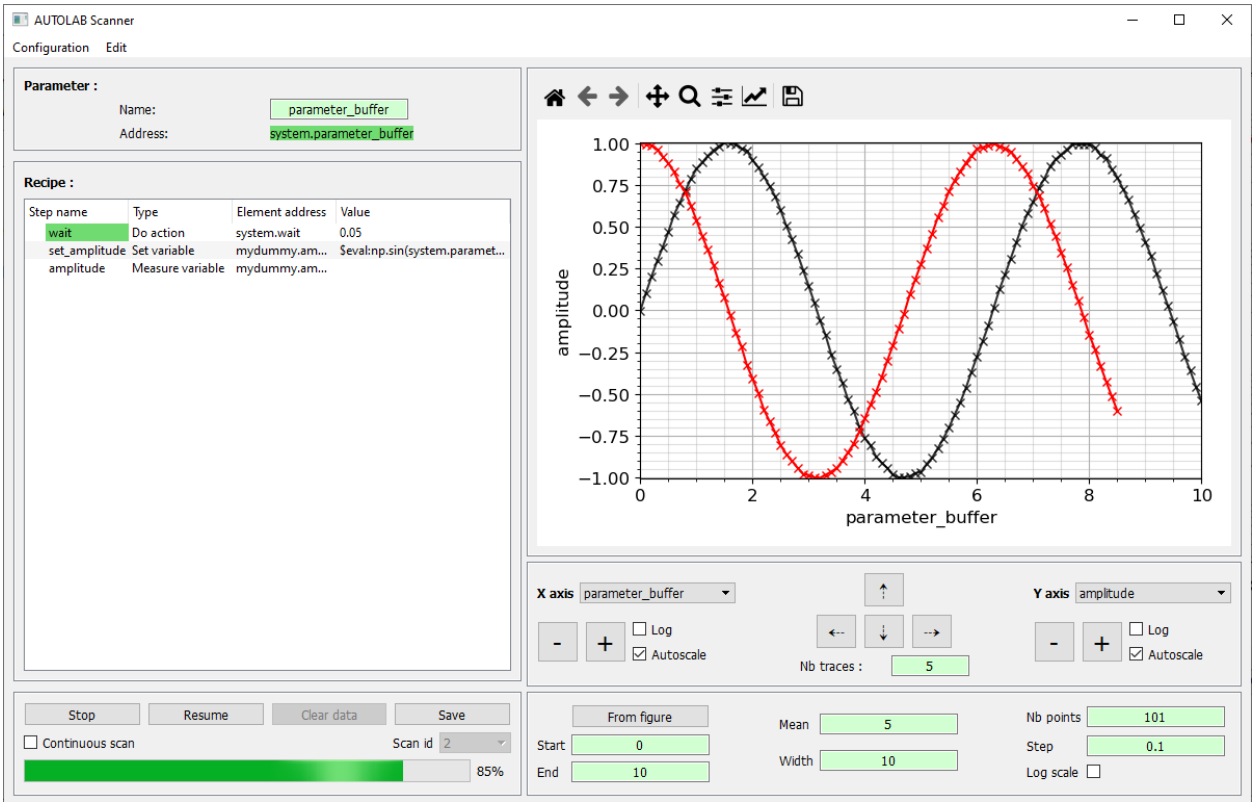


Fig. 3: Scanning panel

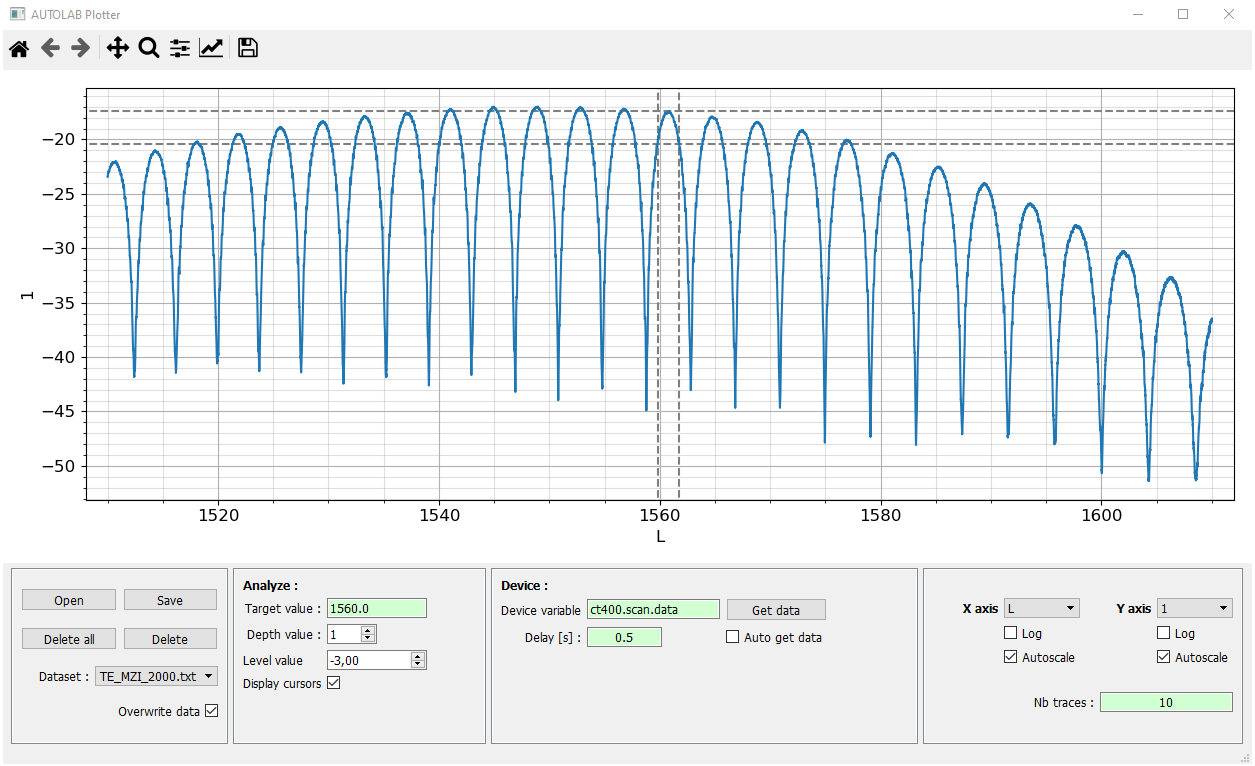
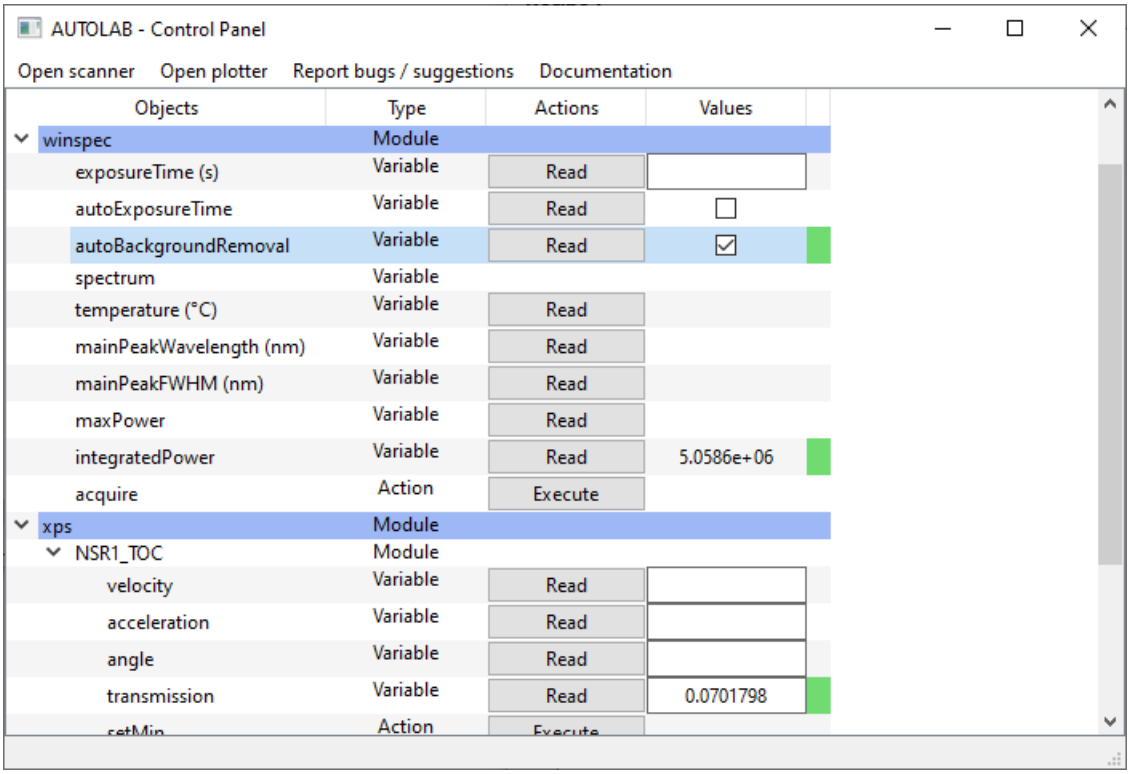


Fig. 4: Plotting panel

```
>>> autolab gui
```

5.1 Control panel

The Autolab GUI Control Panel provides an easy way to control your instruments. From it, you can visualize and set the value of its *Variables*, and execute its *Action* through graphical widgets.



5.1.1 Devices tree

By default, the name of each local configuration in represented in a tree widget. Click on one of them to load the associated **Device**. Then, the corresponding *Element* hierarchy appears. Right-click to bring up the close option.

The help of a given **Element** (see *Devices (High-level interface)*) can be displayed though a tooltip by passing the mouse over it (if provided in the driver files).

Actions

A button **Execute** is present in each *Action* line. Clicking the button executes the associated action. If the *Action* has a parameter, fill its value in the associated widget.

Variables

The value of a *Variable* can be set or read if its type is numerical (integer, float or boolean).

If the *Variable* is readable (read function provided in the driver), a **Read** button is available on its line. When clicking on this button, the *Variable*'s value is read and displayed in a line edit widget (integer / float values) or in a checkbox (boolean).

If the *Variable* is writable (write function provided in the driver), its value can be edited and sent to the instrument (return pressed for interger / float values, check box checked or unchecked for boolean values). If the *Variable* is also readable, a **Read** operation will be executed automatically after that.

To read and save the value of a *Variable*, right click on its line and select **Read and save as...**. You will be prompted to select the path of the output file.

The colored displayed at the end of a line corresponds to the state of the displayed value:

- The orange color means that the currently displayed value is not necessary the current value of the **Variable** in the instrument. The user should click the **Read** button to update the value in the interface.
- The green color means that the currently displayed value is up to date (except if the user modified its value directly on the instrument. In that case, click the **Read** button to update the value in the interface).

5.1.2 Monitoring

A readable and numerical *Variable* can be monitored in time (single point, 1D and 2D array versus time). To start the monitoring of this *Variable*, right click on it and select **Start monitoring**. Please visit the section *Monitoring*.

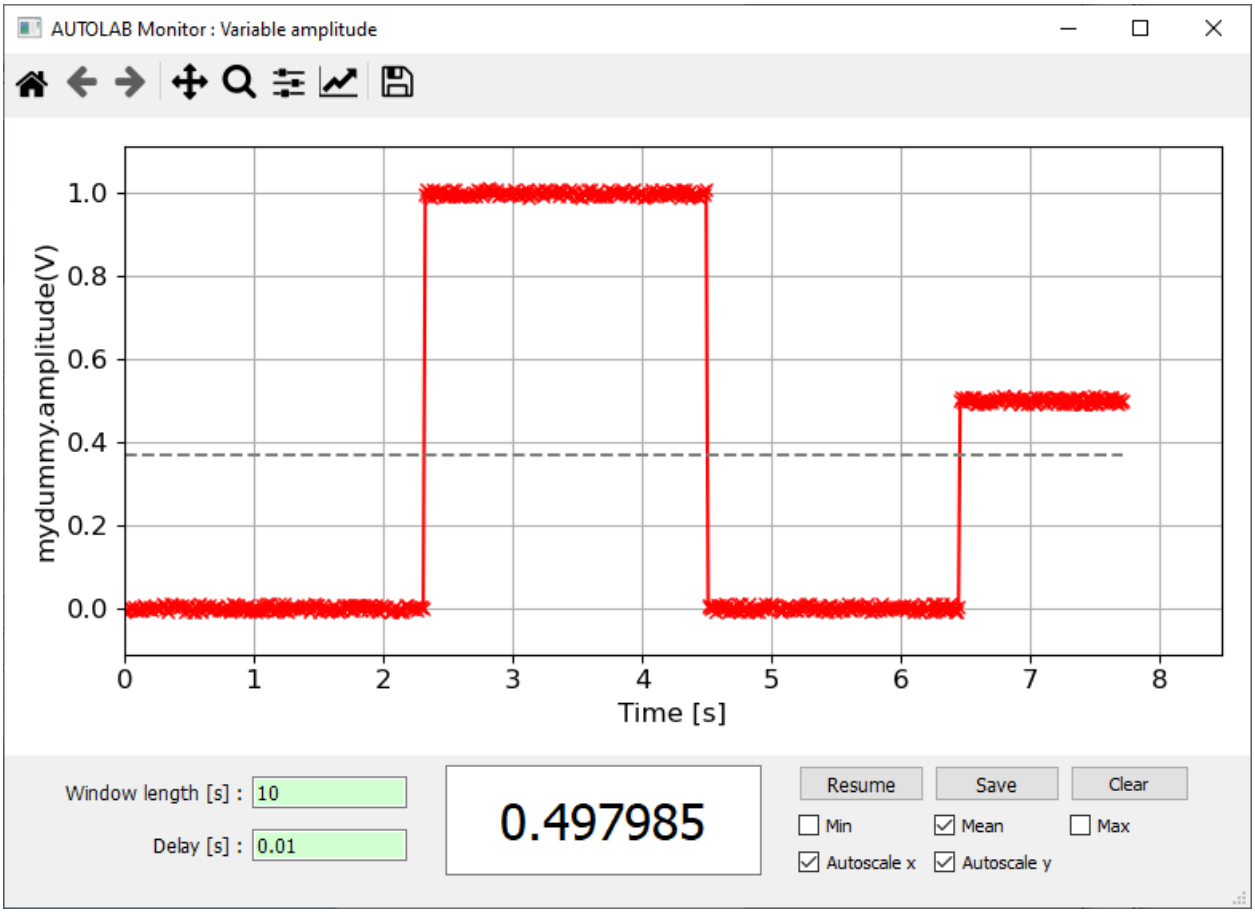
5.1.3 Scanning

You can open the scanning interface with the associated button 'Open scanner' in the menu bar of the control panel window. To configure a scan, please visit the section *Scanning*.

5.1.4 Plotting

You can open the plotting interface with the associated button 'Open plotter' in the menu bar of the control panel window. See section *Plotting*.

5.2 Monitoring



The Autolab GUI Monitoring allows you to monitor a *Variable* in time. To start a monitoring, right click on the desired *Variable* in the control panel, and click **Start monitoring**. This *Variable* has to be readable (read function provided in the driver) and numerical (integer, float value or 1 to 2D array).

In the Monitoring window, you can set the **Window length** in seconds. Any points older than this value is removed. You can also set a **Delay** in seconds, which corresponds to a sleep delay between each measure.

You can pause the monitoring with the **Pause** button, and save the current graph and data with the **Save** button. You will be prompted to give a folder path where the data will be saved.

You can clear the displayed data with the **Clear** button.

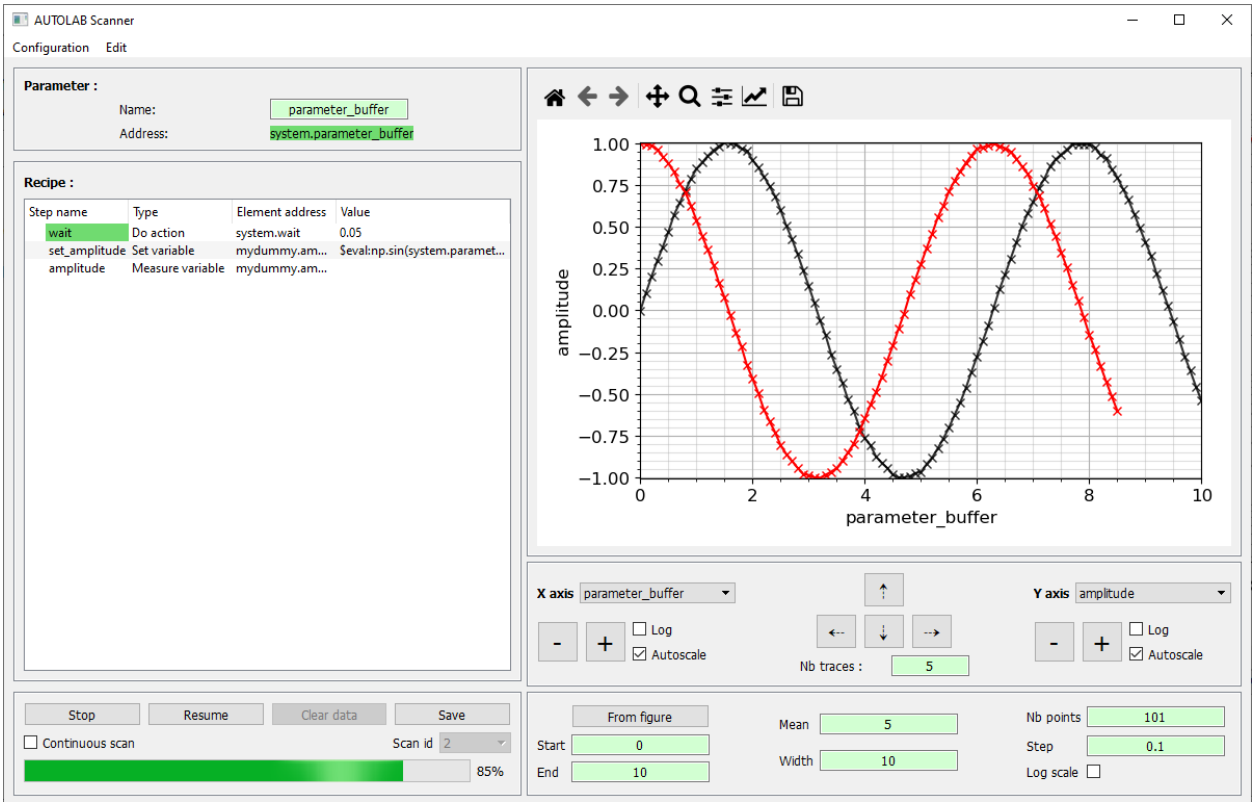
You can display a bar showing the **Min** or **Max** value reached since the beginning of the monitoring. Use the **Clear** button to start back with new min and max value.

The **Mean** option display the mean value of the currently displayed data (not from the beginning).

You can also toggle **autoscaling** of the x and y axis.

5.3 Scanning

The Autolab GUI Scanning interface allows the user to sweep a parameter over a certain range of values, and execute for each of them a custom recipe.



5.3.1 Scan configuration

Parameter

The first step to do is to configure the scan parameter. The parameter is a *Variable* which is writable (write function provided in the driver) and numerical (integer or float value). To set a *Variable* as scan parameter, right click on it on the control panel window, and select **Set as scan parameter**.

The user can change the name of the parameter with the line edit widget. This name will be used is the data files.

Parameter range

The second step is to configure the range of the values that will be applied to the parameter during the scan. The user can set the start value, the end value, the mean value, the range width, the number of points of the scan or the step between two values. The user can also space the points following a log scale by checking the **Log** check box.

Recipe

The third step is to configure the recipe, that will be executed for each value of the parameter. There are three kinds of recipe steps:

- **Measure the value of a Variable**. Right click on the desired *Variable* in the control panel and select **Measure in scan recipe** to append this step to the recipe.
- **Set the value of a Variable**. Right click on the desired *Variable* in the control panel and select **Set value in scan recipe** to append this step to the recipe. The variable must be numerical (integer, float or boolean value). To set the value, right click on the recipe step and click **Set value**. The user can also directly double click on the value to change it.
- **Execute an Action**. Right click on the desired *Action* in the control panel and select **Do in scan recipe** to append this step to the recipe.

Each recipe step must have a unique name. To change the name of a recipe step, right click on it and select **Rename**, or directly double click on the name to change it. This name will be used in the data files.

The recipe steps can be dragged and dropped to modify their relative order. They can also be removed from the recipe using the right click menu **Remove**.

All changes made to the scan configuration are kept in a history allowing changes to be undone or restored using buttons **Undo** and **Redo**. These buttons are accessible using the **Edit** button in the menu bar of the scanner window.

Store the configuration

Once the configuration of a scan is finished, the user can save it locally in a file for future use, by opening the menu **Configuration** and selecting **Export current configuration**. The user will be prompted for a file path in which the current scan configuration (parameter, parameter range, recipe) will be saved.

To load a previously exported scan configuration, open the menu **Configuration** and select **Import configuration**. The user will be prompted for the path of the configuration file.

5.3.2 Scan execution

- **Start** button: start / stop the scan.
- **Pause** button: pause / resume the scan.
- **Continuous scan** check box: if checked, start automatically a new scan when the previous one is finished. The state of this check box can be changed at any time.
- **Clear data** button: delete any previous datapoint recorded.
- **Save** button: save the data of the last scan. The user will be prompted for a folder path, that will be used to save the data and a screenshot of the figure.

Note: The scan configuration cannot be modified or loaded when a scan is started. Stop it first.

Note: During a scan, the background color of each item (parameter or recipe step) indicates its current state. An orange item is being processed, a green one is finished.

5.3.3 Figure

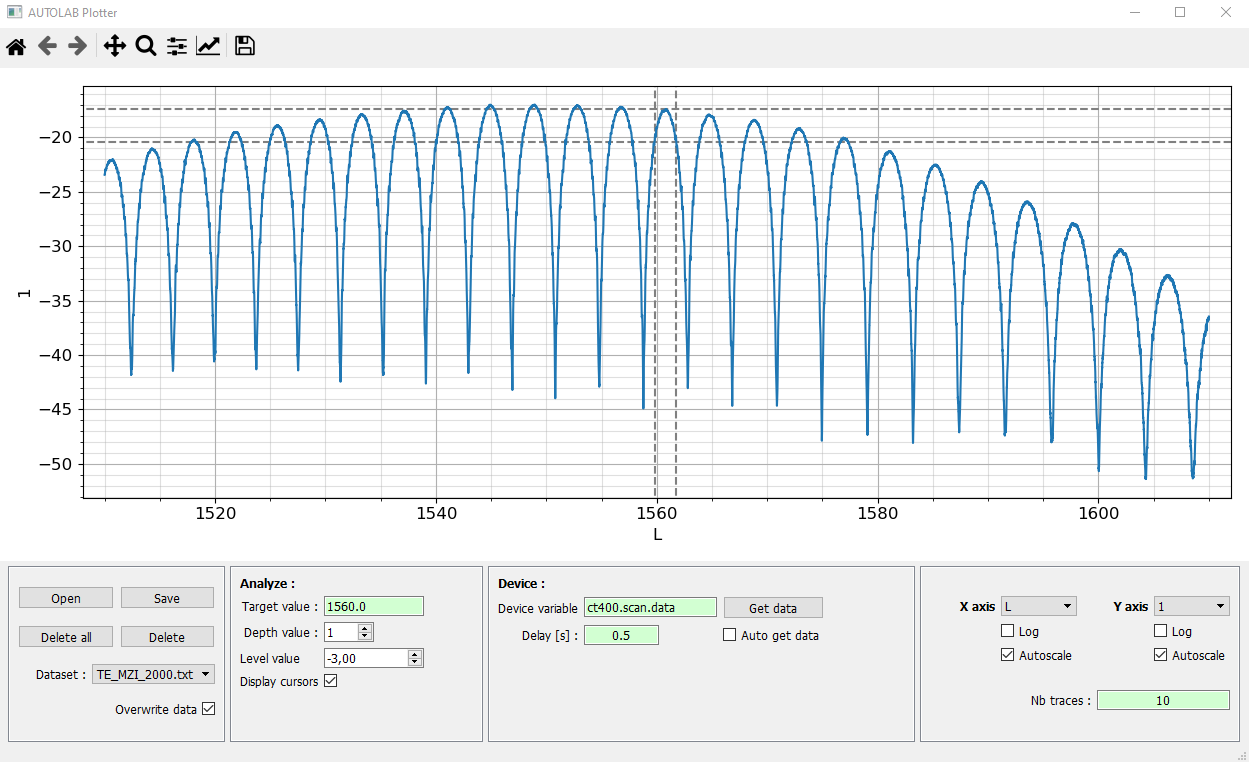
The user can interact with the figure at any time (during a scan or not).

To zoom/unzoom the current view of the figure, use the buttons **Zoom** and **Unzoom** of each axis, or use the navigation bar above the figure. To move the current view of the figure, use the buttons **Left**, **Right**, **Up** and **Down**, or use the navigation bar above the figure.

After a first loop of the scan has been processed, the user can select the **Variable** displayed in x and y axis of the figure. If the *Variable* of the x axis is the parameter, the user can use the button **From figure** to set the current x axis range of the figure, as the new parameter range.

Finally, the user can display the previous curves of a scan by setting the **Traces** number.

5.4 Plotting



The plotter is still in experimental phase at the moment (april 2023). Import data ————

It is currently possible to plot data from previous experiments or any supported data type using the **Open** button.

5.4.1 Device connection

It is also possible to plot a variable from an open device instance with an automatic plot refresh option.

To do this, you need to provide a **device variable**, e.g. `mydummy.array_1D` to create a link between the plotter and the `array_1D` variable of the `mydummy` device (based on the `dummy` driver).

Once the variable is linked, use the **Get data** button to call the variable that returns the array to be plotted (will execute the mydummy.array_1D() command).

To automatically update the plot, check the **Auto get data** option.

5.4.2 Analysis tool

An analysis box can currently be used to display a bandwidth around a local maximum of the displayed curve. Note that a driver named `plotter` copies the analyze class and can be used to access bandwidth information for use in the control panel or directly in a scan recipe.

This tool box could in the future be a way to add custom analysis functions to the plotter.

5.5 Extra function

An experimental function for executing python code directly in the GUI can be used to change a variable based on other device variables or purely mathematical equations.

To use this function both in the control panel and in a scan recipe, use the special `$eval:` tag before defining your code in the corresponding edit box. This name was chosen in reference to the python function eval used to perform the operation and also to be complex enough not to be used by mistake and produce an unexpected result. The eval function only has access to all instantiated devices and to the pandas and numpy packages.

```
>>> # Usefull to have a recipe taking the loop number
>>> $eval:system.parameter_buffer()

>>> # Useful to define a recipe according to a measured data
>>> $eval:laser.wavelength()

>>> # Useful to define the recipe according to an analyzed value
>>> $eval:plotter.analyze.bandwitdh.x_left()
>>> $eval:np.max(mydummy.array_1D())

>>> # Usefull to define a filename which changes during an analysis
>>> $eval:"data_wavelength="+f"{laser.wavelength()}"+".txt"

>>> # Usefull to add a dataframe to a device variable (for example to add data using the␣
↪action plotter.data.add_data)
>>> $eval:mydummy.array_1D()
```

CHAPTER

SIX

OS SHELL

Most of the Autolab functions can also be used directly from a **Windows** or **Linux** terminal without opening explicitely a Python shell.

Just execute the command `autolab` or `autolab -h` or `autolab --help` in your terminal to see the available subcommands.

```
C:\Users\qchat>      autolab
C:\Users\qchat>      autolab -h
Hostname:/home/User$ autolab --help
```

The subcommands are :

- `autolab gui` : a shortcut of the python function autolab.gui() to start the graphical interface of Autolab.
- `autolab driver` : a shortcut of the python interface Driver (see *Command driver*)
- `autolab device` : a shortcut of the python interface Device (see *Command device*)
- `autolab doc` : a shortcut of the python function autolab.doc() to open the present online documentation.
- `autolab report` : a shortcut of the python function autolab.report() to open the present online documentation.
- `autolab infos` : a shortcut of the python function autolab.infos() to list the drivers and the local configurations available on your system.

Table of contents:

The two sections that follow are equivalent for the commands `autolab driver` and `autolab device` (unless specified). They will guide you through **getting basic help** and minimal formatting of command lines (minimal arguments to pass) to **instantiate your instrument** (set up the connection with it, etc.).

6.1 Getting help

Three helps are configured (device or driver may be used equally in the lines bellow):

1) Basic help of the commands autolab driver/device:

```
>>> autolab driver -h
```

   It including arguments and options formatting, definition of the available options and associated help and informations to retrieve the list of available drivers and local configurations (command: autolab infos).

2) Basic help about the particular name driver/device you provided:

```
>>> autolab driver -h -D driver_name
```

It includes the category of the driver/device (e.g. Function generator, Oscilloscope, etc.), a list of the implemented connections (-C option), personnalized usage example (automatically generated from the driver.py file), and examples to use and set up a local configuration using command lines (see *Local configuration* for more informations about local configurations).

3) Full help message **about the driver/device**:

```
>>> autolab driver -D driver_name -C connection -A address -h
>>> autolab device -D nickname -h
```

For driver:

It includes the list of the implemented connections (-C option), the list of the available additional modules (classes **Channel**, **Trace**, **Module_MODEL**, etc.; see *Write your own Driver*), the list of all the methods that are instantiated with the driver (for direct use with the command: autolab driver; see *Command driver*), and an extensive help for the usage of the pre-defined options.

For device:

It includes the hierarchy of the device and all the defined *Modules*, *Variables* and *Actions* (see *Function get_driver_model (in each class but Driver_CONNECTION)* and *Command device* for more informations on the definition and usage respectively).

Note that this help requires the instantiation of your instrument to be done, in other words it requires valid arguments for options -D, -C and -A (that you can get for previous helps) and a working physical link.

6.2 Instantiate a driver/device

The commands autolab driver/device will set up a connection to your instrument, perform the requested operation(s), and finally close properly the connection. To **set up the connection** you need to give valid arguments as requested by the driver (build to suit the physical instrument requirements).

A typical command line structure is:

```
>>> autolab driver -D <driver_name> -C <CONNECTION> -A <address> (optional)
>>> autolab device -D <config_name> (optional)
```

To set up the connection for the first time, we recommand to follow the different help states (see *Getting help*), that usually guide you through filling the arguments corresponding to the above options. To use one of Autolab's driver to drive an instrument you need to provide its name. This is done with the option -D. -D option accepts a driver_name for a driver (e.g. agilent_33220A, etc) and a config_name for a device (nickname as defined in your device_config.ini, e.g. my_agilent). A full list of the available driver names and config names may be found using the command `autolab infos`. Due to Autolab's drivers structure you also need to provide a -C option for the connection type (corresponding to a class to use for the communication, see *Write your own Driver* for more informations) when instantiating your device. The available connection types (arguments for -C option) are driver dependent (you need to provide a valid -D option) and may be access with a second stage help (see *Getting help*). Lately you will need to provide additional options/arguments to set up the communication. One of the most common is the address for which we cannot help much. At this stage you need to make sure of the instrument address/set the address (on the physical instrument) and format it the way that the connection type is expecting it (e.g. for an ethernet connection with address 192.168.0.1 using VISA connection type: `TCPIP::192.168.0.1::INSTR`). You will find in the second stage help

automatically generated example of a minimal command line (as defined in the driver) that should be able to instantiate your instrument (providing you modify arguments to fit your conditions).

Other arguments may be necessary for the driver to work properly. In particular, additional connection argument may be passed through the option -O, such as the port number (for SOCKET connection type), the gpib board index (for GPIB connection) or the path to the dll library (for DLL connection type). In addition, for *complex* instruments (such as instruments with 'slots'), this options provides you with a reliable way to indicate the physical configuration of your instrument [e.g. Module_TEST111 is physically inserted in slot 1, Module_TEST222 is physically inserted in slot 5 (-O slot1=Module_TEST111 slot5=Module_TEST222); see *4 - Additional class (optionnal)* for more informations].

6.3 Command driver

See *Instantiate a driver/device* for more informations about the connection. Once your driver is instantiated you will be able to perform **pre-configured operations** (see *Driver utilities structure (<manufacturer>_<MODEL>_utilities.py file)* for how to configure operations) as well as **raw operations** (-m option). We will discuss both of them here as well as a quick (bash) **scripting example**. In the rest of this sections we will assume that you have a driver (not device) named instrument that needs a connection named CONN.

6.3.1 Usage of pre-configured operations

You may access an extensive driver help, that will particularly **list the pre-defined options**, using:

```
>>> autolab driver -D instrument -C CONN -h
```

It includes the list of the implemented connections, the list of the available additional modules (classes **Channel**, **Trace**, **Module_MODEL**, etc.; see *Write your own Driver*), the list of all the methods that are instantiated with the driver (for direct use with the command: autolab driver; see *Command driver*), and an extensive help for the usage of the pre-defined options. For instance if an option -a has been defined in the file driver_utilities.py (see *Driver utilities structure (<manufacturer>_<MODEL>_utilities.py file)*), one may use it to perform the associated action, say to modify the amplitude, this way:

```
>>> autolab driver -D instrument -C CONN -a 2
```

This modifies the amplitude to 2 Volts (if the unit is set to Volt).

In addition, if the instrument has several channels, an channel option is most likely implemented and one can modify the amplitude of channel 4 and 6 to 2 Volts using:

```
>>> autolab driver -D instrument -C CONN -a 2 -c 4,6
```

Warning: No space must be present within an argument or option (e.g. do not write `- c` or `-c 4, 6`).

Furthermore, several operations may be perform in a single and compact script line. One can modify the amplitude of channel 4 and 6 to 2 Volts and the frequencies (of the same channel) to 50 Hz using:

```
>>> autolab driver -D instrument -C CONN -a 2 -c 4,6 -f 50
```

Note: The arguments are non-positional, which means that the previous line is formally equivalent to:

```
>>> autolab driver -D instrument -C CONN -c 4,6 -f 50 -a 2
>>> autolab driver -D instrument -C CONN -f 50 -a 2 -c 4,6
```

6.3.2 Raw operations (-m option)

Independently of the user definition of options in the file driver_utilities.py, you may access any methods that are instantiated with the driver using the -m option.

Important: This is not a *safe* environment, but it allows you to access all the functionnalities of a driver and doesn't rely on a user configuration.

You may access the **full list of instantiated methods** along with their argument definition, using:

```
>>> autolab driver -D instrument -C CONN -h
```

This allow you to simply copy and paste the method you want to use from the list into the following command, directly as *python code*:

```
>>> autolab driver -D instrument -C CONN -m get_amplitude()
>>> autolab driver -D instrument -C CONN -m set_amplitude(value)
```

One may also call several methods separated with a space after -m option:

```
>>> autolab driver -D instrument -C CONN -m get_amplitude() set_amplitude(2) slot1.get_
↪power()
```

Note: It is possible to combine pre-defined options and -m option in a single script line.

6.3.3 Script example

One may stack in a single file several script line in order to perform custom measurement (modify several control parameters, etc.). This is a bash counterpart to the python scripting example provided there *Script example*.

```
#!/bin/bash                    # Very first line of the file (this is bash code)

i=1                            # Definition of a variable

for volts in $(seq 0 0.1 5)    # Definition of a loop (variable volts goes from 0 to 5␣
↪with steps of 0.1)
do

echo $volts                    # Print the value of the volts variable

autolab driver -D function_generator -C CONN -a $volts  # Increase the amplitude of␣
↪function_generator
autolab driver -D oscilloscope -C CONN -c 1,2,4 -o $i   # Get channels 1, 2 and 4 from␣
↪oscilloscope and save the according files with a name starting with the number of␣
```

(continues on next page)

(continued from previous page)

```
↪iteration of the loop (i)

i=$(($i+1))                    # Increment i variable of 1 at each loop iteration
done                           # End of the for loop
```

Note:

1) Any time the command `autolab driver` is called it sets up the connection. It is then inherently slightly slower (instrument dependant for the amount of time that usually range from 0.1 to 0.5 seconds) than scripting in python.

2) The whole script looks sightly simpler and shorter than its python counterpart.

6.4 Command device

To read, write or save the value of a **Variable**, or to execute an **Action**, use the command `autolab device` in your terminal with the following general format:

```
autolab device -D <CONFIG_NAME> -e <ELEMENT_ADDRESS> <OPTIONS>
```

The **Element address** indicates the address of the desired **Variable** or **Action** in the Autolab Device hierarchy, using a point separator. This command will establish a connection to your instrument, perform the requested operation, and finally close properly the connection. See *Instantiate a driver/device* for more informations about the connection.

The available operations are listed below:

- **To read and print** the value of a readable **Variable** in the terminal, provide its address without any other options:

```
>>> autolab device -D myTunics -e wavelength
1550.00
```

- **To read and save** the value of a readable **Variable** in a file, provide its address with the option `-p` or `--path` with the desired output file or folder path:

```
>>> autolab device -D myPowerMeter -e line1.power -p .\data\power.txt
```

- **To set** the value of a writable **Variable**, provide its address and the option `-v` or `--value` with the desired value:

```
>>> autolab device -D myTunics -e wavelength -v 1551
```

- **To execute** an **Action**, provide its address without any options (or with the option `-v` or `--value` with the desired value if the **Action** has a parameter):

```
>>> autolab device -D myLinearStage -e goHome
```

- **To display the help** of any **Element**, provide its address with the option `-h` or `--help` :

```
>>> autolab device -D myLinearStage -e goHome -h
```

CHAPTER

SEVEN

DOC / REPORTS / STATS

7.1 Documentation

You can open directly this documentation from Python by calling the function `doc` of the package:

```
>>> autolab.doc()
```

```
>>> autolab doc
```

7.2 Bugs & suggestions reports

If you encounter some problems or bugs, or if you have any suggestion to improve this package, or one of its driver, please open an Issue on the GitHub page of this project https://github.com/qcha41/autolab/issues/new

You can also directly call the function `report` of the package, which will open this page in your web browser:

```
>>> autolab.report()
```

```
>>> autolab report
```

Alternatively, you can send an email to the authors (see *About*).

7.3 Statistics of use

At startup, Autolab is configured to send only once a completely anonymous signal (sha256 hashed ID) over internet for statistics of use. This helps the authors to have a better understanding of how the package is used worldwide. No personal data is transmitted during this process. Also, this is done in background, with no impact on the performance of Autolab. You can manage the state of this feature in the local configuration file `autolab_config.ini`. You can use the function `statistics` to know its current state.

```
>>> autolab.statistics()
```

CHAPTER

EIGHT

ABOUT

This Python package has been created in 2019 by Quentin Chateiller (PhD student) and Bruno Garbin (post-doc researcher) from the ToniQ team of the C2N-CNRS laboratory (Center for Nanosciences and Nanotechnologies, Palaiseau, France).

The first developpements of the core, the GUI, and the drivers started initially in 2017 by Quentin. Bruno arrived in the team in 2019, providing a new set of Python drivers from its previous laboratory. In order to propose a Python alternative for the automation of scientific experiments in our research team, we finally merged our works in a Python package based on a standardized and robust driver architecture, that makes drivers easy to use and to write by the community.

Thanks to Maxime, Giuseppe and Guilhem for their contributions.

You find this package useful? We would be really grateful if you could help us to improve its visibility ! You can:

- Add a star on the GitHub page of this project
- Spread the word around you
- Mention this package in your research publications

Contacts: quentin.chateiller@c2n.upsaclay.fr, bruno.garbin@c2n.upsaclay.fr

Last edit: Apr 15, 2023 for the version 1.1.12