UNIVERSITY OF TWENTE.

Quantum Transport in Matter
Interfaces and Correlated Electron systems

QTM measurement toolbox

Documentation on a Python framework for measurements

Authors
D.H. Wielens, PhD

Faculty of Science and Technology

QTM Measurement toolbox
Version 2.6.0 - March, 2025

University of Twente
Faculty of Science and Technology

D.H. (Daan) Wielens, PhD
d.h.wielens@utwente.nl

This document has been composed using the LaTeXtypesetting system.

Contents

Chapter 1

Introduction

The toolbox presented in this document has been developed in order to perform measurements in Python. The introduction of a measurement toolbox in this language is a natural step that follows from the bachelor programme of Applied Physics at our university, where Python was introduced as main programming language in 2018.

1.1 Requirements

We assume that Python has been installed through Anaconda. The following software and packages are required:

- Python 3.x
- pyVISA package
- pySerial package
- pyqtgraph package

If Anaconda is installed for all users (typically in `C:\ProgramData\Anaconda3`), make sure to install the packages using the Anaconda Prompt with elevated user rights (in the Windows Start Menu, right click on the Anaconda Prompt software and choose 'Run as administrator'. If it is installed only for you (typically in `C:\Users\<username>\Anaconda3`) you can start the Anaconda Prompt normally.

The packages can be installed by typing the following into the Anaconda prompt:

```
conda install -c conda-forge pyvisa
conda install -c anaconda pyserial
conda install -c anaconda pyqtgraph
```

To use the live plotting tool as a standalone program (meaning that you can just double-click on a desktop icon to launch it), make sure that the Python PATH has been added to the System's Environment Variables. For that, open the Control Panel, in the search box type Environment, click the option 'Edit the system environment variables'. In the pop-up window, choose 'Environment Variables...'. In the System variables section, double-click on the 'Path' variable. A list of file paths appears. Add the following lines to the list of paths:

```
C:\ProgramData\Anaconda3\
C:\ProgramData\Anaconda3\Library\mingw-w64\bin
C:\ProgramData\Anaconda3\Library\usr\bin
C:\ProgramData\Anaconda3\Library\bin
C:\ProgramData\Anaconda3\Scripts
```

Finally, when you double-click the 'QTMplot.pyw' program in the QTMtoolbox folder, Windows may ask with what software to open the file. Choose 'More', then at the bottom 'Select another app from this PC'. Browse to the installation folder of Anaconda and select `pythonw.exe`.

Chapter 2

Structure of the QTM toolbox

The QTM toolbox consists of various scripts that interact with each other. Here, we clarify the structure of the software.

2.1 Program structure

Let's start with the different folders. In the *root directory*[1], one finds:

- The file `Measurement_script.py`. This is the main program that the user executes. The main program sets up connections to all specified GPIB devices and loads the required modules to perform measurements. Depending on the user, the main program may also contain a list of measurement commands.
- The folder `functions`. This folder contains Python scripts that hold all functions that can be used during the measurements. Some examples are `move`, `sweep` and `measure`. The functions are all stored in a Python file called `qtmlab.py`. If a user wants to create custom functions which would be irrelevant to others, he/she might create a separate file in this folder and add his/her functions there.
- The folder `instruments`. This folder contains definitions for each instrument that can be used for measurements. Essentialy, these files tell Python how to convert our "simple" commands to the actual commands that are sent through GPIB. More specifically, the scripts are wrappers for the pyVISA module, where the most used commands of each device have been implemented[2].
- The folder `icons`. This folder contains icons used by the plot software.
- The folder `images`. This folder contains images used in the online documentation of GitHub.

[1] The root directory is the main project folder. All other paths mentioned are relative to this folder.

[2] Naming conventions originate from previous measurement software, and might therefore be inconsistent / unexpected. We chose to continue with these names to make the transition as smooth as possible for our group members.

Chapter 3

Starting and preparing the toolbox

Before measurements can be performed, the toolbox must be initialised. In this chapter, we walk through the steps necessary to complete this process. We assume that Python 3.x is installed through Anaconda, and furthermore that required packages (pyVISA, pySerial) are installed through the Anaconda prompt. Experienced Python users can skip the first two sections of this chapter.

3.1 Launching Spyder

The easiest way to perform measurments is throughout the software **Spyder**. Spyder is an IDE (Integrated Development Environment). The layout is similar to the MATLAB software. Sypder can be found in the application menu of the measurement computer, in the 'Anaconda3' folder.

When launched, a window like Fig. 3.1 appears. The window consists of three smaller windows:

1. The **Editor**. The editor (shown at the left side of the program) enables the user to write scripts and edit existing scripts. A script can be executed by pressing the green 'Play' button in the menu bar, or by pressing `<F5>` on the keyboard.
2. The **Variable explorer** is shown at the top right panel. Here, used variables are presented. Note that not all variables (for example, instruments that we define) will be shown here. With the tabs at the bottom of the panel, one can also bring up the **File explorer**, **Help** or the **Profiler**.
3. The **IPython console** is the place where one can type code that will be executed when `<Enter>` is pressed. This resembles the 'Command window' of MATLAB.

3.2 Open the toolbox and change the working directory

The next step is to open the toolbox. For this, click the 'Open' button or type `<Ctrl>+<O>` and browse to the `Measurement_script.py` file, then open it.

Just as MATLAB, Python has a *working directory*. It can only find functions, modules and files within its installation folder or this specific working directory. Our program is located in a different folder, and although the default working directory can be chosen to be our toolbox folder, it might be

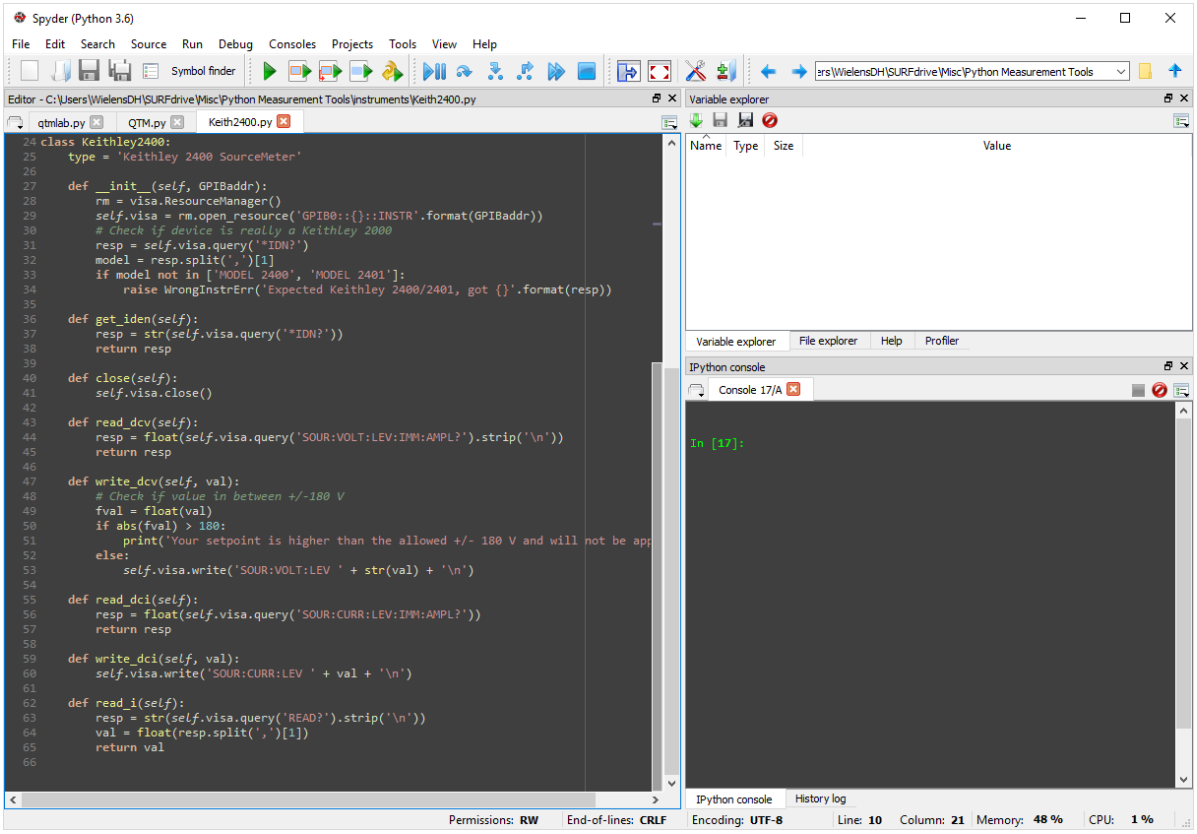

Figure 3.1: The user interface of Sypder. The **editor** is placed at the left side of the window. The right side is divided into two different regions, the **Variable explorer** (or **File explorer**, **Help** or **Profiler** window) and the **IPython console**

.

that one has to adjust it manually.

To change the working directory, click on the 'Folder' icon next to the textbox that contains the path of the current working directory in the top right corner of the window. Choose the folder that contains the `Measurement_script.py` file.

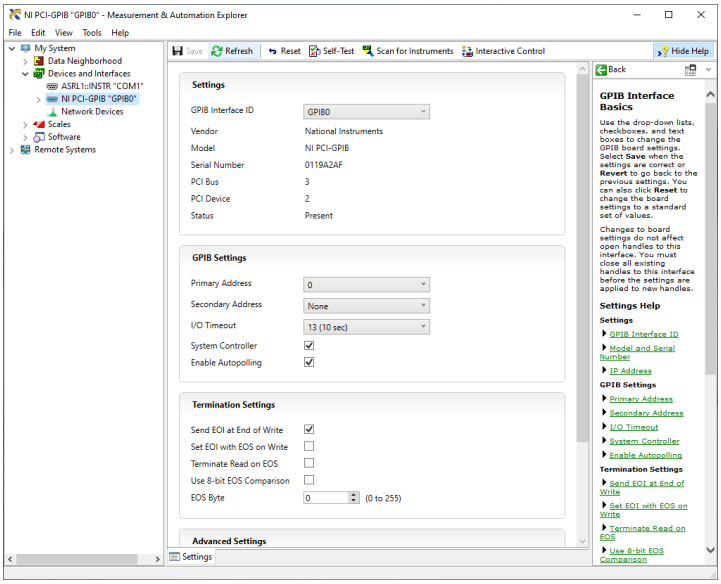

Figure 3.2: Changing the GPIB Interface ID in NI-MAX

3.3 Prepare your GPIB Controller

NOTE: Only read this section if you are setting up a new PC for measurements.

Although we usually think of GPIB addresses as a number, the address in fact is more complicated. A typical address is of the form `GPIB0::15::INSTR`. Here, `15` would be the 'address' that we usually talk about. If your computer has multiple GPIB communication devices (USB-GPIB adapter, GPIB PCI cards, etc.) every card will have its unique number. This number determines the prefix of the address (so, in our example, `GPIB0`).

The QTMtoolbox is written in such a way that users only have to enter the GPIB 'number' (`15`) instead of the string as a whole. Unfortunately, for this we have to assume that the controller's address is `GPIB0`. If it's not, the toolbox will not function correctly. One can do the following things to fix it:

- Change the address of the controller. To do so, open NI-MAX, go to **Devices and Interfaces** in the menu at the left, select your GPIB interface and change its **GPIB Interface ID** to `GPIB0`.
- If you can't change the GPIB interface ID because other equipment relies on that specific ID, you can replace every `GPIB0` by your address in the instrument source code files.

3.4 Prepare the toolbox

In this section, we prepare the toolbox by telling it what instruments we will use and which GPIB addresses are used by these instruments. Then, we tell our toolbox what values of instruments should be recorded during measurements. Finally, we run our - just modified - `Measurement_script.py` file to initialise our system.

3.4.1 Adding instruments

To add instruments, we modify the `Measurement_script.py` file. In the **Setup** section, we first add all different *types* of instruments that will be used[1].

For example, if we want to use two Keithley 2400 SourceMeters, a Keithley 2000 Multimeter and a LakeShore 332 Temperature controller, we modify the `# Import device definitions` section as follows:

```
from instruments.Keithley2400 import *
from instruments.Keithley2000 import *
from instruments.Lake332 import *
```

Importing the modules means that Python adds the modules to its *namespace*, which essentially means that all functions within these modules can be accessed and used from now on.

3.4.2 Connecting to instruments

Now that the definitions for the instruments have been imported, we can bind instruments. For every instrument, we give it a name and then use the corresponding *Class* from the imported module. As argument, we specify the GPIB address. The general syntax is of the form

```
<devicename> = <DeviceClass>(<GPIBaddress>)
```

For our example, we would add the following lines of code to the `# Connect to devices` section:

```
keithBG = Keithley2400(20)
keithTG = Keithley2400(22)
lake = Lake332(12)
```

3.4.3 Define waiting time

If a variable of a device is moved (i.e. during a sweep), you may want to let the system/sample stabilize before a measurement is taken. The time (in seconds) it takes before a data point is recorded is called `dtw` and is defined in the measurement file as follows:

[1] This means that if you use 4 sr830 lock-in amplifiers, you only have to import the corresponding instrument module once

```
# Define wait time between 'reaching setpoint' and 'taking measurement' (in seconds)
dtw = 3
```

3.4.4 Define sample name

For automatic data storage purposes (such as the university's data management plan), the data will be stored in a subfolder that is based on your sample name and the day you loaded the sample in the measurement equipment. The QTMtoolbox software will take care of creating this subfolder and placing the data in the correct folder, but the sample name and the loading date need to be provided in the script as follows:

```
# Define sample name
samplename = '2021-12-14_Test-sample'
```

Note: if the timestamp is missing or incorrect, the script will throw an error and does not execute the measurement. Please **do not use spaces in the filename**.

3.4.5 Define measurement variables

In this step, we tell the toolbox what variables of which devices will be recorded during a measurement. All variables are stored into a single *string*[2], called `meas_list`. The structure of the string must be as follows:

```
meas_dict = 'keithBG.dcv, keithBG.i, keithTG.dcv, KeithTG.i, lake.temp'
```

The list contains comma-separated variables of devices that we want to record during our measurement. We supply the name of the device (must be the name as specified above when we connected to the device), then a period (.) and then the variable that we want to measure.

To see what variables can be measured, either look further into this manual (chapter 6) or manually open the files that define the instruments (so, all files within the `instruments` folder). Every variable that starts with `read_` can be measured.

NOTE: We do *NOT* specify the `read_` prefix in the `meas_list` . The script will take care of this automatically.

Tip! If your list is very long, you can extend it over multiple lines in your code:

[2]In a previous version of the software, we used a dictionary for this. The dictionary was cumbersome, but easier for the software itself. Nowadays, the dictionary is still used, but it is automatically generated from the provided list. But, if you have old versions of a measurement script that only contain the dictionary, you can use that script with the new software as well.

```
meas_list = 'trit.temp5, trit.temp8, trit.temp9, sr1.auto_x,'\
            'sr1.y, sr1.r, sr1.amp, sr1.freq,'
```

3.4.6 Loading the QTMtoolbox code

The last part of the code makes sure that you can access the functions of the toolbox (i.e. move, sweep, measure, etc.). It furthermore saves a snapshot of your measurement script in your data folder as well, so that the exact code used to perform the measurement is archived too. These lines of code are necessary, but should not be altered:

```
# Import measurement tools
from functions import qtmlab
from functions import qtmstartup
meas_dict = qtmlab.generate_meas_dict(globals(), meas_list)
qtmlab.meas_dict = meas_dict
qtmlab.dtw = dtw
qtmlab.samplename = samplename
#%% Batch commands
qtmstartup.copy_script(__file__, samplename)
"""
Add your batch commands below this comment. Alternatively, run commands individually
from the IPython console
"""
```

3.5 Start the toolbox

With that, our setup process is complete. Run the `Measurement_script.py` file by clicking on the Run button (green Play button) or by pressing `<F5>`.

3.6 Template: a standard measurement script

The code below is a basic measurement script that one can run without any GPIB equipment, to test whether the toolbox works correctly.

```
# -*- coding: utf-8 -*-
#%% Setup
import numpy as np

# Import device definitions
from instruments.curtime import *

# Connect to devices
ct = curtime()

# Define wait time between 'reaching setpoint' and 'taking measurement' (in seconds)
dtw = 3

# Define sample name
samplename = '2021-12-14_Test-sample'

# Define what variables need to be measured
meas_list = 'ct.time, ct.timems'

# Import measurement tools
from functions import qtmlab
from functions import qtmstartup
meas_dict = qtmlab.generate_meas_dict(globals(), meas_list)
qtmlab.meas_dict = meas_dict
qtmlab.dtw = dtw
qtmlab.samplename = samplename
#%% Batch commands
qtmstartup.copy_script(__file__, samplename)
"""
Add your batch commands below this comment. Alternatively, run commands individually
from the IPython console
"""

qtmlab.record(1, 10, 'test.csv')
```

Chapter 4

Executing commands and measurements

With our toolbox ready, we can execute commands to change setpoints, to get the current value of a device, and to perform measurements.

4.1 Individual commands

To run individual commands, head over to the **IPython console**. From the IPython console, you can either directly control devices or issue move, sweep, ... commands.

Some commands (such as readings and measurements) return values, some commands do not. To see whether a command is finished, simply wait until the IPython console presents a new `In [<i>]` statement.

Typical commands that can be issued - after preparing and executing our `Measurement_script.py` - from the console are

```
In [1]: keithTG.read_dcv()
Out[1]: 0.0

In [2]: keithBG.read_i()
Out[2]: -1.94671e-11

In [3]: keithBG.write_dcv(2.1)

In [4]: qtmlab.move(keithBG, 'dcv', 0, 0.5)
```

Note that we can directly interact with devices, but that functions from `qtmlab` have the module name as prefix for the command. This is because of the way how we imported the modules, which, in turn, is necessary for the toolbox to work as we intended.

4.2 Batch file

If you want to perform multiple measurements in a sequence, you can program them below the `#%% Batch commands` header in the `Measurement_script.py`. Simply enter all commands that must be executed sequentially in the file and run the code to start your batch file[1]!

Keep in mind that you are by no means limited to the instruments and functions of the QTMtoolbox here! You can define your own for-loops, functions, classes, etc. and use them in the script to measure custom things.

4.3 Plotting your data

A new feature (introduced in April, 2020) is the capability of visualising the data. In the root folder of the QTMtoolbox you'll find `QTMplot.pyw` which can be launched to plot the data. Please note the following.

NOTE: to keep the code in `qtmlab.py` as clean as possible (for future updates) we have implemented the plot tool as a standalone script. To run this script, double-click on the file[2]. Windows will ask with what program you want to open the file.
Select `C:\ProgramData\Anaconda3\pythonw.exe` (if that is the location of your Python distribution).

The GUI is shown in Fig. 4.1. At the top, the user can toggle the "live" feature on/off. When on, the script monitors the `Data` folder (this folder will be generated by the first measurement) and automatically plots the data file with the most recent timestamp. It will automatically update the plot too. If the toggle is off, the plot will not update. Furthermore, one can click on the folder icon at the top to load a data file. This will automatically stop the live plot.

Once a file is shown (either live or manually opened), one can choose the variables for the x, y-axes at the bottom. The plot and labels will update automatically. The textbox at the top simply displays the filename of the file that is currently being displayed in the plot.

To zoom in, right-click and hold. Drag up/down to zoom in/out vertically, and drag left/right to zoom in/out horizontally, respectively. To move the plot around, left-click and hold. To access PyQtGraph's plot options, simply right-click to pop-up a context menu with options.

Life has been made easier by introducing some shortcuts within the plotting app:

- `Ctrl+O` : open a file for plotting (same as clicking on the folder icon)
- `Ctrl+L` : toggle live plotting on/off
- `Ctrl+R` : rescale the plot *once* (does not update axes automatically afterwards when using live plotting)
- `Ctrl+A` : enable automatric rescaling (does update axes automatically when using live plotting)

[1] Note that it might be useful to add some `print` commands to tell yourself what measurement is running / finished.
[2] For this to work, Python should be added to the System PATH. If not, try to run the file from a separate console in Spyder.

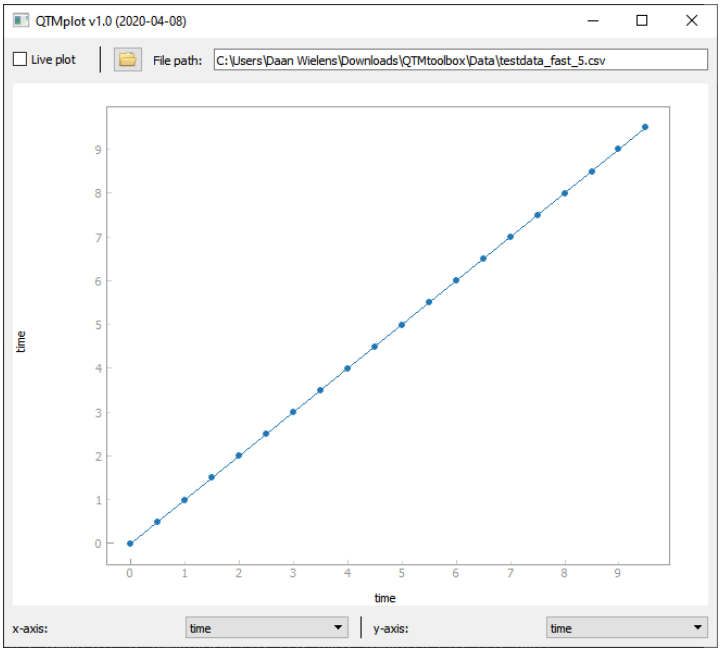

Figure 4.1: GUI of QTMplot.

Chapter 5

List of functions

In this section all functions within the `qtmlab` module will be listed.

5.1 move

The move command can be used to *move* a variable (setpoint) to a specific value at a specified rate.

The general syntax is

```
qtmlab.move(<device>, '<variable>', <setpoint>, <rate>)
```

An example would be

```
qtmlab.move(keithBG, 'dcv', 10, 0.5)
```

The function accepts the following arguments:

- `<device>` : the device identifier. Should be the variable name as defined in `QTM.py`.
- `<variable>` : the variable that will be moved. Note that a variable can only be moved if it contains both a *read* and *write* function in the instrument definition. **The <variable> does not need a `read` or `write` prefix here!**
- `<setpoint>` : the setpoint that the device will move to. Must be provided as `float`.
- `<rate>` : the rate at which the device will move to the setpoint. Must be provided as `float`.

5.2 measure

The measure command can be used to acquire data of all devices that are specified in the measurement dictionary `meas_dict` in the `QTM.py` file.

The general syntax is

```
qtmlab.measure()
```

The function will return a `NumPy array` containing all the values of all variables.

The function does not require any inputs. However, it may look that `md` must be passed as argument. This is automatically done in the function definitions and when the `qtmlab` module is imported.

5.3 sweep

The sweep command can be used to measure datapoints while one device is constantly changing its setpoint in between the measurements. The device that is being swept will *move* to a point, *measure* data, *move* to the next point, etc.

Note: please be aware of the fact that although the first column will store the variable that is swept, this is only the *setpoint* of this variable and hence the actual value *can be different*[1]! Therefore, we suggest one to always also include the swept variable into the measurement directory so that the actual values are always retrievable.

The general syntax is

```
qtmlab.sweep(<device>, '<variable>', <start>, <stop>, <rate>, <npoints>, '<filename>',
    sweepdev='<sweepdev>', scale='<scale>')
```

An example would be

```
qtmlab.sweep(keithBG, 'dcv', -10, 10, 0.5, 21, 'Gatesweep.csv', sweepdev='Vbg(V)', scale
    ='lin')
```

The function accepts the following arguments:

- `<device>` : the device identifier. Should be the variable name as defined in `QTM.py`.
- `<variable>` : the variable that will be moved. Note that a variable can only be moved if it contains both a *read* and *write* function in the instrument definition. **The <variable> does not need a `read` or `write` prefix here!**
- `<start>` : the starting point of the variable that is swept. Must be provided as `float`.
- `<stop>` : the end point of the variable that is swept. Must be provided as `float`.
- `<rate>` : the rate at which the device will move to the setpoint. Must be provided as `float`.
- `<npoints>` : the number of datapoints that will be acquired during the sweep. Must be provided as `int`.
- `<filename>` : the filename of the dataset that will be saved.
- `<sweepdev>` : the name specified here will serve as the name of the swept variable in the header of the data file.
- `<scale>` : **OPTIONAL** The user can choose whether the set of datapoints between the 'start' and 'stop' values will be linearly spaced (`scale='lin'`) or logarithmically spaced (`scale='log'`). Note that for the latter the sweep rate

[1] An example is when one sets the amplitude of the sr830 lock-in amplifier to zero. Although the setpoint is zero, the actual value can never be lower than 4 mV.

is still the same for all data points, so that the time between two successive data points increases exponentially. When not specified, `'lin'` will be used. A third option, `scale='IVVI'`, will be addressed below.

Using the `scale='IVVI'` option:
When the user sweeps the DAC voltages of the IVVI rack, it is important to realize that the DAC has a finite resolution of 16-bit over the entire voltage range (for the *bipolar* mode, this is [-2 V, 2 V]). This means that the full range of 4 V is divided by $2^{16} = 65536$ values. The step size is thus $4\ \mathrm{V}/(2^{16} - 1) = 61.04\ \mu\mathrm{V}$. If you now try to sweep the DAC with a step size of 100 uV, it means that the DAC needs to jump 120 uV for a typical step, which is too much. So, sometimes the DAC will jump only 60 uV to 'keep up' with the setpoint. Hence, your step size is no longer uniform, and if you then perform calculations on the measured data set (where the DAC setpoints are used as x), you may get unexpected peaks in your data. To illustrate the importance of *DAC step syncing* (i.e. using

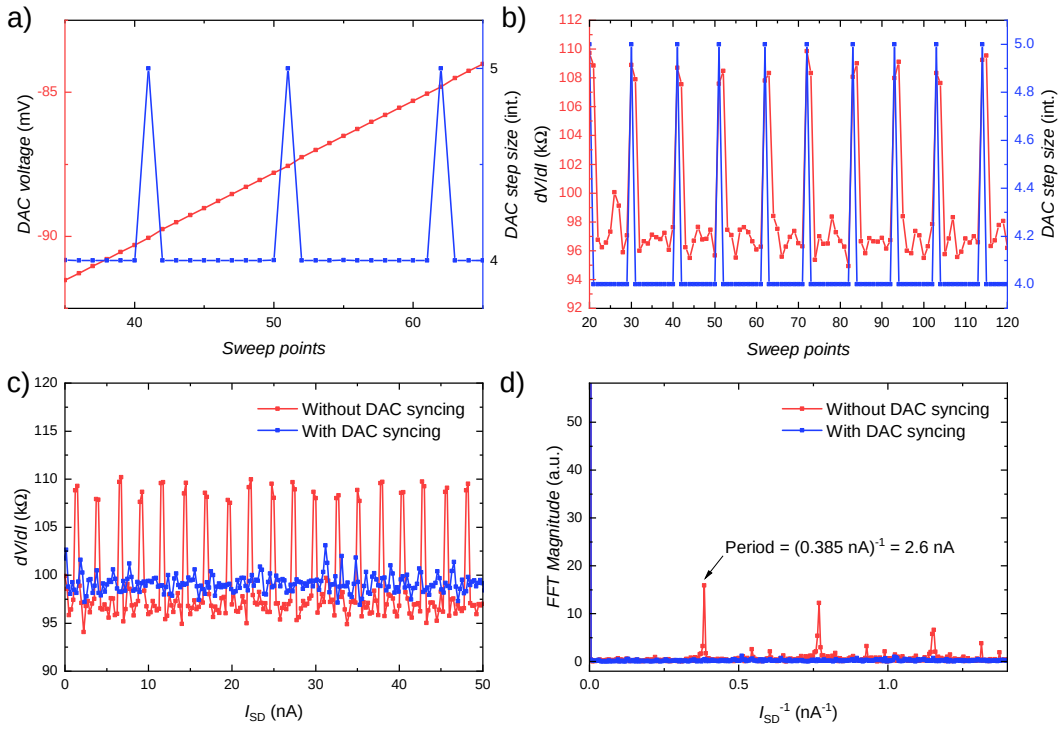

Figure 5.1: **DAC syncing.** See main text for explanation of the panels.

the `scale='IVVI'` option) becomes clear in the figure above. In panel **a)** we sweep the DAC voltage from -0.1 V to 0.1 V in 801 steps. The figure shows a zoom of part of the sweep. The left axis (red curve) denotes the *measured* output voltage, the right axis (blue curve) denotes the amount of DAC steps taken for each sweep point (i.e. $\Delta V/61.04 \cdot 10^{-6}$). If you look carefully at the red curve, you'll see a kink for every spike in the blue curve. Here, the DAC needs to increase with a different number of steps to keep up with the setpoint.
Let's see what the consequences are for an experiment. In panel **b)** we plot the analyzed results of the same data set as in **a)**. We used the DAC to source a voltage, converted it with an S4c module to a

current, fed this current into the R1 sample simulator and measured the voltage across the simulator with an M2d module and a Keithley multimeter. Then, we numerically differentiated dV/dI to compute the resistance. It is evident that for every jump in DAC step size, the resistance spikes as well.
In panel **c)**, we compare the experiment without DAC syncing (`scale='lin'`) to the experiment with DAC syncing (`scale='IVVI'`). Here, we clearly see that DAC syncing removes the artificial spikes in the data and gives a better value for the measured resistance (set to 100 kΩ for this experiment). Furthermore, in panel **d)** we show that while the non-synced data shows a clear periodic trend, the synced data has no clear period and thus possesses no artifacts anymore.

Additional note on the `scale='IVVI'` option:
In the examples above, we use the *DAC setpoint* as x value when we take the numerical derivative of our data. If the DAC voltages are also recorded during the measurement, we can also try to use the *measured DAC* as x values for our calculations. The figure below shows the results of the numerical derivation for the different scenarios. Here, we clearly see that we could omit DAC syncing if we also measure the value of the DAC, and then use this measured DAC voltage for our derivation.

So, should I use DAC syncing?
In conclusion, both methods (using the DAC syncing or using the measured DAC voltages) gives the same result. Then why bother with DAC syncing? Well, if you perform a 2D map scan you'd like to have a regular grid of X and Y values (provided you measure $Z(X, Y)$ in this example). If you use the measured DAC steps, the grid will likely not be regular, while using the synced voltages will give a regular grid.

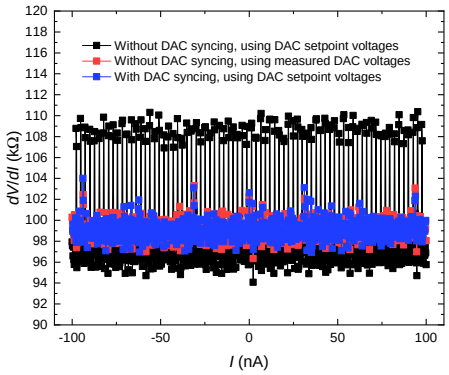

Figure 5.2: **Comparison in use of DAC measurements or DAC setpoints.**

5.4 waitfor

The waitfor command can be used to wait until a certain parameter has reached a certain value (within a specified margin) for a specified timespan. Thus, the waitfor command can be used to 'pause' the script and wait for parameters (mostly temperatures) to stabilise.
Note however that when the waitfor command releases its 'pause' there is no guarantee anymore that the parameter will remain stable afterwards. You should choose appropriate PID settings yourself for the parameter you want to stabilise.

The general syntax is

```
qtmlab.waitfor(<device>, '<variable>', <setpoint>, <threshold>, <tmin>)
```

An example would be

```
qtmlab.waitfor(lake, 'temp', 4.5, 0.005, 120)
```

The function accepts the following arguments:

- `<device>` : the device identifier. Should be the variable name as defined in `QTM.py`.
- `<variable>` : the variable that will be monitored until it is stable.
  Note that a variable can only be stabilised if it contains has a *read* and function in the instrument definition.
  The <variable> does not need a `read` or `write` prefix here!
- `<threshold>` : the threshold for the variable being stable. The variable x is regarded as stable within threshold t when $|x - x_{\text{setpoint}}| \leq t$. Must be provided as `float`. Default value is 0.05.
- `<tmin>` : the minimum time for which the variable must stay within the threshold in seconds. Must be provided as `float`. Default value is 60.

5.5 record

The record command can be used to perform measurements at a specified interval for a specified amount of data points.
Note that you can always use `<Ctrl+C>` *in the Console to quit the record command.*

The general syntax is

```
qtmlab.record(<dt>, <npoints>, '<filename>')
```

An example would be

```
qtmlab.record(10, 20000, 'cooldown.csv')
```

The function accepts the following arguments:

- `<dt>` : timestep between successive measurements in seconds.
  Must be provided as `float`.
- `<npoints>` : number of datapoints that will be taken.
  Must be provided as `float`.
- `<filename>` : the filename of the dataset that will be saved.

5.6 record_until

The record_until command can be used to perform measurements at a specified interval. The amount of data points is determined by a condition, which can be set by user. The condition can either be 'measured value must be smaller, larger or be equal to the given value'.
Note that you can always use `<Ctrl+C>` *in the Console to quit the record command. Furthermore, note that the 'equal to' condition requires that the measured value and given value match exactly. There is no margin, like for the* `waitfor` *command.*

The general syntax is

```
qtmlab.record(<dt>, '<filename>', <device>, '<variable>', '<operator>', '<value>', '<
    maxnpoints>')
```

An example would be

```
qtmlab.record(10, 'cooldown.csv', ppms, 'temp', '<', 2, '50000')
```

In this example, the script will record measurements with an interval of 10 seconds between each measurement. The measurements will continue until the temperature of the PPMS will fulfill $T < 2$ K, *or* when 50000 data points have been taken.

The function accepts the following arguments:

- `<dt>` : timestep between successive measurements in seconds. Must be provided as `float`.
- `<filename>` : the filename of the dataset that will be saved.
- `<device>` : the device for which a variable will be checked.
- `<variable>` : the variable that will be checked.
- `<operator>` : the operator associated with the condition. Options are:
- `['larger', '>']`
- `['smaller', '<']`
- `['equal', '=', '==']`
- `<value>` : the value that is used for the condition.
- `<maxnpoints>` : the maximum number of data points that will be recorded, in case the condition is never met.

5.7 multisweep

The multisweep command is an extension of the sweep command, where multiple devices can be swept simultaneously. Just like with a regular sweep, the script will first move - in this case all devices provided - the device(s) to their setpoint(s), after which a single measurement is taken.

The general syntax is

```
qtmlab.multisweep(<sweeplist>, <npoints>, '<filename>')
```

Here, sweeplist contains separate lists for every device that one wants to sweep, i.e.

```
sweeplist = [
        [<device1>, <variable1>, <start1>, <stop1>, <rate1>, <sweepdev1>],
        [<device2>, <variable2>, <start2>, <stop2>, <rate2>, <sweepdev2>],
        ....
      ]
```

The meaning of all entries in the list are explained in *sweep* section, so we will not introduce them again.
An example would be

```
sweeplist = [
        [sr1, 'dac1', 0, 1, 0.2, 'sr1.dac1'],
        [sr1, 'dac2', 0, 2, 0.2, 'sr1.dac2']
      ]

qtmlab.multisweep(sweeplist, 21, 'multisweep.csv')
```

5.8 megasweep

The megasweep command can be used to sweep two variables. Say that variable z must be measured for certain x and y, then a megasweep will measure $z(x, y)$ for the two linear spaces that span x and y. It is important to realise that there are different ways in which a megasweep can be performed.

The most basic mode is the 'standard' mode. Fig. 5.3 shows how the standard mode measures the map $z(x, y)$. In this example, we have chosen for $x = np.linspace(0, 6, 7)$ and $y = np.linspace(0, 3, 4)$. We start at the green dot and measure $z(0, 0)$. Then, we follow the red line, which denotes a *sweep* being performed. Hence, we measure $z(1, 0)$, $z(2, 0), \ldots, z(6, 0)$. Then, we *move* x back to its first value (0) and *move* y to its second value (1). This is denoted by the grey arrow. Now, we sweep x for $y = 1$, move, sweep, etc., until we end up at the blue dot.

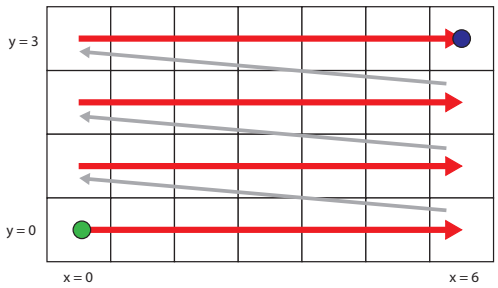

Figure 5.3: The 'standard' mode.

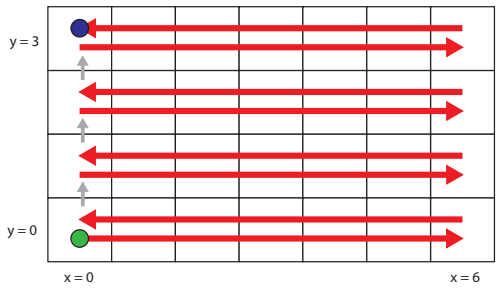

Figure 5.4: The 'updown' mode.

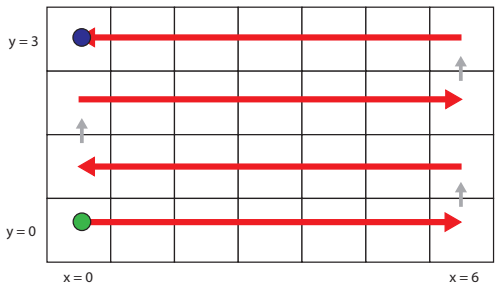

Figure 5.5: The 'serpentine' mode.

The general syntax is

```
qtmlab.megasweep(<device1>, '<variable1>', <start1>, <stop1>, <rate1>, <npoints1>, <
    device2>, '<variable2>', <start2>, <stop2>, <rate2>, <npoints2>, 'filename',
    sweepdev1='<sweepdev1>', sweepdev2='<sweepdev2>', mode='<mode>', scale='<scale>')
```

An example would be

```
qtmlab.megasweep(oxM, 'fvalue', -7, 7, 0.005, 1401, sr1, 'amp', 0, 1, 0.05, 11, '
    Megasweep.csv', sweepdev1='oxM.fvalue', sweepdev2='sr1.amp', mode='updown')
```

As most of the arguments have been addressed in the *sweep* section, we will not list them again but refer to reader to that specific section. The only new option is

- `<mode>` : **OPTIONAL** sweep mode, as explained above.
  Options are: `standard`, `updown`, `serpentine`.
  When not specified, `'standard'` will be used.

Note: the scale option enables one to use DAC syncing (see the *sweep* section for more details on this procedure) on the **fast axis** (variable 2).

5.9 multimegasweep

The multimegasweep command is a combination of the multisweep and the megasweep - one can perform two-axis measurements, where both the 'fast' and 'slow' axes can have multiple variables that need to be changed.

Note: only the 'standard' mode of the megasweep has been implemented for the multimegasweep. Refer to the megasweep section for more details.

The general syntax is

```
qtmlab.multimegasweep(<sweep_list1>, <sweep_list2>, <npoints1>, <npoints2>, '<filename>'
    )
```

All of the arguments have been explained in the *multisweep* section, so please check that section for an explanation of the arguments.

5.10 snapshot

The snapshot command can be used to once measure *all* parameters of all devices that are specified in the `meas_list`. The results are stored in a snapshot file with a corresponding timestamp. The function first checks the `meas_list` to retrieve a list of unique devices, and then for every devices requests all attributes where `_read` is in the name.

*Note: this function is in beta release and thus **may be unstable**. Some read parameters (such as the 'auto' read parameters for a lock-in amplifier) should not be queried, since they would alter the settings of the equipment. For some of these parameters (such as the 'auto' attributes of the sr830 code, the code has been adapted. But, for others, this has not been checked and thus may lead to unwanted behavior. Use this function at your own risk!*

The general syntax is

```
qtmlab.snapshot()
```

5.11 getScope

The getScope command captures the curves that are present on the screen of the oscilloscope and stores them in a measurement file. The getScope function is not always used and is therefore written in a separate file, named `scopelab.py`. To use the function, one has to import this module at first as

```
from functions import scopelab
```

The general syntax is

```
scopelab.getScope('<filename>', <GPIBaddr>)
```

An example would be

```
scopelab.getScope('Measurement1.csv', 1)
```

The function accepts the following arguments:

- `<filename>` : the filename of the dataset that will be saved.
- `<GPIBaddr>` : the GPIB address of the scope.

Chapter 6

List of instrument commands and variables

In this section all instruments and the available commands that can be issued will be listed. Note that - as with documentation for any piece of code - the list below may be incomplete and the python instrument files may already contain more functions, so be sure to also check the files itself if you can't find what you're looking for.

6.1 Keithley 2400 SourceMeter

To connect to a Keithley 2400 / 2401 SourceMeter, use

```
from instruments.Keithley2400 import *
keithBG = Keithley2400(<GPIBaddress>)
```

The following commands can be issued to a Keithley 2400 / 2401 SourceMeter.

- `get_iden()` : returns the identification string of the device as a `string`.
- `close()` : closes the GPIB connection to the device.
- `query()` : allows the user to send SCPI/GPIB commands to the device directly. The response is returned without any processing.
- `read_dcv()` : reads the DC voltage when in *Voltage source mode* and returns the value in Volts as `float`.
- `write_dcv(val)` : sets the DC voltage to `val` (must be a `float`) when in *Voltage source mode*. Note that the voltage is limited to ±180 V by the code.
- `read_dci()` : reads the DC current when in *Current source mode* and returns the value in Ampères as `float`.
- `write_dci(val)` : sets the DC current to `value` (must be a `float`)when in *Current source mode*.

- `read_i()` : reads the current and returns the value in Ampere as `float`.
- `read_v()` : reads the voltage[1] and returns the value in Volts as `float`.
- `write_Vrange()` : changes the sensitivity of the voltage measurement. Inputs can be:
  `['MAX', 'Max', 'maximum', '210']`
  `['DEF', 'def', 'default', '21']`
  `['MIN', 'min', 'minimum']`
- `read_output()` : reads whether the source output is on and returns a `boolean (integer)`.
- `write_output()` : turns the output on or off. Inputs can be:
  `[1, 'ON', 'On', 'on']`
  `[0, 'OFF', 'Off', 'off']`

[1] Useful for DMM ("Measure-only") mode. Refer to the Keithley manual for instructions and warnings!

6.2 Keithley 2000 MultiMeter

To connect to a Keithley 2000 MultiMeter, use

```
from instruments.Keithley2000 import *
keithR = Keithley2000(<GPIBaddress>)
```

The following commands can be issued to a Keithley 2000 Multimeter.

- `get_iden()` : returns the identification string of the device as a `string`.
- `close()` : closes the GPIB connection to the device.
- `read_dcv()` : reads the DC voltage and returns the value in Volts as `float`.

6.3 LakeShore 332 Temperature Controller

To connect to a LakeShore 331 / 332 Temperature Controller, use

```
from instruments.Lake332 import *
lake = Lake332(<GPIBaddress>)
```

The following commands can be issued to a LakeShore 331 / 332 Temperature Controller.

- `get_iden()` : returns the identification string of the device as a `string`.
- `close()` : closes the GPIB connection to the device.
- `read_temp()` : returns the current temperature in Kelvin as `float`.
- `write_PID(P, I, D)`: sets the PID values of the controller to `P`, `I` and `D`.
  These values should all be of the type `float`
- `write_setp(val)` : sets the temperature setpoint in Kelvin as to `val` (a `float`).
- `write_range(val)` : changes the setpoint range. Inputs can be of the following form:
  `['Off', 'off', 0]`
  `['Low', 'low', 1]`
  `['Medium', 'medium', 2]`
  `['High', 'high', 3]`
- `heater_off()` : turns off the heater.

6.4 Stanford Research 830 Lock-In Amplifier

To connect to a Stanford Research 830 Lock-In Amplifier, use

```
from instruments.sr830 import *
lake = sr830(<GPIBaddress>)
```

The following commands can be issued to a Stanford Research 830 Lock-In Amplifier.

- `get_iden()` : returns the identification string of the device as a `string`.
- `close()` : closes the GPIB connection to the device.
- `read_x()` : returns the X reading as a `float`.
- `read_y()` : returns the Y reading as a `float`.
- `read_r()` : returns the R reading as a `float`.
- `read_theta()` : returns the θ reading as a `float`.
- `read_freq()` : returns the frequency as a `float`.
- `write_freq(val)` : sets the frequency to `val` (a `float`).
- `read_amp()` : returns the amplitude of the Sine Out as `float`.
- `write_amp(val)` : sets the Sine Out amplitude to `val` (a `float`).
- `read_phase()` : returns the phase as `float`.
- `write_phase(val)` : sets the phase to `val` (a `float`).
- `read_sens()` : returns the sensitivity as `int`.
- `write_sens(val)` : sets the sensitivity to `val` (an `int`).
- `read_auto_x()` : returns the X reading as a `float`. In addition, the code checks whether the measured value (of R) is within 10% and 90% of the measurement range. If not, the code automatically adjusts the sensitivity until the value is within the specified range. It then waits for 5 seconds to allow for stabilization of the readings.

For all four DAC outputs, one can use the following commands (here shown for DAC1 - just replace 1 by the number of the output that is used).

- `read_dac1()` : returns the DAC1 reading as `float`.
- `write_dac1(val)` : sets the DAC1 output to `val` (a `float`).

6.5 Oxford IPS120-10 Magnet Controller

To connect to a Oxford IPS120-10, use

```
from instruments.ips120 import *
oxM = ips120(<GPIBaddress>)
```

The following commands can be issued to an Oxford IPS120-10 Magnet Controller.

- `get_iden()` : returns the identification string of the device as a `string`.
- `query()` : allows the user to send SCPI/GPIB commands to the device directly. The response is returned without any processing.
- `close()` : closes the GPIB connection to the device.
- `unlock()` : unlocks the magnet controller after the device has been powered on.
- `hold()` : puts the magnet's activity state to hold.
- `clamp()` : clamps the outputs of the magnet power supply.
- `read_fvalue()` : returns the field value in Tesla as a `float`.
- `write_fvalue(val)` : sets the field value setpoint to `val` Tesla (a `float`). The magnet is also set to 'To setpoint' so that it will automatically go there with the ramp rate as specified in the `rate` parameter.
- `read_rate()` : returns the sweep rate value in Tesla/min as a `float`.
- `write_rate(val)` : sets the sweep rate to `val` Tesla/min (a `float`).
- `write_gotozero()` : ramps the magnet down to 0 Tesla with the rate as set by `rate`.
- `hON()` : turns the switch heater on.
- `hON()` : turns the switch heater off.
- `read_setp()` : returns the field setpoint in Tesla as a `float`.
- `read_heater()` : returns whether the switch heater is on as a `boolean (int)`.
- `status()` : returns a human-readable status report of the power supply.

6.6 Oxford MercuryiPS Magnet Controller

The MercuryiPS power supply can be controlled over GPIB or ethernet. The appropriate instrument class must be loaded. Both classes have almost[2] the same commands and therefore switching between communication protocols should be easy (from a QTMtoolbox point of view).

To connect to the device over GPIB, use

```
from instruments.MercuryiPS_GPIB import *
VRM = MercuryiPS()
```

NOTE: We use the standard GPIB address that is given to the magnet, i.e. `GPIB0::1::1::INSTR`

To connect to the device over ethernet, use

```
from instruments.MercuryiPS_eth import *
VRM = MercuryiPS(<IPaddress>, <Port=7020>)
```

The following commands can be issued to an Oxford MercuryiPS Magnet Controller (over GPIB). We assume that a vector magnet is installed with three groups, `GRPX`, `GRPY` and `GRPZ`.

- `close()` : closes the GPIB connection to the device.
- `visa_query(val)` : allows one to query SCPI commands directly. The response is not processed or edited. Use for debugging or special purposes only.
- `get_iden()` : returns the identification string of the device as a `string`.
- `read_fvalueX()` : returns the X-axis field strength in Tesla as a `float`.
- `read_fvalueY()` : returns the Y-axis field strength in Tesla as a `float`.
- `read_fvalueZ()` : returns the Z-axis field strength in Tesla as a `float`.
- `read_vector()` : returns the [X,Y,Z] field strength in Tesla as a `list`.
- `write_fvalueX(val)` : sets the X-axis field value setpoint to `val` Tesla (a `float`). The X-axis is also set to 'RTOS' so that it will automatically go there with the ramp rate as specified in the `rateX` parameter.
- `write_fvalueX(val)` : sets the Y-axis field value setpoint to `val` Tesla (a `float`). The Y-axis is also set to 'RTOS' so that it will automatically go there with the ramp rate as specified in the `rateY` parameter.
- `write_fvalueX(val)` : sets the Z-axis field value setpoint to `val` Tesla (a `float`). The Z-axis is also set to 'RTOS' so that it will automatically go there with the ramp rate as specified in the `rateZ` parameter.

[2] Work in progress. To port functions from one to the other, add or remove carriage return characters.

- `write_vector([valX, valY, valZ])` : sets the vector field value setpoint to the provided values. All axes are also set to 'RTOS' so that they will automatically go there with the ramp rates as specified in the `rate` parameters.
- `read_rateX()` : returns the X-axis sweep rate value in Tesla/min as a `float`.
- `read_rateY()` : returns the Y-axis sweep rate value in Tesla/min as a `float`.
- `read_rateZ()` : returns the Z-axis sweep rate value in Tesla/min as a `float`.
- `read_rates()` : returns the [X,Y,Z] sweep rate values in Tesla/min as a `list`.
- `write_rateX(val)` : sets the X-axis sweep rate value to `val` Tesla/min (a `float`).
- `write_rateY(val)` : sets the Y-axis sweep rate value to `val` Tesla/min (a `float`).
- `write_rateZ(val)` : sets the Z-axis sweep rate value to `val` Tesla/min (a `float`).
- `read_state()` : returns the state of all axes as a `string`.
  Responses can be either:
  `CLMP` : axis is clamped
  `HOLD` : axis is on hold
  `RTOS` : axis is changing field (Rotate TO Setpoint)
  The response will be returned as follows:
  `(X) : HOLD, (Y): RTOS, (Z): RTOS`
- `read_temp()` : returns the temperature sensor reading in Kelvin as a `float`.
- `read_status()` : returns the state of all axes as a *single-word* `string`).
  The response can be `HOLD` or `MOVING`:
  `HOLD` : all three axes return HOLD as ACTN parameter
  `MOVING` : at least one axis does not return HOLD
- `read_alarm()` : returns alarm messages as a `string`.
- `read_gotozero()` : sets the vector setpoint to [0, 0, 0] and moves there.
- `read_clamp()` : clamps the output of all axes.
- `read_hold()` : sets all axes to hold.
- `read_setpX()` : returns the X-axis *setpoint* in Tesla as a `float`.
- `read_setpY()` : returns the Y-axis *setpoint* in Tesla as a `float`.
- `read_setpZ()` : returns the Z-axis *setpoint* in Tesla as a `float`.

6.7 Triton Remote Interface

To connect to the Triton RI (Remote interface, i.e. the computer that controls the LakeShore controller and system controls such as valves and pumps), use[3]

```
from instruments.Triton import *
trit = Triton(<IPaddress>, <Port>)
```

The following commands can be issued to the Triton RI:

- `close()` : closes the GPIB connection to the device.
- `query()` : allows the user to send SCPI/GPIB commands to the device directly. The response is returned without any processing.

The following commands can be used for any temperature channel (1-16). The examples below use channel 5. Refer to the LakeShore configuration to see which channel belongs to which sensor. This may differ from system to system.

- `read_temp5()` : returns the temperature of channel 5 in Kelvin as a `float`.
- `read_Tenab5()` : returns whether temperature channel 5 is enabled in the LakeShore scanner as a `boolean (int}`.
- `write_Tenab5()` : sets whether channel 5 must be enabled or disabled in the LakeShore scanner. Inputs are `'ON'` or `'OFF'`.

The following command can be used for any pressure sensor (1-6). The pressure sensors are defined as follows:

1: Tank pressure
2: Condense pressure
3: Still pressure
4: Turbo back pressure
5: Foreline back pressure
6: OVC pressure

The example reads the pressure of channel 2.

- `read_pres2()` : returns the temperature of channel 2 in bar as a `float`.

The following commands can be issued to the Triton RI:

- `read_Tchan()` : returns the temperature channel that is used for temperature control as an `integer`.
- `write_Tchan(val)` : sets the temperature control channel to `val` (an `integer`).
- `read_Tchan()` : returns the temperature control setpoint in Kelvin as a `float`.
- `write_Tchan(val)` : sets the temperature control setpoint to `val` Kelvin (a `float`).
- `read_PID()` : returns the temperature control PID values as a `list` of `float` values.

[3]The default value of the port is 33576, which is automatically selected when no port is specified.

- `write_PID(p, i, d)` : writes the PID values according to the specified values for P, I, and D (all `float` inputs).
- `read_range()` : returns heater range in Amps as a `float`.
- `write_range(val)` : sets the heater range to the `val` in mA (a `float`), or to the value closest to it.
- `loop_on()` : turns the PID controlled loop on.
- `loop_off()` : turns the PID controlled loop off.
- `read_loop()` : returns the state of the loop as a `float`.
- `read_Trate()` : returns the temperature ramp rate in K/min as `float`.
- `write_Trate(val)` : sets the temperature ramp rate to `val` K/min (a `float`).
- `read_ratestatus()` : returns whether the ramping mode is enabled as `boolean (int)`.
- `write_ratestatus(val)` : enables or disables the ramping mode. Inputs are `'ON'` or or `'OFF'`.
- `read_status()` : returns the system status `string`.
- `read_action()` : returns the current automated action of the system `string`. Options are:

```
'Precooling'
'Empty precool loop'
'Condensing'
'Condensing and circulating'
'Idle'
'Collecting the mixture'
```

- `read_compstate()` : returns the status of the 3He compressor (`'ON'` or `'OFF'`).
- `read_fpstate()` : returns the status of the forepump (`'ON'` or `'OFF'`).
- `read_PTRstate()` : returns the status of the PTR compressor (`'ON'` or `'OFF'`).
- `read_turbstate()` : returns the status of the turbo pump (`'ON'` or `'OFF'`).
- `read_turbspeed()` : returns the speed of the turbo pump in Hz as a `float`.
- `read_turbhours()` : returns the cumulative operational hours of the turbo pump as a `float`.
- `read_Tchandefs()` : returns a list of all assigned temperature channels in the system.
- `info()` : returns an extensive system summary in human-readable form.

The following commands can be used for any actuated valve (1-9). The example uses valve 5.

- `read_valve5()` : returns status of valve 5. Responses are `'OPEN'`, `'CLOSE'`, `'TOGGLE'`.

Experimental features (use with care):

- `read_Hchamber()` : returns the chamber heater power (in ??) as `float`.
- `write_Hchamber(val)` : sets the chamber heater power to `val` (??) (a `float`).
- `read_Hstill()` : returns the still heater power (in ??) as `float`.

- `write_Hstill(val)` : sets the still heater power to `val` (??) (a `float`).

6.8 Delft IVVI rack

To connect to the Delft IVVI rack, use

```
from instruments.IVVI import *
DF = IVVI(<COMport>)
```

The following commands can be issued to the Delft IVVI rack.

NOTE: all commands are programmed for a bipolar range for 16 DACs *only*.
NOTE: every command opens and closes the COM port, to minimise interference with other scripts.

- `read_dacs()` : returns the voltages of all DAC channels as a `list`.
- `write_dac(dac, val)` : sets DAC no. `dac` (an `int`) to `val` V (a `float`).

For all 16 DAC channels, one can use the following commands (here shown for DAC1 - just replace 1 by the number of the DAC that is used).

- `read_dac1()` : returns the voltages of DAC 1 as a `list`.
- `write_dac1(val)` : sets the voltage of DAC 1 to `val` V (a `float`).

6.9 Tektronix TDS 3012C Oscilloscope

To connect to the Tektronix TDS 3012C Oscilloscope, use

```
from instruments.TekTDS3012C import *
tek = TekTDS3012C(<GPIBaddress>)
```

The following commands can be issued to the oscilloscope

- `get_iden()` : returns the identification string of the device as a `string`.
- `close()` : closes the GPIB connection to the device.
- `write_horzdiv(val)` : changes the main time per division to `val` seconds (a `float`).

The scope can measure (by using the `<MEAS>` button on the front panel) up to four different properties of the two inputs. The properties can be read by using the following command (the example is for position 1 - replace it by the number (ordered from top to bottom) you'd like to read. Note that the response only gives the value, so you'll have to write down for yourself what quantity it is actually measuring (for example, make this clear in the `meas_dict`)

- `read_meas1()` : returns the quantity measured at position 1 and returns a `float`.

6.10 Tektronix AFG 1022 Arbitrary Function Generator

To connect to the Tektronix AFG 1022, use

```
from instruments.TekAFG1022 import *
tek = TekAFG1022(<USBaddress>)
```

The following commands can be issued to the function generator (*to control Channel 1*)

- `get_iden()` : returns the identification string of the device as a `string`.
- `close()` : closes the GPIB connection to the device.
- `read_amp()` : returns the amplitude of the waveform in Volts as a `float`.
- `write_amp(val)` : sets the amplitude of the waveform to `val` Volts (a `float`).
- `read_dcv()` : returns the DC offset of the waveform in Volts as a `float`.
- `write_dcv(val)` : sets the DC offset of the waveform to `val` Volts (a `float`).
- `read_freq()` : returns the frequency of the waveform in Volts as a `float`.
- `write_freq(val)` : sets the frequency of the waveform to `val` Volts (a `float`).
- `read_waveform()` : returns the waveform type as a `string`.
- `write_waveform(val)` : sets the waveform type to `val` (a `string`).
  The input should be one of the following options:
  `SIN` : a sine wave
  `SQU` : a square wave
  `PULS` : a pulse
  `RAMP` : a ramp
  `PRN` : noise
- `read_output()` : returns whether the hardware output is on or off as a `boolean`.
- `write_output(val)` : turns the hardware output on or off.
  The input should be one of the following options:
  `['OFF', 'off', 0]`
  `['ON', 'on', 1]`

6.11 Current time module

This module does not connect to any physical device. To 'connect' to the module, use

```
from instruments.curtime import *
ct = curtime()
```

The following commands can be issued to the time module:

- `read_time()` : returns the epoch time[4] in seconds.
- `read_timems()` : returns the epoch time in milliseconds.

[4]The epoch time standard starts on the 1st of January, 1970. From then on, epoch time displays the time as the amount of seconds since this specific timestamp.

6.12 Quantum Design PPMS DynaCool system

This module is designed to be run directly on the PPMS computer itself (in local mode). The Python-NET module must be installed (via `pip install pythonnet`). Furthermore, the `QDinstrument.dll` file must be obtained from Quantum Design's Pharos website, and must then be placed in the QTM-toolbox root folder. To be able to use the DynaCool instrument, make sure that both the MultiVu software and the Python shell (via Spyder) are started as administrator.

To connect to the dynacool system, use

```
from instruments.dynacool import *
ppms = Dynacool()
```

The following commands can be issued to the time module:

- `read_temp()` : returns the system temperature in Kelvin as `float`.
- `write_temp(temp, rate)` : sets the system temperature to `temp` Kelvin with rate `rate` in Kelvin/min (both are a `float`).
- `read_fvalue()` : returns the magnetic field in Gauss as `float`.
- `write_fvalue(field, rate)` : sets the magnetic field to `field` Gauss with rate `rate` in Gauss/sec (both are a `float`).
- `read_position()` : returns the horizontal rotator position in degrees as `float`.
- `write_temp(position, speed)` : sets the horizontal rotator to `position` degrees with speed `speed` in degrees/sec (both are a `float`).
- `waitForTemperature_temp(delay, timeout)` : uses the MultiVu waitfor sequence to check temperature stability, given a `delay` and `timeout` in seconds (as `float`).
- `waitForField_temp(delay, timeout)` : uses the MultiVu waitfor sequence to check field stability, given a `delay` and `timeout` in seconds (as `float`).
- `waitForPosition_temp(delay, timeout)` : uses the MultiVu waitfor sequence to check position stability, given a `delay` and `timeout` in seconds (as `float`).
- `read_position_status()` : returns the status of the horizontal rotator position as `string`.
- `read_fvalue_status()` : returns the status of the magnet controller as `string`.
- `read_temperature_status()` : returns the status of the temperature controller as `string`.
- `read_chamber_status()` : returns the status of the sample chamber controller as `string`.
- `status()` : returns a human-readable status report of the system.

6.13 HP 34401A Multimeter

To connect to a HP 34401A Multimeter, use

```
from instruments.hp34401A import *
hpR = hp34401A(<GPIBaddress>)
```

The following commands can be issued to a Keithley 2000 Multimeter.

- `get_iden()` : returns the identification string of the device as a `string`.
- `close()` : closes the GPIB connection to the device.
- `read_dcv()` : reads the DC voltage and returns the value in Volts as `float`.

6.14 Scientific Instruments 9700 Temperature Controller

To connect to a Scientific Instruments 9700 Temperature Controller, use

```
from instruments.si9700 import *
siT = si9700(<GPIBaddress>)
```

The following commands can be issued to a Scientific Instruments 9700 Temperature Controller.

- `get_iden()` : returns the identification string of the device as a `string`.
- `close()` : closes the GPIB connection to the device.
- `read_tempA()` : reads the temperature of channel A and returns the value as `float`.
- `read_tempB()` : reads the temperature of channel B and returns the value as `float`.
- `read_setp()` : reads the temperature setpoint and returns the value as `float`.
- `write_setp(val)` : sets the setpoint to `val` (a `float`).

6.15 Digilent Analog Discovery 2

The use of a Digilent Analog Discovery 2 requires a bit of extra code compared to other instruments. In the following, we assume that the user wants to generate a signal on W1 (waveform generator 1) and measure it on CH1 (scope channel 1), including triggering. The Digilent device should be connected to the PC via a USB cable. The code always assumes that no more than one Digilent unit is connected to the PC.

```
# Load module
from instruments.DigiAD2 import *

# Bind to instrument (the argument here denotes the configuration of the device,
# see the list of commands below for more info on this.
a = DigiAD2(1)

# Open scope first
a.open_scope(20e6, 10e3)

# Set up trigger
a.trigger(True, 'analog', 0, 3, edge_rising=False, level=0.25, hysteresis=0.1)

# Write waveform
a.set_wav1('triangle', offset=0, frequency=3e3, amplitude=1)

# Measure 10 times and plot
plt.figure()
for i in range(10):
    t, y = a.get_wav1()
    plt.plot(t, y)

# Note: a.close() alone does not kill the wavegen/scope.
a.close_wav1()
a.close_scope()
a.close()
```

When the device is initialized, you don't provide a GPIB address as number, but rather you provide a value for the configuration of the device. Most often, you either want to use configuration 0 or 1, being:

- 0: The scope buffer is max 8k (8192) points, the waveform generator max 4k (4096) points
- 1: The scope buffer is max 16k (16384) points, the waveform generator max 1k (1024) points

Note that if you exceed the buffer size, the device will return only zeros as data.

The following commands can be issued to a Digilent Analog Discovery 2.

- `close()` : closes the USB connection to the device. If the device is not closed properly after use, initializing a new instance of the Digilent device will not cause any errors, but will return zeros as data.

- `open_scope(freq, npoints, offset, amp_range)` : opens a connection to the scope process on the Digilent device. The user must provide several arguments, listed below:
  - `freq` : sampling rate or frequency. Default is 20 MHz.
  - `npoints`: number of points to be acquired. Default is 8192. Note that the maximum number of points depends on the configuration that is chosen during initialization.
  - `offset` : offset to be subtracted from the data. Default is 0 V.
  - `amp_range` input range of the scope. Default is ± 5 V.
- `close_scope()` : closes the connection to the scope.
- `read_volt1()` : returns a single reading of CH1 in Volts as a`float`.
- `read_volt2()` : returns a single reading of CH2 in Volts as a`float`.
- `get_wav1()` : returns a list of timestamps (seconds) and voltages (Volts) in two lists of `float` values. Frequency and number of points depend on the `open_scope` parameters. Returns values for CH1.
- `get_wav21()` : returns a list of timestamps (seconds) and voltages (Volts) in two lists of `float` values. Frequency and number of points depend on the `open_scope` parameters. Returns values for CH2.
- `trigger(enable, source, channel, timeout, edge_rising, level, hysteresis)` : configures a trigger for the oscilloscope signals. The user must provide several arguments:
  - `enable` : Boolean to determine whether the trigger should be enabled or not. Default is `True`.
  - `source` : determines the trigger source. Can be `none`, `analog`, `digital` or `external`. Default is `analog`.
  - `channel` : determines the trigger channel. Can be 1:4 for an analog source, or 0-15 for a digital source.
  - `timeout` : determines how long the trigger tries to trigger before a timeout. Default is 0 seconds.
  - `edge_rising` :determines whether the trigger should trigger on rising signals (`True`) or falling signals (`False`). Default is `True`.
  - `level` : the voltage at which the trigger should trigger.
  - `hysteresis` : determines the hysteresis (voltage change) before a trigger can be reset. Useful if there is substantial noise on the signal, crossing the trigger level. In that case, the hysteresis value should be larger than the noise.
- `set_wav1(function, frequency, amplitude, offset, symmetry, wait, run_time, repeat, data)` : can be used to generate a signal on the waveform generator channel 1:
  - `function` : determines the shape of the wavefunction. Options:
    - `sine`
    - `square`
    - `triangle`
    - `noise`
    - `dc`
    - `pulse`
    - `trapezium`

`sine_power`
`ramp_up`
`ramp_down`
`custom` - For custom, see also the `data` argument.

`frequency` : sets the frequency of the generated signal in Hz. Default is 1 kHz.

`amplitude` : sets the pk-pk voltage of the signal in V. Default is 1 V.

`offset` : sets the offset of the generated signal in V. Default is 0 V.

`symmetry` : sets the symmetry of the generated signal in percents. Default is 50 %.

`wait` : sets the wait time before the signal is generated in seconds. Default is 0 s.

`run_time` : sets the duration for how long the signal is generated in seconds. Default is 0 s (indefinite).

`repeat` : sets the number of periods for which the signal must be generated. Default is 0 (indefinite).

`data` : if a `custom` waveform is selected, this `data` parameter must contain an array of voltage values. This array is then repeated with the given amplitude, frequency, etc.

Chapter 7

Data storage

In this section we will discuss how the `qtmlab` module stores its data.

7.1 Data from sweep / record / megasweep commands

Data that has been recorded by any of `sweep`, `record` and `megasweep` commands will be stored in the following format:

- The first line of the file denotes the timestamp that marks the beginning of the experiment (e.g. when the sweep was initialized). After the separator (the `|` character), a list with `s` and `g` values is displayed. Each letter corresponds to a data column, and it denotes whether the displayed value is a setpoint (`s`) or whether it was retrieved (get, `g`) via a measurement.
- The second line of the file shows the command that was invoked in order to produce the data set. From this line one can always deduce what variable was swept and what settings were used.
- The third line contains the list of variables, which can be used to import the data with correct variable names in Origin or Python.
- The rest of the file contains the measurement data.

An example file is shown below.

```
08/21/2019 10:23:05|ggggg
sweep of sr1.dac1 from 0 to 1 in 11 steps (lin spacing) with rate 0.1
sr1.dac1, sr1.x, sr1.y, sr1.r, sr1.amp, sr1.freq
 0.00000e+00,  3.22955e-02, -6.90464e-04,  3.22993e-02,  1.00000e+00,  1.30000e+01
 1.00000e-01,  3.22993e-02, -6.86649e-04,  3.23069e-02,  1.00000e+00,  1.30000e+01
 2.00000e-01,  3.22993e-02, -6.86649e-04,  3.23069e-02,  1.00000e+00,  1.30000e+01
 3.00000e-01,  3.23031e-02, -6.86649e-04,  3.23107e-02,  1.00000e+00,  1.30000e+01
 4.00000e-01,  3.23031e-02, -6.86649e-04,  3.23107e-02,  1.00000e+00,  1.30000e+01
```

Chapter 8

Additional help

In this section we will address some additional topics that may require help during the measurements.

8.1 Using the Moku devices

The Moku devices (Moku:Pro, Moku:Lab, Moku:Go) are powerful devicse that can perform many tasks and can be used for many different purposes. While rudimentary drivers are supplied with QTMtoolbox, the Moku software is still in active development, and as such one might run into situations where the firmware of the device needs to be updated before one can use the Moku again.

Here, we aim to provide some guidelines on to how to get the software up to date.
Note: updating the Moku software requires elevated rights (admin rights) on the computer. If you're not an admin, contact your local system administrator.

Below, we'll outline a typical workflow for installing and updating Moku systems.

8.1.1 Installing Moku software

Firstly, one needs to install the latest version of the Moku desktop software. Go to `https://liquidinstruments.com/products/desktop-apps/` and download and install the application for your platform.

In addition, also download 'mokucli', which can be obtained here: `https://liquidinstruments.com/software/utilities/`.

8.1.2 Updating Moku firmware

Connect the Moku to your computer and start the device. As soon as the LED array at the front of the Moku is fully on, the device is ready for use. Launch the Moku software on the computer. The software should display the connected Moku (see Fig. 8.1). If the software displays an orange exclamation mark next to the Moku, the device has firmware which is outdated. In that case, clicking the device will not start the communication and control of the Moku, but rather will prompt you to update the Moku. Proceed with this firmware update and wait for it to finish.

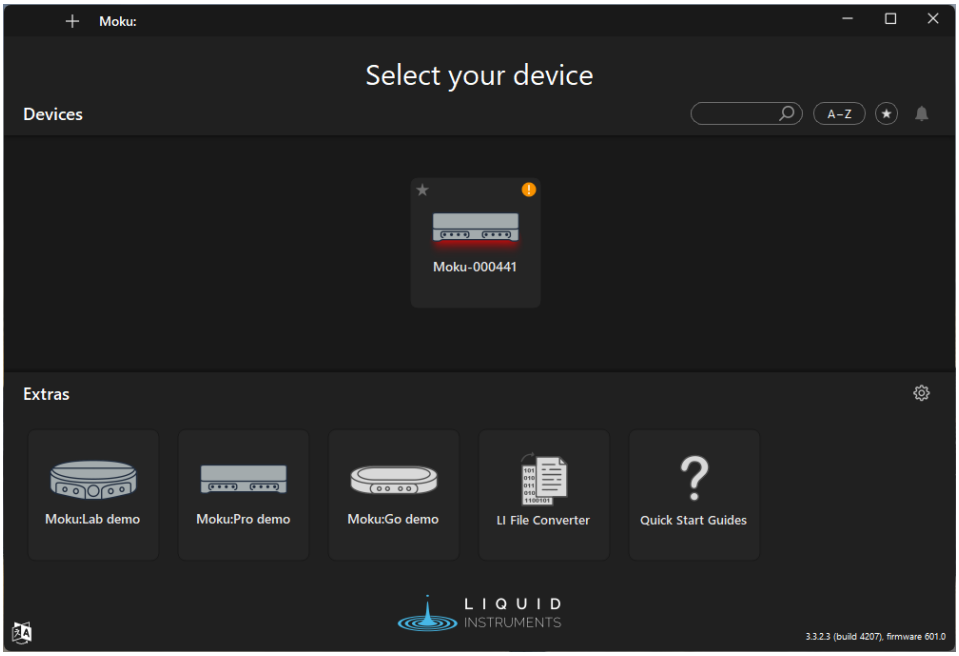

Figure 8.1: Moku software interface. One Moku (Moku-000441) was found. The orange exclamation mark icon indicates that the device's firmware is outdated.

8.1.3 Updating the Python software

In the Python environment, update the moku software to the latest version. If you're using Anaconda, start the Anaconda prompt and run

```
pip install moku --upgrade
```

to update the moku package. Afterwards, we need to get the binaries that correspond to the firmware version of the Moku. To this end, in the Moku software, right click on the device, go to 'Device info' and get the firmware version (see Fig. 8.2). In our example, the firmware version is 601, so downloading the binaries can be done by running the following line of code in the Anaconda prompt

```
mokucli download 601
```

8.1.4 Connecting to the Moku with the measurement code

To communicate to the Moku, one needs to get it's IP address. In Fig. 8.2 you'll see the IP address listed as last option; it can be copied with the button underneath it. To connect to a Moku with Python code, one can use the following snippet of code:

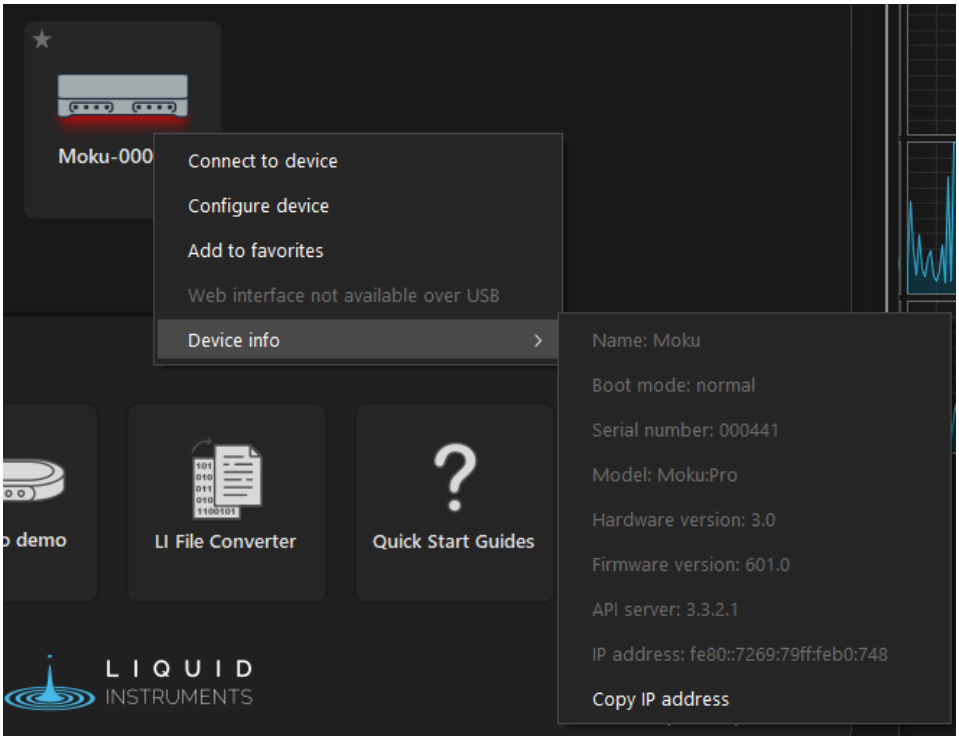

Figure 8.2: Moku device information. Under the 'Device info' tab, you'll find the firmware version, which in this case is 601.

```
from moku.instruments import MultiInstrument
from moku.instruments import Oscilloscope
from moku.instruments import WaveformGenerator

# Connect to Moku:Pro, enforce MultiInstrument mode (reprograms FPGA!)
print('Connecting to Moku:Pro and configuring multi-instrument mode...')
mim = MultiInstrument('[fe80::7269:79ff:feb0:748]', force_connect=True, platform_id=4)
```

Note: the IP address is given as a string which starts and ends with '[' and ']'. These characters are required by the Moku API.

8.1.5 Programming commands

The Moku commands can be found in the Moku Scripting API documentation, which one can find here: `https://apis.liquidinstruments.com/api/`.