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, M.Sc.

Faculty of Science and Technology

QTM Measurement toolbox
Version 2.0.1 - April, 2020

University of Twente
Faculty of Science and Technology

D.H. (Daan) Wielens, M.Sc.
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 (University of Twente), where Python was introduced as main programming language in 2018.

1.1 Requirements

The following software and packages are required:

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

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.

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

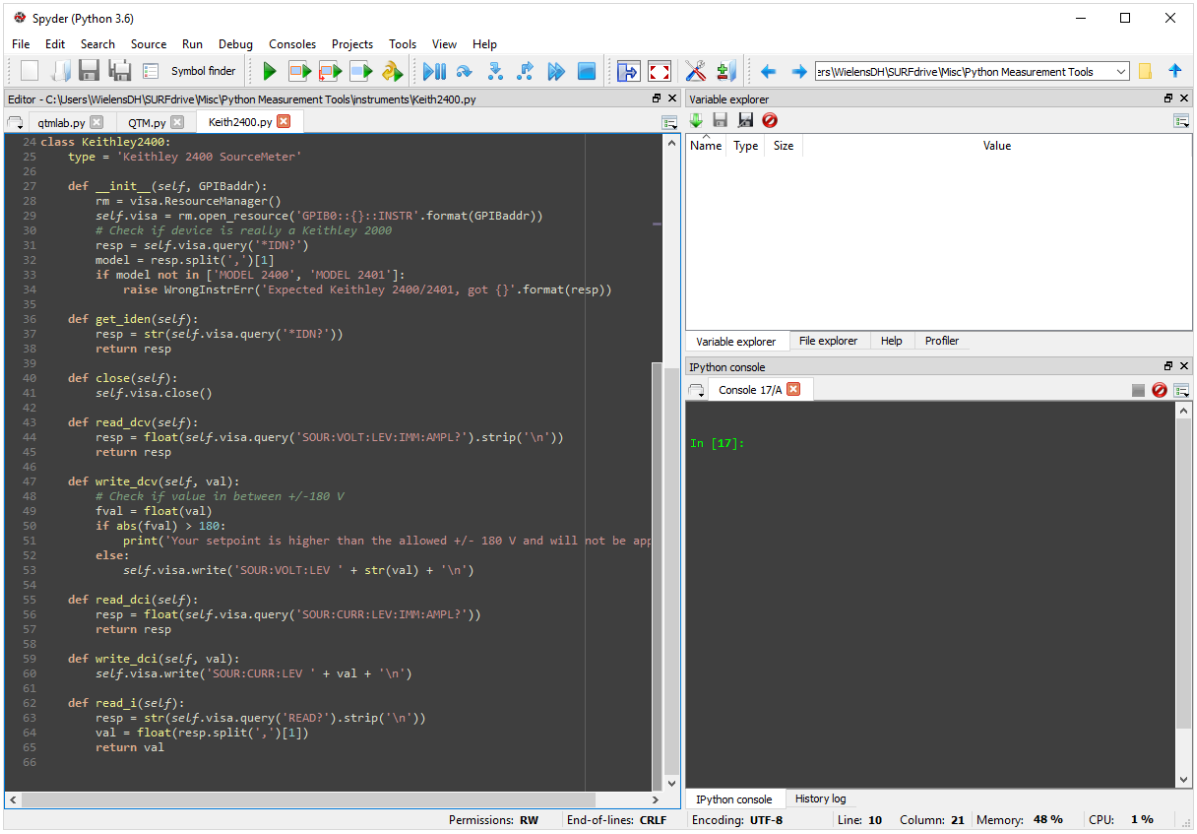

Figure 3.1: The user interface of Sypder. The **editor** is placed at the left side of the window. The right side is divided into two different regions, the **Variable explorer** (or **File explorer**, **Help** or **Profiler** window) and the **IPython console**

.

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.

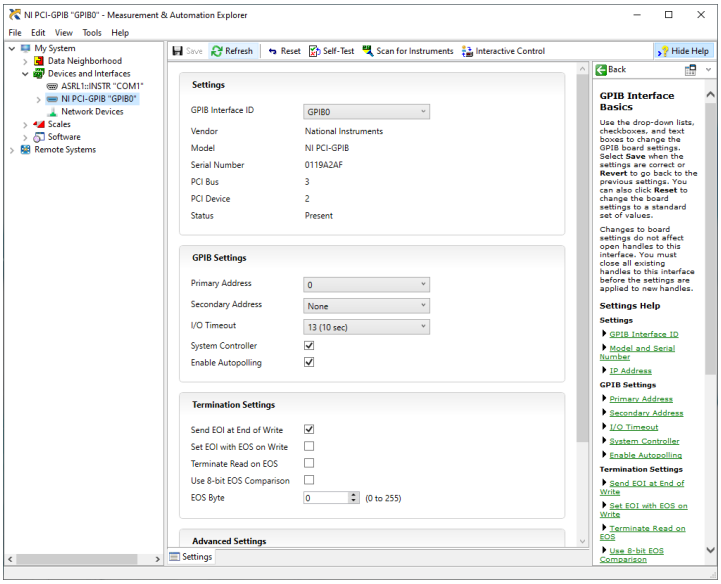

Figure 3.2: Changing the GPIB Interface ID in NI-MAX

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)
```

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

3.4.3 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:

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

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>`.

[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.

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.

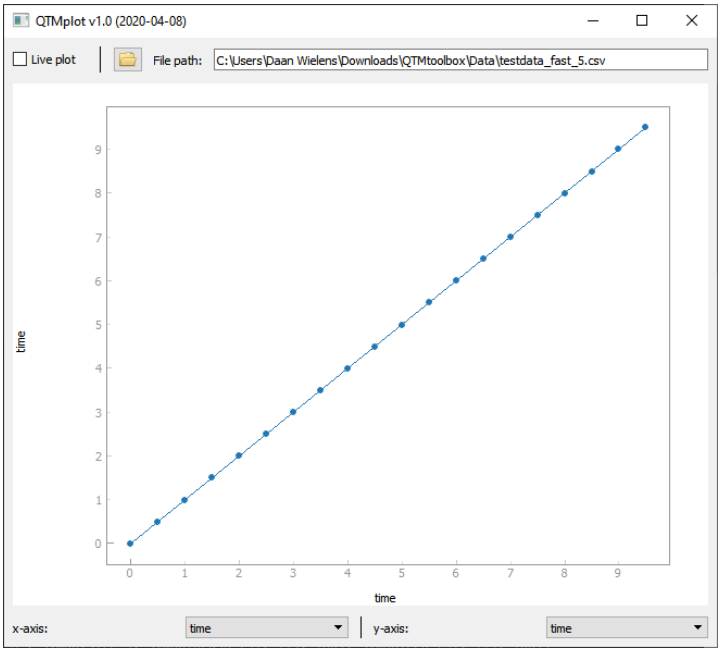

Figure 4.1: GUI of QTMplot.

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. The path can be relative (as in the example. In that case, the file will be stored in the working directory of Python. The path can also be absolute, i.e. `D:\Data\User\Folder\data.csv` for which the data will be stored in the specified path.
- `<sweepdev>` : the name specified here will serve as the name of the swept variable in the header of the data file.

[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.

- `<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 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.

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 datapoints.
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`.
- `<dt>` : number of datapoints that will be taken.
  Must be provided as `float`.
- `<filename>` : the filename of the dataset that will be saved. The path can be relative (as in the example. In that case, the file will be stored in the working directory of Python. The path can also be absolute, i.e. `D:\Data\User\Folder\data.csv` for which the data will be stored in the specified path.

5.6 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. 5.1 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.

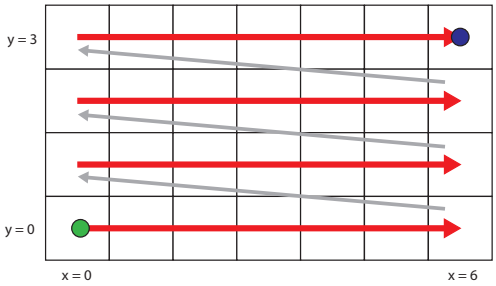

Figure 5.1: The 'standard' mode.

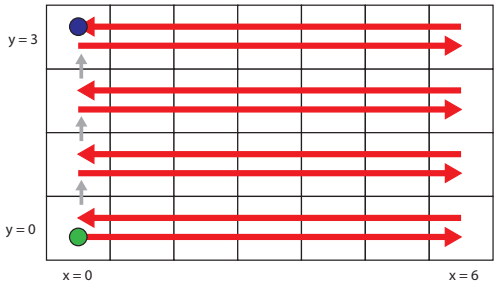

Figure 5.2: The 'updown' mode.

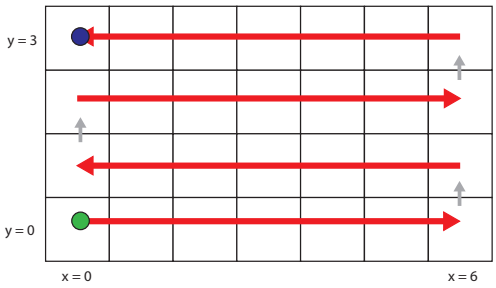

Figure 5.3: The 'serpentine' mode.

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>')
```

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.

5.7 getScope

The getScope command captures the curves that are present on the screen of the oscilloscope and stores them in a subdirectory (named ScopeData) of the currently selected folder. 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. The path must be relative.
- `<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.

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.
- `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`.

[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`).

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`.
- `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.
- `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`.

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

```
from instruments.ips120 import *
oxM = ips120(<IPaddress>, <Port>)
```

The following commands can be issued to the Triton RI.

- `close()` : closes the GPIB connection to the device.
- `read_temp5()` : returns the temperature of channel 5 in Kelvin as a `float`.
- `read_temp8()` : returns the temperature of channel 8 in K as a `float`.
- `read_temp11()` : returns the temperature of channel 11 in K as a `float`.
- `read_PID5()` : returns the temperature setpoint on channel 5 in K as a `float`.
- `read_PID8()` : returns the temperature setpoint on channel 8 in K as a `float`.
- `write_PID5(val)` : puts the temperature setpoint of channel 5 to `val` K (a `float`).
- `write_PID8(val)` : puts the temperature setpoint of channel 8 to `val` K (a `float`).
- `read_range()` : returns heater range as a `string`, including units.
- `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_H1()` : returns the heater power (in ??) as `float`.
- `write_H1(val)` : sets the 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`).

Currently only the first four DACs are implemented, but it is straightforward to add the others if necessary.

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

[3]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 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`.

6.13 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 Keithley 2000 Multimeter.

- `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`).

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).
- 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
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
```