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 1.0.1 - October 2018

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, where Python is introduced as main programming language.

Chapter 2

Structure of the QTM toolbox

The QTM measurement 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 `QTM.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 (in the LabVIEW software, these were so-called *batch files*).
- 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`.
- 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.

2.2 Data flow

The diagram in Fig. 2.1 shows the possible data flows in the QTM measurement toolbox.

As can be seen from the figure, every part of the toolbox can interact with other parts of the toolbox, and the user (via the **IPython console** is able to communicate directly to all modules, as long as the system is initialised[2].

[1]The root directory is the main project folder. All other paths mentioned are relative to this folder.
[2]That is, all modules and instrument definitions are loaded and communication with instruments has been established.

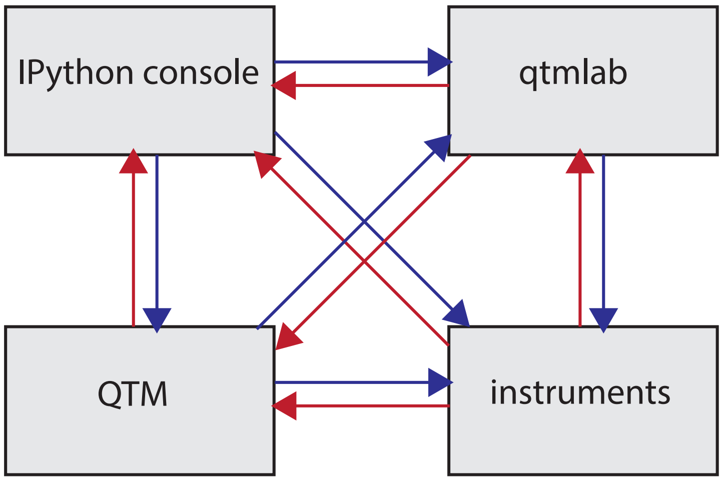


Figure 2.1: Data flow chart for the toolbox. Blue lines represent commands being issued, red lines represent data being returned.

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.

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.

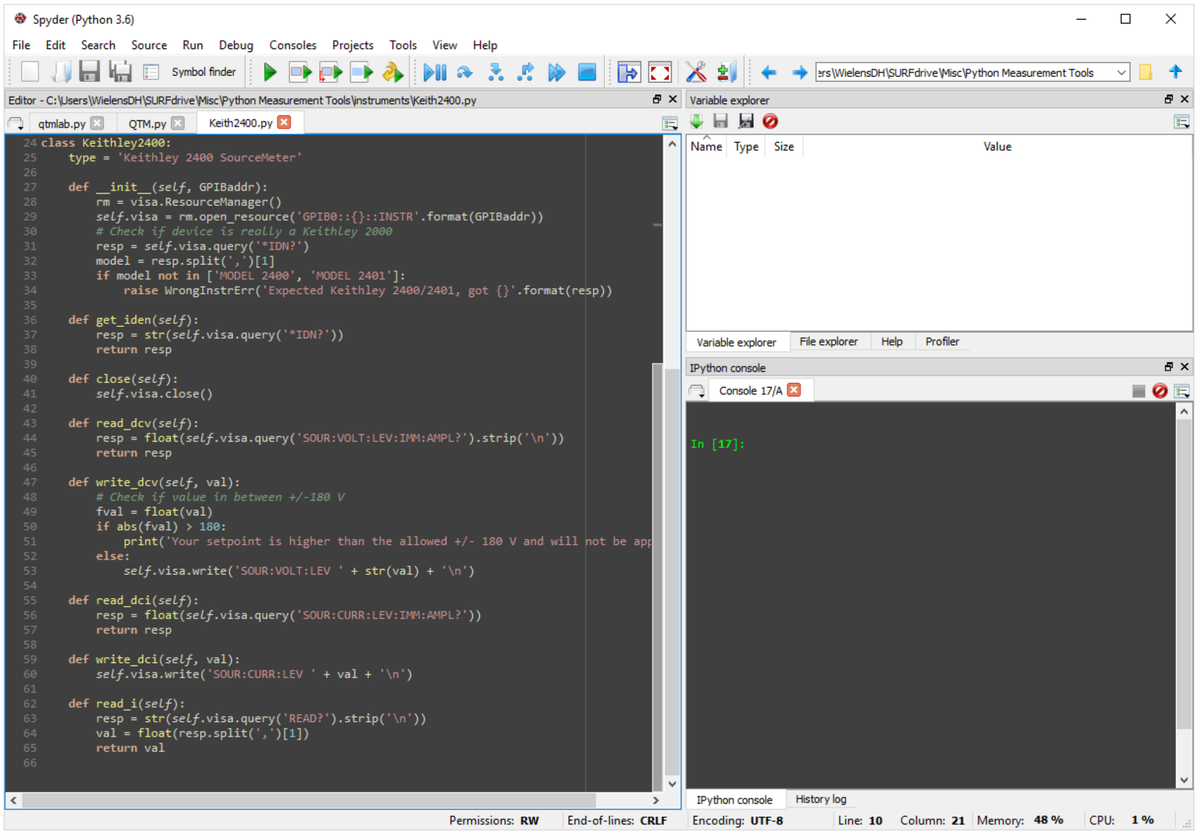
When launched, a window like Fig. 3.1 appears. The window consists of three smaller windows:

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 `QTM.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



```python
class Keithley2400:
    type = 'Keithley 2400 SourceMeter'

    def __init__(self, GPIBaddr):
        rm = visa.ResourceManager()
        self.visa = rm.open_resource('GPIB0::{}::INSTR'.format(GPIBaddr))
        # Check if device is really a Keithley 2000
        resp = self.visa.query('*IDN?')
        model = resp.split(',')[1]
        if model not in ['MODEL 2400', 'MODEL 2401']:
            raise WrongInstrErr('Expected Keithley 2400/2401, got {}'.format(resp))

    def get_iden(self):
        resp = str(self.visa.query('*IDN?'))
        return resp

    def close(self):
        self.visa.close()

    def read_dcv(self):
        resp = float(self.visa.query('SOUR:VOLT:LEV:IMM:AMPL?').strip('\n'))
        return resp

    def write_dcv(self, val):
        # Check if value in between +/-180 V
        fval = float(val)
        if abs(fval) > 180:
            print('Your setpoint is higher than the allowed +/- 180 V and will not be app
        else:
            self.visa.write('SOUR:VOLT:LEV ' + str(val) + '\n')

    def read_dci(self):
        resp = float(self.visa.query('SOUR:CURR:LEV:IMM:AMPL?'))
        return resp

    def write_dci(self, val):
        self.visa.write('SOUR:CURR:LEV ' + val + '\n')

    def read_i(self):
        resp = str(self.visa.query('READ?').strip('\n'))
        val = float(resp.split(',')[1])
        return val

```

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 `QTM.py` file.

3.3 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 - `QTM.py` file to initialise our system.

3.3.1 Adding instruments

To add instruments, we modify the `QTM.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:

```python
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.

3.3.2 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:

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

3.3.3 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 *dictionary*, called `meas_dict`. The structure of the dictionary must be as follows:

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

```
meas_dict = {
  '<storagename>' : {
    'dev' = : <devicename>,
    'var' = '<variable>'
    },
    ...
  }
```

The <**storagename**> will be used in the header of a datafile when measurements are performed and may be chosen by the user. The <**devicename**> should be the exact name of the device as defined in the previous section. Note that there are no quotes around the variable name. The <**variable**> holds the name of the variable that should be read.

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, in the `functions` folder). Every function that starts with `read_` can be measured.
Note that the prefix `read_` itself is omitted here!

For our example, we would change the dictionary to:

```
meas_dict = {
        'keithBG.dcv' : {
                'dev' : kb,
                'var' : 'dcv'
                },
        'keithBG.i' : {
                'dev' : kb,
                'var' : 'i'
                },
        'keithTG.dcv' : {
                'dev' : kt,
                'var' : 'dcv'
                },
        'keithTG.i' : {
                'dev' : kt,
                'var' : 'i'
                }
        'lake.TA' : {
                'dev' : lake,
                'var' : 'temp'
                }
        }
```

3.4 Start the toolbox

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

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 `QTM.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

In the LabVIEW software, the possibility existed to execute batch files: files that contain a sequence of commands that will all be executed sequentially. In the Python toolbox, we can write our 'batch commands' in the `QTM.py` file directly.

In the `QTM.py` file, you'll find a section called `#%% Batch commands`. Simply enter all commands that must be executed sequentially in the file and run the code to start your batch file[1]!

[1]Note that it might be useful to add some `print` commands to tell yourself what measurement is running / finished.

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.

The general syntax is

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

An example would be

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

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>` : **OPTIONAL** The name specified here will serve as the name of the swept variable in the header of the data file. When not specified, `'sweepdev'` 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 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 for each instrument will be listed.

6.1 Keithley 2400 SourceMeter

To connect to a Keithley 2400 / 2401 SourceMeter, use

```python
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 Volts as `float`.

6.2 Keithley 2000 MultiMeter

To connect to a Keithley 2000 MultiMeter, use

```python
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

```python
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]`

6.4 Stanford Research 830 Lock-In Amplifier

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

```python
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

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

The following commands can be issued to a 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 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 in mA 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_H1()` :	returns the heater power (in ??) as `float`.
• `write_H1(val)` :	sets the heater power to `val` (??) (a `float`).

6.7 Tektronix TDS 3012C Oscilloscope

To connect to the Tektronix TDS 3012C Oscilloscope, use

```python
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`.

6.8 Tektronix AFG 1022 Arbitrary Function Generator

To connect to the Tektronix AFG 1022, use

```python
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]`