

PyGMI 3.0 documentation

Python Generic Measurements Interface

Maxime LEROUX, Argonne National Lab.

8/4/2014

Contents

License and Disclaimer	2
Introduction – What is PyGMI ?	3
Installation	6
Launching the program.....	7
Quick start.....	8
Detailed sections	16
Architecture of the program	16
Configuration menu.....	16
Configuring the instruments connections	17
Plotting data.....	19
How the program works	20
Creating a New Instrument Driver	21
Creating a New Instrument Panel	23
Adding a new element to the User interface.....	37
Creating a New Macro Command	40
Creating a new program of measurements	44
Modifying a program of measurements	45
Technical details on multi-threading/parallel-processing	47

License and Disclaimer

Copyright © 2014 , UChicago Argonne, LLC

All Rights Reserved

Python Generic Measurement Interface (PyGMI)

OPEN SOURCE LICENSE

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. Software changes, modifications, or derivative works, should be noted with comments and the author and organization's name.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the names of UChicago Argonne, LLC or the Department of Energy nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.
4. The software and the end-user documentation included with the redistribution, if any, must include the following acknowledgment:

"This product includes software produced by UChicago Argonne, LLC under Contract No. DE-AC02-06CH11357 with the Department of Energy."

DISCLAIMER

THE SOFTWARE IS SUPPLIED "AS IS" WITHOUT WARRANTY OF ANY KIND.

NEITHER THE UNITED STATES GOVERNMENT, NOR THE UNITED STATES DEPARTMENT OF ENERGY, NOR UCHICAGO ARGONNE, LLC, NOR ANY OF THEIR EMPLOYEES, NOR MAXIME LEROUX MAKES ANY WARRANTY, EXPRESS OR IMPLIED, OR ASSUMES ANY LEGAL LIABILITY OR RESPONSIBILITY FOR THE ACCURACY, COMPLETENESS, OR USEFULNESS OF ANY INFORMATION, DATA, APPARATUS, PRODUCT, OR PROCESS DISCLOSED, OR REPRESENTS THAT ITS USE WOULD NOT INFRINGE PRIVATELY OWNED RIGHTS.

Introduction – What is PyGMI ?

Welcome to the documentation of PyGMI !

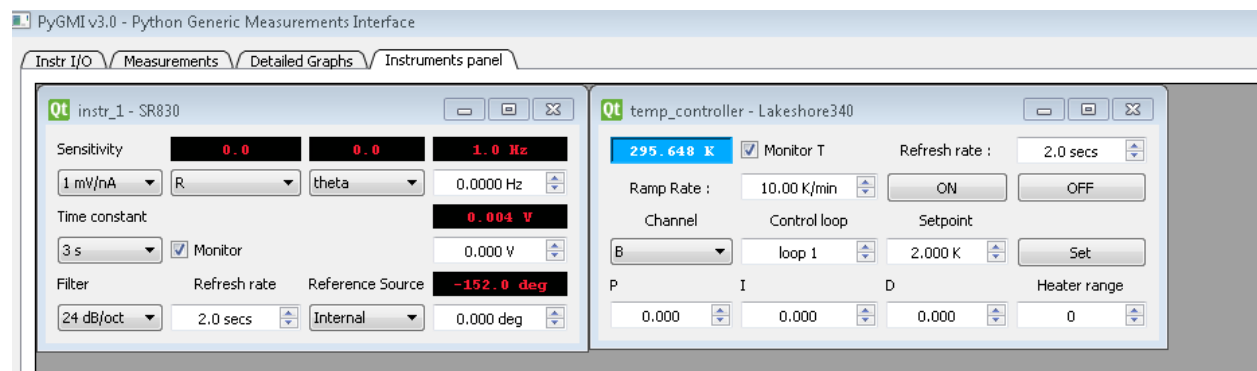
PyGMI is an open source generic interface made in Python/Qt that can take measurements with instruments connected over GPIB, RS232, ethernet or USB using Pyvisa and PySerial. It features:

- live display of the data acquired using pyqtgraph
- a macro editor that can make a series of measurements and even to send e-mails with data!
- generic instruments panels, that can easily be created for a class of instruments using the WYSIWYG interface designer QtDesigner
- a graphical interface that can also be easily modified using QtDesigner
- a main configuration panel that can store instruments types and instruments addresses for easy reconfiguration from one experiment to the next
- measurements that are taken in a separate process so that they do not burden the graphical interface
- generic python drivers can easily be created for new instruments

Its philosophy is that acquiring data with instruments, displaying it live and saving these data is a very generic thing, whereas the core of an experiment is: what instruments do what and in which order. PyGMI therefore provides a framework that relieves the user from doing all of those generic tasks, so that taking measurements become as simple as writing “what instruments do what and in which order” such a measurements script:

```
#reserve the access to the instruments, then discuss with them
with reserved_bus_access:
    #Measure T
    if f.temp_controller_on:T=m.temp_controller.query_temp('B')
    #Measure R and theta
    if f.instr_on_1:freq,R1,theta1=m.instr_1.query_f_R_theta()
    #Measure R and theta
    if f.instr_on_1:freq,R2,theta2=m.instr_1.query_f_R_theta()
R=(R1+R2)/2.0
theta=(theta1+theta2)/2.0
```

Here are some generic instruments panels



Here is a simple macro

Macro Editor

```
Set voltage 1 to 1e-1 V
Set Save file to measurements data/demo-0.1V.txt
Start Demo_script()
Set heater range to 4
Set Loop 1 to 295 K
Set Loop 1 to 300 K @ 0.5 K/min
Wait for channel B to reach 300 +/- 0.1 K
Stop Measurements

Set voltage 1 to 0.5 V
Set Save file to measurements data/demo-0.5V.txt
Start Demo_script()
Set heater range to 4
Set Loop 1 to 295 K
Set Loop 1 to 300 K @ 0.5 K/min
Wait for channel B to reach 300 +/- 0.1 K
Stop Measurements
```

Here is the instruments configuration panel

PyGMI v3.0 - Python Generic Measurements Interface

Instr I/O Measurements Detailed Graphs Instruments panel

List of Instruments

	Instrument Name	Visa Address
1	LSC1MODEL340,341236,001903	ID12_LS340_Temp_Controller
2	Stanford_Research_Systems,SR830,s/n37650,ver1.06	ID11_SR830_Yellow
3	Stanford_Research_Systems,SR830,s/n36895,ver1.06	ID20_SR830_Orange

Scan for Instruments
Initialize Instruments

Save Instruments configuration
Load Instruments configuration

Visa Addresses

Instr	Visa Address
Instr1	GPB1::18
Instr2	GPB1::18
Instr3	GPB1::13
Instr4	GPB1::11
Instr5	GPB1::14
Instr6	GPB1::18
Instr7	GPB1::10
Instr8	ID11_SR830_Yellow
Instr9	ID20_SR830_Orange

Temperature Controller

Parameter	Value
ID12_LS340_Temp_Controller	Lakeshore340
Magnet power supply X - axis	AAA_Test_Instruments
Magnet power supply Y - axis	AAA_Test_Instruments
Magnet power supply Z - axis	AAA_Test_Instruments
ID20_AM420_magnet	AAA_Test_Instruments
Instr14 Lho Level	AAA_Test_Instruments
Current source	AAA_Test_Instruments
rotator	AAA_Test_Instruments
COMP8	AAA_Test_Instruments
BNC switch	AAA_Test_Instruments

Parameters

Active channels - list 1
A,B

Active channels - list 2
A,B,C

Mapping
A:1,B:2

Instruments testing

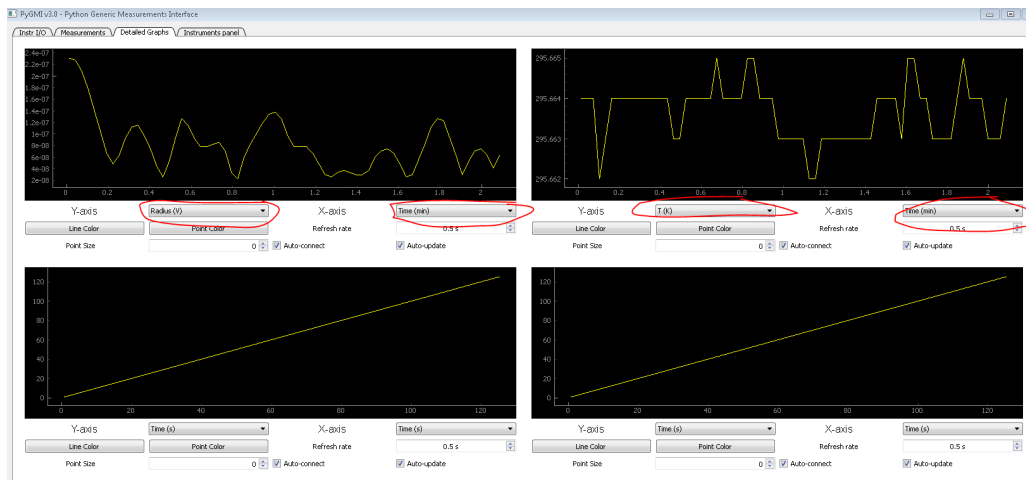
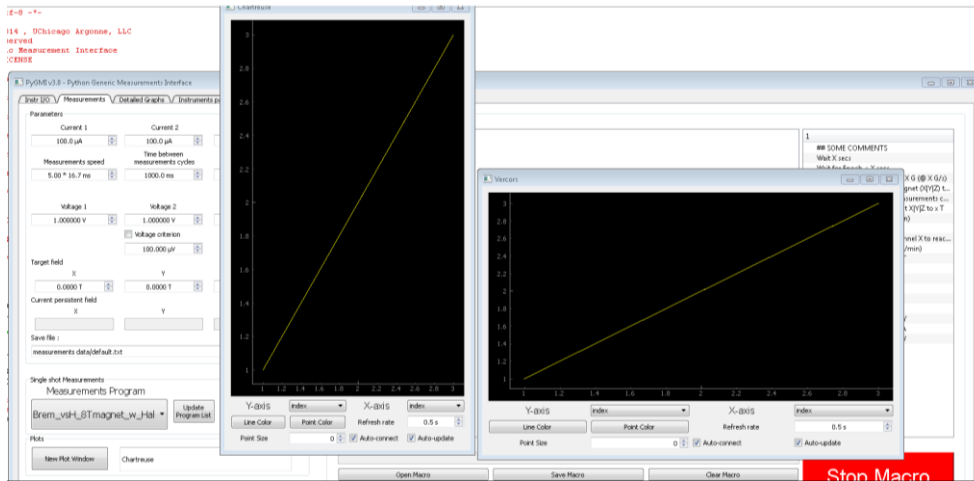
Send command
A,B,C

To
GPB0::0

Answer

Write
Ask
Read

And here are some screenshots of the windows displaying live data



Installation

The PyGMI script itself does not require installation; it can be launched from the folder where it was copied to, however the packages it relies on must be installed beforehand.

PyGMI was designed for use with Python 2.7 and it requires a few python packages. It is known to work on Windows 7 after installing the following packages

- PySide 1.2.2 (GUI) (and Qt 4.8 (qt-win-opensource-4.8.5-vs2008), but PySyde now seems to come bundled with the necessary Qt 4.8 libraries so you don't have to install them separately)
- Pyvisa 1.5 (VISA connections to instruments over GPIB, USB, ethernet, etc..)
- PySerial 2.7 (RS232 connections to instruments)
- Pyqtgraph 0.9.10 (fast, live plotting)

PyGMI also uses the following packages that come preinstalled with Python 2.7: os, time, sys, re (commands parsing), Queue (data exchange between threads), threading (multithreading handling), and for the emailing capability: smtplib, zipfile and email.

Numpy and Scipy are not required, but you may find their functions convenient for some standard data processing in the measurements scripts.

NB: At the time of writing, all of these packages were open source or at least free for academic use.

On Linux, the installation of these packages should be straightforward using the package manager or pip.

On Windows, installation should also be straightforward with the executable installers that you can find on most python packages homepages. Note that the installer is usually provided for the 32-bit version of Python 2.7, rather than the 64 bit version, so if you are not familiar with the subtleties of compiling for windows, you would better install and use the 32-bit version of Python 2.7 (also referred to as x86 sometimes). Note that you can install and use the 32 bit version of Python even if your Windows and CPU are 64 bit versions.

You can also install the packages using "pip" the PyPA recommended tool for installing Python packages. See the homepage of pip for easy installation instructions, but at the time of writing it was as simple as:

1. Go to pip homepage (<https://pip.pypa.io>) to get "get-pip.py"
2. open a command line (windows key+R > type "cmd">press Enter)
3. go to the folder where you downloaded get-pip.py, for example
 - "cd C:\Users\myusername\Downloads"
4. Then run :
 - "python get-pip.py"

NB: if the command "python" is not recognized, make sure that your python installation folder (by default "C:\Python27") is in the PATH environment variable. You can check the content of PATH with the command:

echo %PATH%

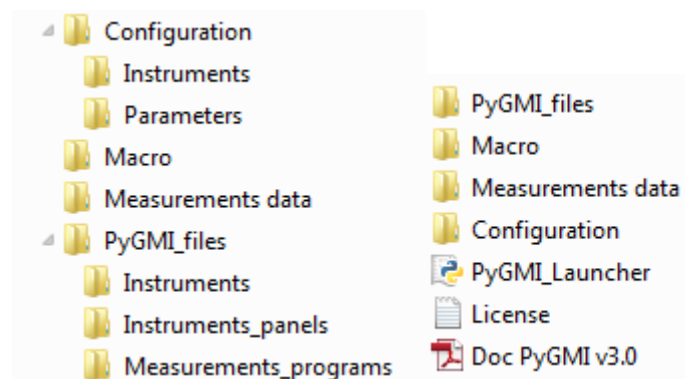
If your Python installation folder is not in there, you can append the Python path to the PATH variable using the following command

set PATH=%PATH%;C:\Python27

5. Let the download and installation of pip finish
6. Then run :
 - “python –m pip install PySide”
 - “python –m pip install pyvisa”
 - “python –m pip install pyserial”
 - “python –m pip install pyqtgraph”
7. That’s it !

Launching the program

The PyGMI folder should contain the following folder structure and base files:



You can launch the program by double-clicking on PyGMI_Launcher.py, or by opening this file with IDLE and then running it by pressing F5, or by using any other Python editor that can run python scripts. The first time it may take up to a few minutes to launch as it compiles various .py files, and then it should just take a few seconds.

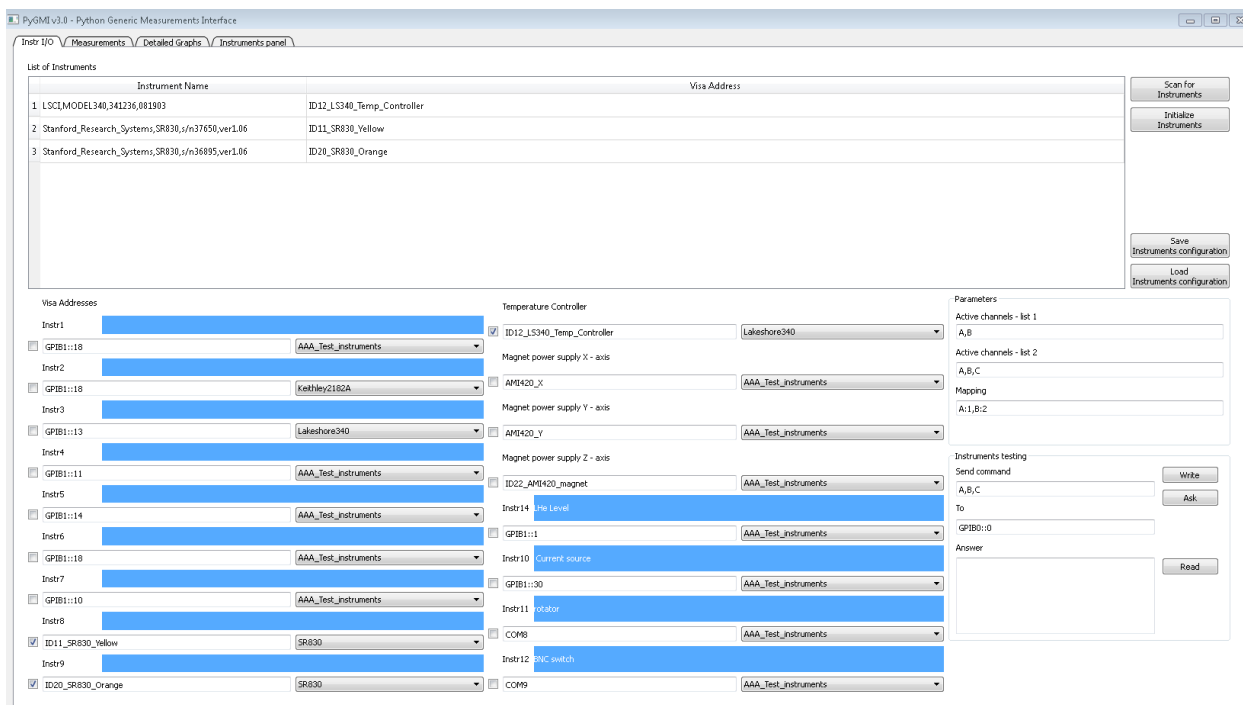
When the program starts it should open on the Instr I/O tab. The following Quick start section will make you familiar with most of the functions of the program by showing how to make a simple voltage measurement with fixed current as a function of temperature. Otherwise you can jump directly to the detailed sections which will present more advanced things such as adding a new instrument driver, creating a new instrument panel, or creating a Macro command.

Quick start

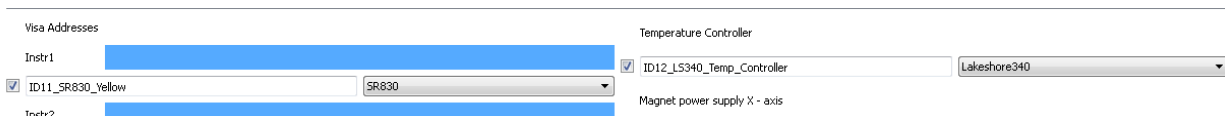
For this quick introduction to the program we will run a simple measurement with one SRS 830 lock-in and one Lakeshore 340 temperature controller.

First, we will configure the instruments connection. Make sure both instruments are connected through GPIB to the computer, and then launch PyGMI (see instructions above).

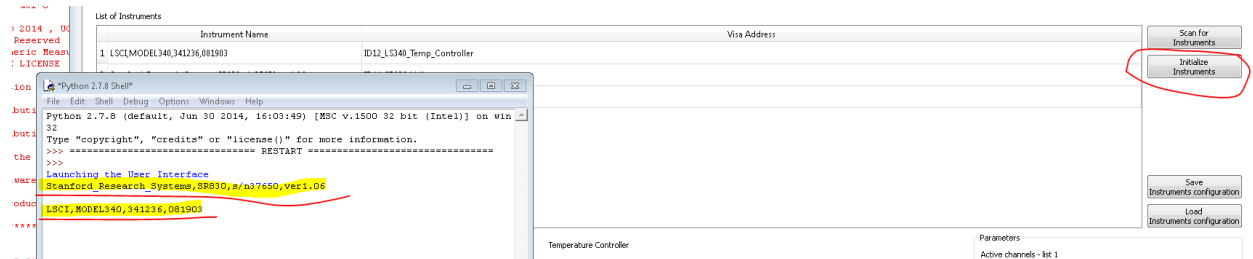
On the first tab “Instr I/O”, press the button “Scan for instruments”. This will poll all available visa connected instruments. After a few seconds, we see this:



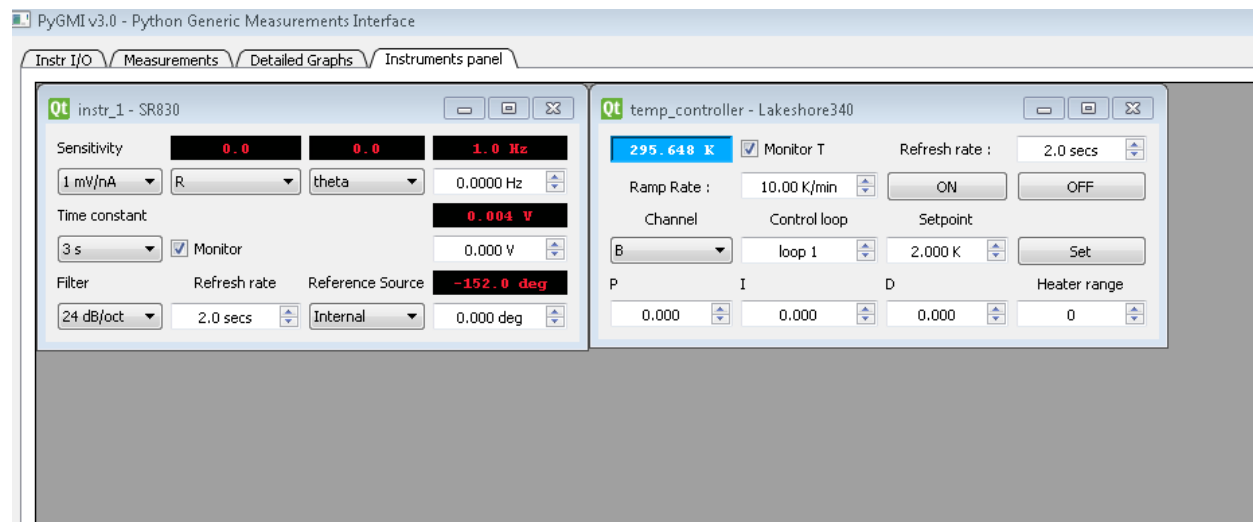
The serial names of instruments appear in the left column, while their corresponding visa address appear in the right column. For this simple measurement we will define the first lock-in as “instr1” and the lakeshore340 as “Temperature controller”. So below Instr1 we enter the address of the lock-in and select SR830 from the adjacent drop-down menu to specify the instruments type, and check the box on the left to notify that this instrument will be used. Then in Temperature controller, we do the same: enter the address of the Lakeshore340, select Lakeshore340 from the drop-down menu, and click the checkbox.



Now that our connection set-up is ready, we click initialize instruments (which instantiate the instruments driver objects). In the Python Shell, the connected instruments should give a standard answer, such as their serial name. If Python complains, check the instruments addresses and connections, then retry the previous step.



Now, we can move to the fourth tab “Instruments Panel” where a small interface window has been created for each instruments. By clicking “Monitor” the live values of the instruments will be updated every few seconds. The instruments can also be configured as needed (with or without Monitor checked)

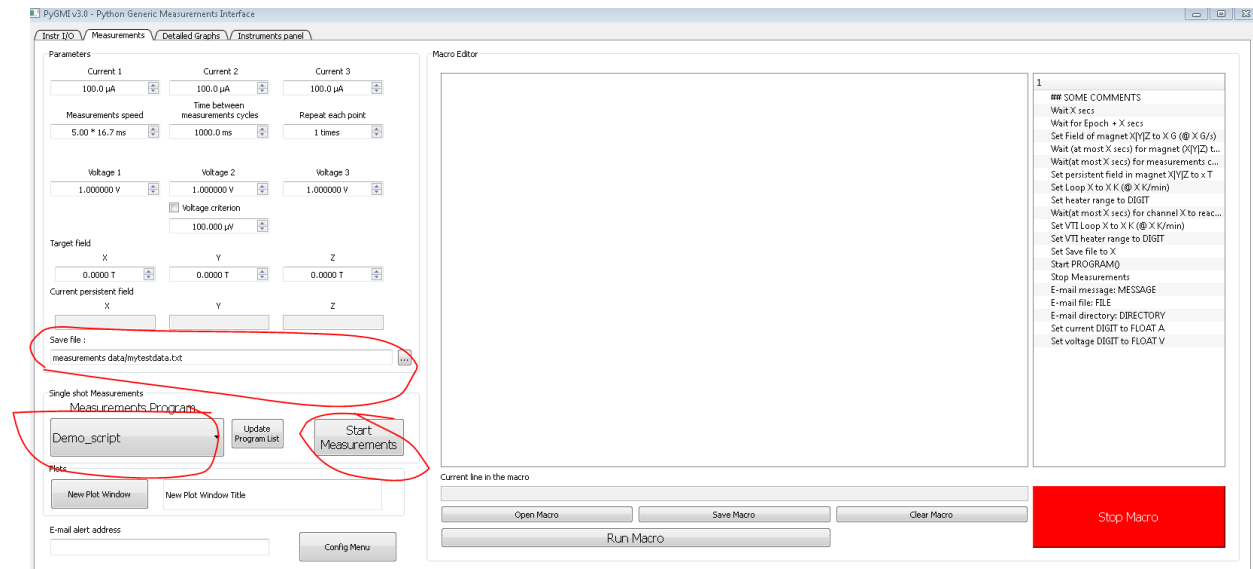


NB1: **Not all** instruments drivers have a graphical interface, but it’s fairly easy to create one yourself that suit your needs, refer to section Creating a New Instrument Panel for more details.

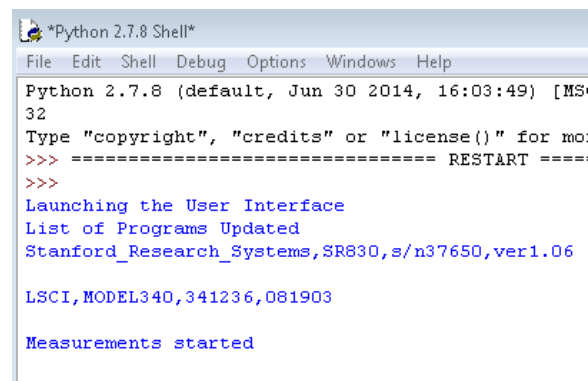
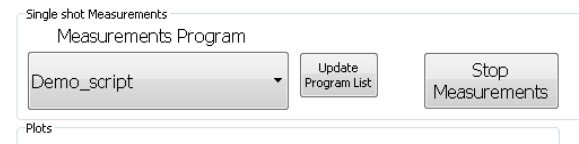
NB2: Refer to section Creating a New Instrument Driver to see how to create a driver for new types of instruments which are not already available in the drop-down menu.

Now, let’s move to the second tab “Measurements” to prepare our measurement script. As you can see some small boxes are available to enter values. These values will be fetched when a measurement script is started, and will be available inside the measurement script (but they won’t be updated if the user change them during the measurements). (See Adding a new element to the User interface for how to add a new entry in the interface with the editor QtDesigner.)

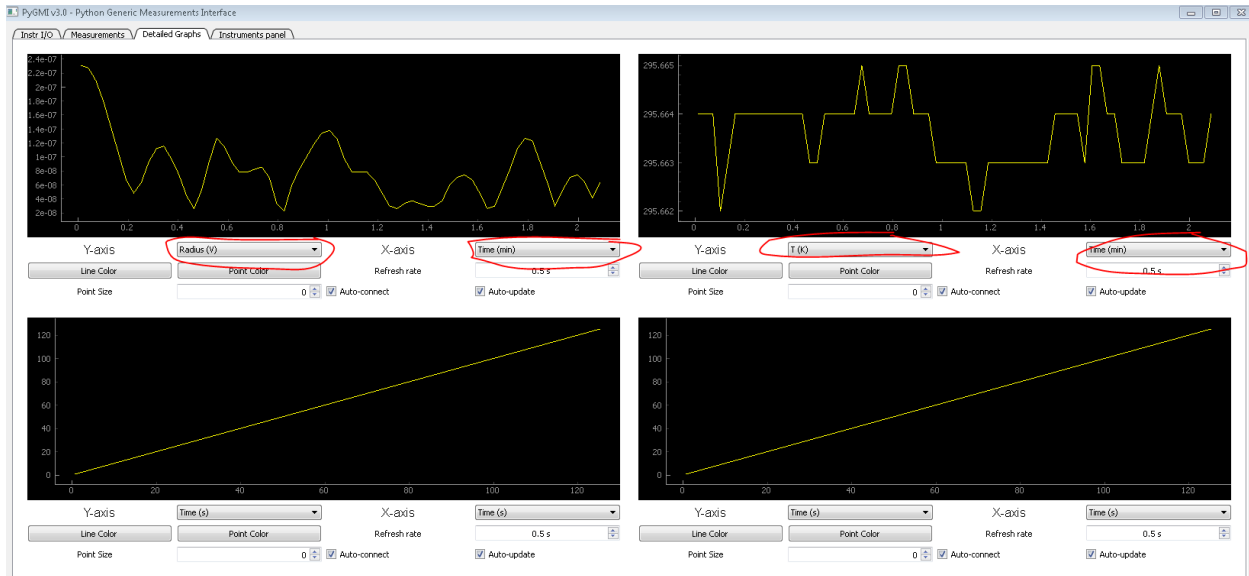
PyGMI makes measurements by running simple user-written short python script located in \PyGMI_files\Measurements_programs (see section Adding a new element to the User interface page 40 for more details on how to write such a script). This script will connect to the instruments defined in the first tab “Instr I/O” and run the measurements in a background process. Let’s enter a new data file name under “Save File”, for instance “measurements data/mytestdata.txt”, then let’s select “Demo_script” in the drop-down menu below “Measurements Scripts”, and finally press “Start measurements”.



The Start measurements button now switches to “Stop measurements”, and the Python Shell indicates that the measurements started.



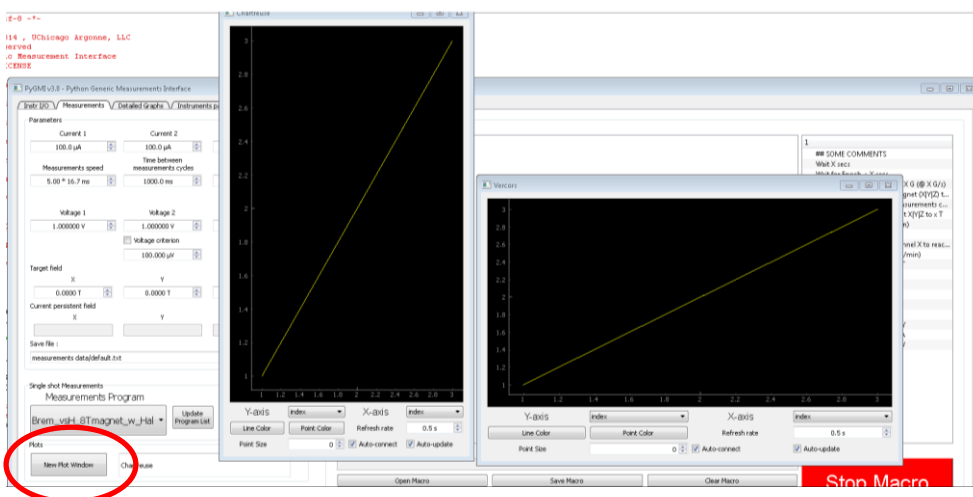
Let's now switch to the third tab "Detailed Graphs": four data plot windows appear. You can select the x and y for each plot. Let's select "Radius (V)" and "Time (min)" as y and x respectively for the first graph, and "Temperature (K)" and "Time (min)" for the second graph.



As you can see a live plot of the data is displayed. You can stop the live update by unchecking the box "auto-update"

NB: unchecking the box "auto-connect", will prevent the data inside that particular plot to be replaced with the new dataset when a new measurement is started.

NB: if four is not enough, you can create any number of independent plot windows that can be moved and rescaled, by pressing the button New Plot Window in the Measurements tab (see picture below).



Now go back to the Measurements tab, press stop measurements. The data that was taken is available in the file "measurements data/mytestdata.txt". (NB: the data is buffered and regularly written to the file, so that all is not lost even if the program or computer crashes in the middle of a measurement). So

let's open \PyGMI_files\Measurements_programs\Demo_script.py (right-click on it -> "Edit with Idle") to see what's in there.

```
#Multithreading
import threading
#Time measurement
import time

#####create a separate thread to run the measurements without freezing the front panel#####
class Script(threading.Thread):
    def __init__(self,mainapp,frontpanel,data_queue,stop_flag,GPIB_bus_lock,**kwargs):
        #nothing to modify here
        threading.Thread.__init__(self,**kwargs)
        self.mainapp=mainapp
        self.frontpanel=frontpanel
        self.data_queue=data_queue
        self.stop_flag=stop_flag
        self.GPIB_bus_lock=GPIB_bus_lock

    def run(self):
        #this is the part that will be run in a separate thread
        #####
        #SHORTCUTS
        m=self.mainapp          #a shortcut to the main app, especially the instruments
        f=self.frontpanel       #a shortcut to frontpanel values
        reserved_bus_access=self.GPIB_bus_lock    #a lock that reserves the access to the GPIB bus
        #data_queue=self.data_queue #a shortcut to a FIFO queue to send the data to the main thread
        #####
```

At the beginning, there is a bunch of generic command, to make instruments and interface entries available into the script. You should not have to modify these (but reading the rest of the manual, should make you understand why they are there). What's important is the remaining core of the script which is fairly simple to understand, and which is the only part you need to modify if you want to make a new script.

First, a header is created to initialize the save file, and then it is passed to the main program which handles saving, and live plotting data. We create as many items in the header as datapoints we are going to pass to the main program each time we take a measurement. So here it will consist of :

Time in seconds, Time in minutes, Temperature, radius and phase of the signal of the lock-in

```
#####
#SAVEFILE HEADER - add column names to this list in the same order
#as you will send the results of the measurements to the main thread
#for example if header = ["Time (s)","I (A)","V (volt)"]
#then you have to send the results of the measurements this way :
#"self.data_queue.put([some time, some current, some voltage],False)"
header=['Time (s)']
header+=['Time (min)']
if f.temp_controller_on:header+=["T (K)"]
if f.instr_on_1:header+=["Radius (V)","theta"]

#####
#ORIGIN OF TIME FOR THE EXPERIMENT
start_time=time.clock()
#####
#SEND THE HEADER OF THE SAVEFILE BACK TO THE MAIN THREAD, WHICH WILL TAKE CARE OF THE REST
self.data_queue.put((header,True))
```

NB: Here, the last three items of the header are created only if f.temp_controller_on and f.instr_on_1 are "True" (the Boolean value). They contain the True/False state of the checkbox next to the instrument address in the Instr I/O tab, so that the script can be used with differing numbers of instruments. They will appear again when taking measurements. This is not necessary, but just to show some of the capabilities that are available to you.

Then

- we initialize the source of the lock-in to the voltage value given by the entry “Voltage 1” on the frontpanel
- we check whether the main program is signaling to stop the script
- we lock access to the instruments bus and take some temperature and voltage measurements
- we pass this fresh data to the main program
- we check again whether the main program is signaling to stop the script
- And finally we wait some time (as indicated by the Time Between Measurements Cycle in the interface) and do it again

```
if f.instr_on_1:m.instr_1.set_amplitude(f.voltage1)

#####Control parameters loop(s)#####
while True:
    #Check if the main thread has raised the "Stop Flag"
    if self.stop_flag.isSet():
        break
    #reserve the access to the instruments, then discuss with them
    with reserved_bus_access:
        #Measure T
        if f.temp_controller_on:T=m.temp_controller.query_temp('B')
        #Measure R and theta
        if f.instr_on_1:freq,R,theta=m.instr_1.query_f_R_theta()

        #####Compile the latest data#####
        t=time.clock()-start_time
        last_data=[t,t/60.0]
        if f.temp_controller_on:last_data.append(T)
        if f.instr_on_1:last_data.extend([R,theta])

        #####Send the latest data to the main thread for automatic display and storage into the savefile#####
        self.data_queue.put((last_data,False))

    #Check if the main thread has raised the "Stop Flag"
    if self.stop_flag.isSet():
        break
    #####Wait mesure_delay secs before taking next measurements
    time.sleep(f.mesure_delay)
```

you can easily tweak that file if you want, for example let's take two measurements in a row, calculate the average, and return that value instead to the main program. Simply modify it as follow:

```
#reserve the access to the instruments, then discuss with them
with reserved_bus_access:
    #Measure T
    if f.temp_controller_on:T=m.temp_controller.query_temp('B')
    #Measure R and theta
    if f.instr_on_1:freq,R1,theta1=m.instr_1.query_f_R_theta()
    #Measure R and theta
    if f.instr_on_1:freq,R2,theta2=m.instr_1.query_f_R_theta()
R=(R1+R2)/2.0
theta=(theta1+theta2)/2.0
```

Now, save the changes to the file and press the button “Update Program List”, the script will be reloaded. Then you can start again and see if that improves the noise level.

Single shot Measurements

Measurements Program

Demo_script

Update Program List

Start Measurements

Plots

New Plot Window

New Plot Window Title

E-mail alert address

Config Menu

Current line in the macro

Open Macro

Run Macro

Sometimes, you may want to run such a measurement script lots of time with slightly different parameters. Technically it can be all done in the script, which is perfectly OK. But as an extra layer of convenience, a more readable Macro system with syntax highlighting is provided.

We will now write and run a Macro that calls the measurement script above while ramping the temperature. Click in the editor in the middle of the Measurements tab. You can now type text with the keyboard. By default the text you type will appear in red because the system will probably not recognize the command you entered. On the right of that panel you can see the list of available valid commands (See Creating a New Macro Command for more details).

For now all we need is to set the voltage of the source of the lock-in, so type:

Set voltage 1 to 1e-1 V

As soon as the command is recognized, it turns to black, with the command parameters values highlighted in bold blue. Then we need to specify the save file, so type

Set Save file to measurements data/demo-0.1V.txt

Then we want to start the measurements, so

Start Demo_script()

Then, assuming the temperature right now is 300 K, we will be ramping the temperature from 295 K to 300 K using loop 1 (main control loop/output of Lakeshore340) using the 4th heater range of the temperature controller (0 is OFF, 5 is the most powerful heater range on a Lakeshore340) , and we will monitor the temperature on channel 'B' thermometer. As soon as the thermometer reading is within 0.1 K of 300K, the measurement script will be stopped and the macro will end.

Set heater range to 4

Set Loop 1 to 295 K

Set Loop 1 to 300 K @ 0.5 K/min

Wait for channel B to reach 300 +/- 0.1 K

Stop Measurements

Finally we copy paste this set of instructions and tweak it so as to run the same measurement at 0.5V.

```
Macro Editor

Set voltage 1 to 1e-1 V
Set Save file to measurements data/demo-0.1V.txt
Start Demo_script()
Set heater range to 4
Set Loop 1 to 295 K
Set Loop 1 to 300 K @ 0.5 K/min
Wait for channel B to reach 300 +/- 0.1 K
Stop Measurements

Set voltage 1 to 0.5 V
Set Save file to measurements data/demo-0.5V.txt
Start Demo_script()
Set heater range to 4
Set Loop 1 to 295 K
Set Loop 1 to 300 K @ 0.5 K/min
Wait for channel B to reach 300 +/- 0.1 K
Stop Measurements
```

That's it, we can now save the Macro in a text file, so as to reuse it later: just click on Save Macro, and a pop-up window will ask you for a ".mac" destination file. As you probably guessed, Open Macro, and Clear Macro do what they say. Finally, to start the macro press "Run macro".

NB1: To abort the Macro, press the red button Stop Macro. Whether you are running a macro or a single measurement, pressing this button should stop all scripts.

NB2: if you quit the program before stopping all scripts with either the red button or the "stop measurements" button, and if a script was running, then this script will keep running in the background. In that case, under windows you probably will have to open the task manager (Ctrl+Alt+Delete>Start Task manager, or look for "Task Manager" in the start menu if you are using a remote desktop connection), go to "Processes", and look for processes called "pythonw.exe", select them and then click "End process".

NB3: some predefined macro commands accept optional arguments, they appear inside parenthesis "()" in the list of valid command. However, to use such a command in a macro, the parenthesis should be removed. Otherwise the command won't be recognized. But, of course the syntax for all commands can be modified as these are just simple "regular expressions", see Creating a New macro command for more details.

Detailed sections

Architecture of the program

The core of the program is located in \PyGMI_files, this folder is organized as follow:

The folder itself contains all the files related to the graphical user interface as well as the main logic of the program. The most likely files that you may want to modify are “Graphical_User_Interface.ui” and “Instruments_connection.ui” which contain the graphical interface design. Use Qt designer to open, modify and save your changes. The compilation will be done automatically at the next restart of PyGMI. See Adding a new element to the User interface.

Then the subfolders contain the following files:

\Instruments

This folder contains one .py file per instruments (and also an auto compiled version ending in ‘.pyc’). This is where you may add a new file for a new type of instrument. See the Creating a New Instrument Driver

\Measurements_programs

This folder contains one .py file per measurements program (and also an auto compiled version ending in ‘.pyc’). This is where you may add a new program of your own. See Creating a new program of measurements

\Instruments_panels

This is where you may add a new instrument panel for direct graphical interaction with the instrument in the fourth tab “Instruments panel” of the program. See Creating a New Panel

Configuration menu

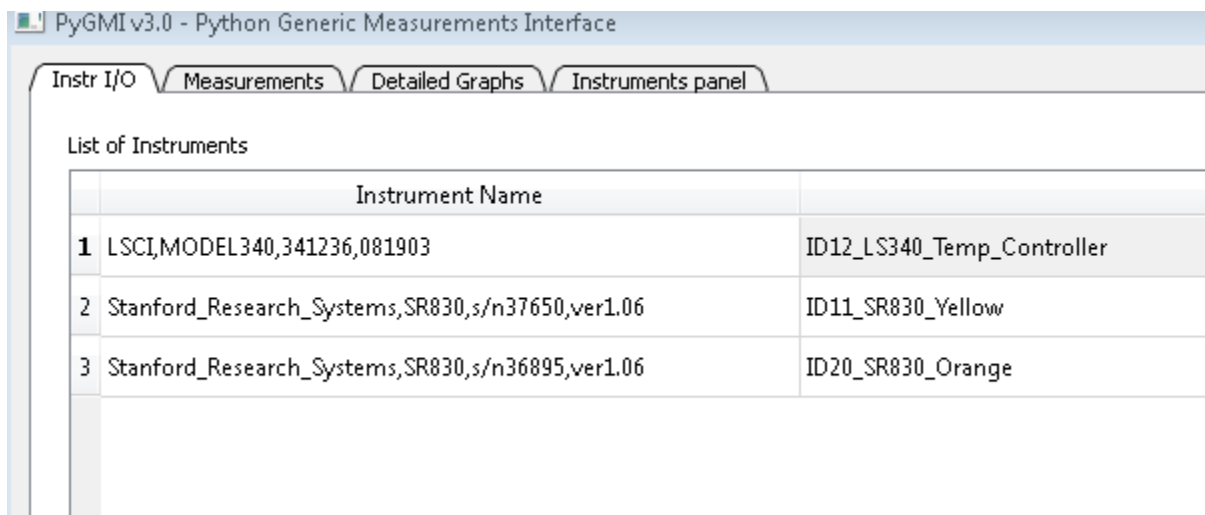
On the measurements tab, the config menu button brings up a short configuration menu where you can configure the default folders or the default appearance of the plot windows, and this is also where the smtp server and login/password can be configured if you want to use the emailing commands in the Macro editor.

NB: Be warned that, even though the password is not displayed in the interface when you type it, it will not be stored securely at all since it will be stored in clear in the file

\Configuration\Parameters\CurrentParameters.cfg

Configuring the instruments connections

In the first tab “Instr I/O”, press the button “Scan for instruments”. This will scan for instruments connected to the computer, which can take a few seconds, or more if some instruments are unresponsive and depending on the connection timeout. It will then display a list of valid visa addresses in the “Instrument List” table.



Instr I/O Measurements Detailed Graphs Instruments panel		
List of Instruments		
	Instrument Name	
1	LSCI,MODEL340,341236,081903	ID12_LS340_Temp_Controller
2	Stanford_Research_Systems,SR830,s/n37650,ver1.06	ID11_SR830_Yellow
3	Stanford_Research_Systems,SR830,s/n36895,ver1.06	ID20_SR830_Orange

The text entries, checkboxes and drop-down menus are used to configure the connections to the instruments. For example, if you want the name “instr_8” in the measurement script to be connected to the Keithley 2182 at GPIB address “GPIB0::11” (card 0, address 11) then you must enter that address in the text entry below the label “Instr 8” and check the checkbox to its left so as to indicate that you want to connect this instrument. Finally choose the type K2182 in the drop-down menu to specify the instrument type. When you have configured and checked all the instruments you want to connect, click on “Initialize instruments”.



Instr8

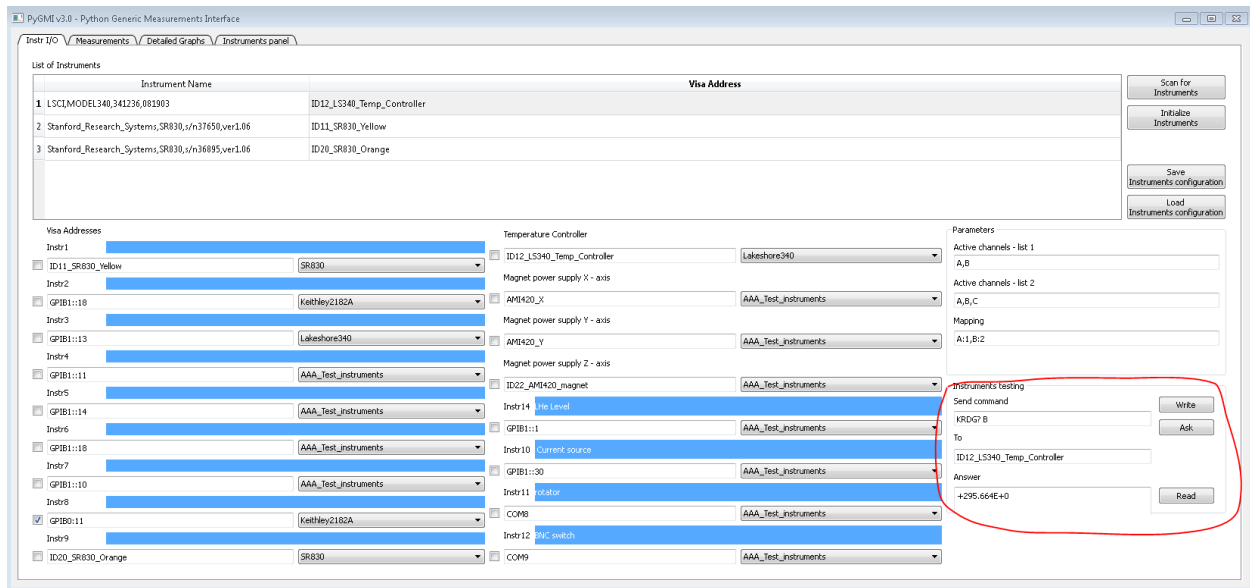
☒ GPIB0:11 Keithley2182A

NB: custom visa addresses and GPIB addresses can be used indistinctively in these textboxes

NB2: you can double click on, and then copy-paste, the addresses that appear after “Scan for Instruments”

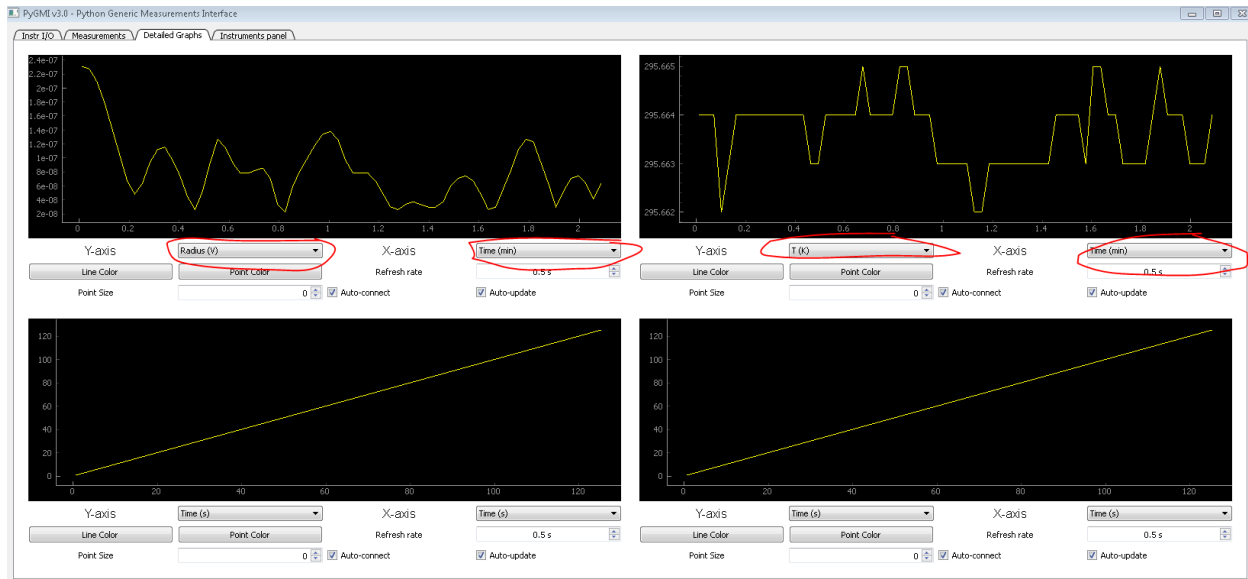
The configuration of all these instruments can be saved to a text file by pressing “Save Instruments configuration”. Similarly, the configuration can be loaded from a text file by pressing “Load Instruments configuration”.

NB: the current instruments configuration will be saved to a default config file once you pressed “Initialize Instruments”, if it succeeds, so that the next time you relaunch the program it will open with the last known working configuration.



NB: you can use the small interface on the bottom right hand side to try standard (SCPI GPIB etc...) commands on the instruments

Plotting data



Once the measurement script has been started, the X and Y drop-down menu of all plot windows will be populated with the same header as in the savefile (provided “auto-connect” is checked). Just select the X and Y that you want and the plots will appear.

For each plot, the line color, point color, refresh rate and point size (zero=no points) can be modified. Their default values can be changed through the configuration menu on the Measurements tab.

The Auto-connect checkbox selects whether you want the plot and the XY drop-down menus to be updated as soon as a new measurement is started (erasing the previous plot in the process), or whether you want to keep the plot even when after a new measurement started (in which case you will still be able to look at the various columns of the retained data using the X and Y menus)

The Auto-update checkbox selects whether you want to keep adding freshly measured datapoints to the plot, or conversely pause the update of the plot (in which case datapoints are still being measured, and they can be added later when you resume the update of the plot)

Holding the left or right click buttons on a plot while moving the mouse allows respectively panning and magnifying the plot. Right clicking on a plot will also bring the pyqtgraph drop-down menu to tweak the axes (autoscale, log scale etc...)

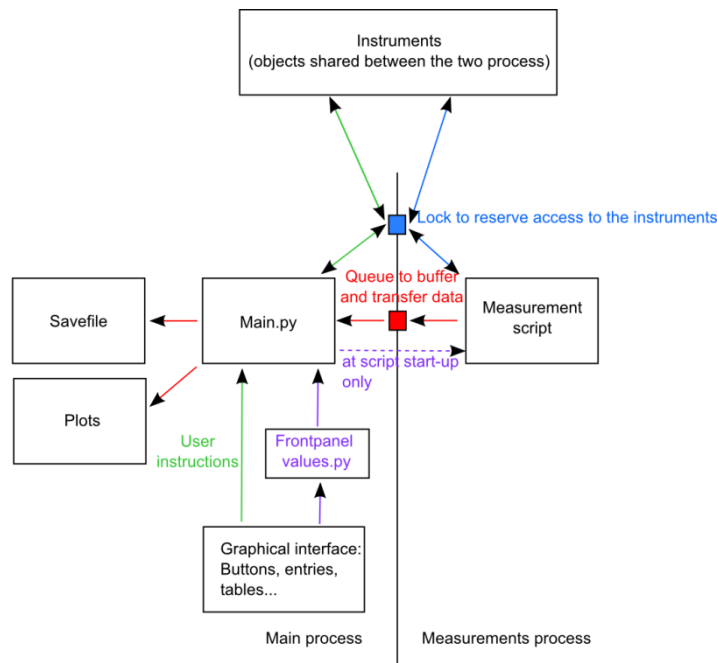
How the program works

The Main program

- 1) Instruments_connection.py handles the connection and initialization of the instruments
- 2) Frontpanel_values.py fetches and records all the values of the front panel elements (text entries, tables, numeric entries etc...)
- 3) Main.py starts a separate process to run the measurements script selected by the user and it also provides all the values that it fetched from the front panel to that script (the measurements process will not be aware of any changes made to the front panel after it started, except for the STOP Macro and STOP Measurements buttons. It's not impossible to implement, but it's a choice).
- 4) Main.py also handles plotting the data sent by the measurements process, and saving this data to a text file

The Measurements process

- 1) Communicates with the instruments, following what the user told him to do in the corresponding .py file, like : set the current to this, wait 1 s, measure the voltage
- 2) Send the results of the measurements back to the Main thread
- 3) Loop until told to stop by the main thread or having reached the last command of the measurements script



Creating a New Instrument Driver

- 1) Make a copy of the file "New_Instrument_template.py" and place it in the folder

"\PyGMI_files\Instruments"

- 2) Rename your copy of the file with the name that you want while leaving the ".py" extension, but an important rule must be followed: the name of the file should be a valid Python variable name. It means that the name you choose can contain only the following characters: a-z/A-Z/1-9 and the underscore sign "_". It also must not contain any whitespace " " and must not start with a digit 1-9.

Example of acceptable names:

- Keithley2182A.py
- My_dummy_instrument.py

Example of invalid names:

- 2182A_Keithley.py (*starts with a digit*)
- Keithley 2182A.py (*whitespace in the middle of the name*)

An Instrument file consist of three parts :

Normally, you don't need to modify the first part. Those are generic commands used to connect to the instrument. In some very special case you may need to modify the line "visa.instrument(VISA_address)" to add appropriate communication parameters such as termination characters, or bit parity. Refer to the Pyvisa documentation for available communication options, and to the instrument manual for the appropriate values of those parameters. But most of the time, especially with GPIB, USB and ethernet it works right out-of-the-box.

```
# -*- coding: utf-8 -*-
import visa,time

class Connect_Instrument():
    def __init__(self,VISA_address="GPIB1::22"):
        #part to be run at instrument initialization, the following commands are mandatory
        self.io = visa.instrument(VISA_address)
        self.VISA_address=VISA_address
        print self.query_unit_Id()
```

A Serial/RS232 connection will use PySerial instead of Pyvisa, so it has a slightly different beginning and it may require some connection parameters. For instance, in the driver for a VXM stepper motor:

```
import serial
#import time
class Connect_Instrument():
    def __init__(self,COM_port="COM3"):
        #Serial(N) open COM port N+1, e.g. "2" for "COM3"
        self.io = serial.Serial(float(COM_port.strip("COM"))-1,baudrate=9600, bytesize=8, parity='N', stopbits=1)
        self.io.write('F,C')
        #"F" puts the VXM On-Line
        #"C" clear the previous Index command from the VXM's memory
        ..
```

The following part of the program contains two mandatory functions that are called by the main process and that you may have to modify

```
#mandatory function
#return instrument identification. This command is common to almost all GPIB instruments. Modify if necessary.
def query_unit_Id(self):
    return self.io.ask("*IDN?")

#mandatory function that will be called just after the computer successfully connected to the instrument.
#If you don't need it, just leave it empty but do leave the command "return 1" or "pass"
def initialize(self):
    ###your initialization commands
    self.io.write("set:type:voltmeter") #dummy example in pseudo SCPI language
    print "setting the instrument to voltmeter" #dummy information to the user
    ###
    return 1
```

The last part of the file is the functions that you want to use in the measurements scripts for that type of instrument. Any function name is valid as long as it is a valid Python variable name (see rules above). The typical function used to send and read instructions to an instrument are

self.io.ask('some text') : send the command 'some text', wait for the answer and return it

self.io.write('some text') : send the command 'some text'

self.io.timeout= 5 : set the timeout for 'ask' and 'read' commands to 5 seconds

self.io.read() : returns a string sent from the instrument to the computer

self.io.read_values() : returns a list of decimal values (floats) sent from the instrument to the computer.

self.io.ask_for_values('some text') : send the command 'some text', wait for the answer and return the answer as a list of values, just as read_values() does

For further details on these commands, refer to Pyvisa documentation.

NB: Instrument language

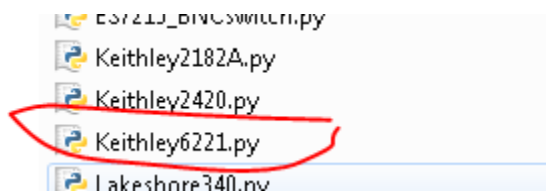
The commands to be sent to the instrument are normally found in the user manual. The SCPI language is an attempt to unify those commands, but beware of subtle variations between similar instruments.

Creating a New Instrument Panel

In this section we will see how to create a functioning instrument panel. We will take the example of a Keithley 6221 current source.

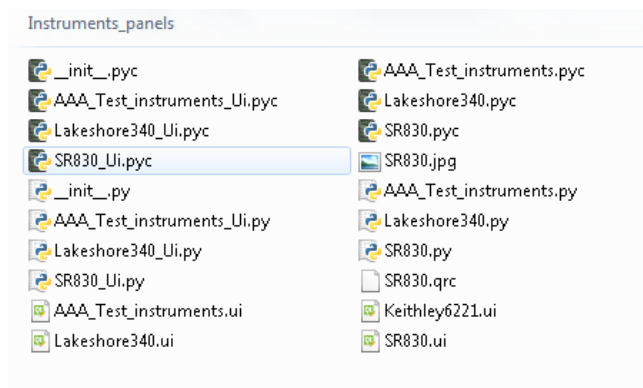
The first thing we need to do is to make sure that an instrument driver already exists for that type of instrument.

Go to \PyGMI_files\Instruments



As you can see Keithley6221.py is already there. (Refer to the previous section to see how to create a New Instrument Driver)

Now let's go to \PyGMI_files\Instruments_panels



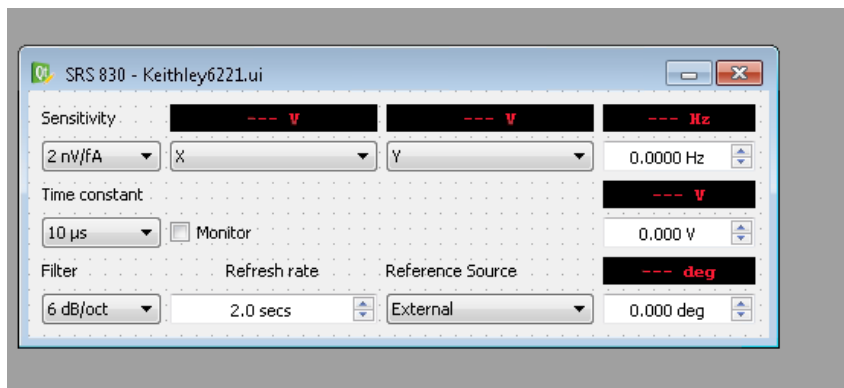
In that folder we will need to create two files :

- 1) Keithley6221.py
 - A file, with the exact same name as the instrument Driver (mandatory). It will handle which commands to send to the instrument when the user click on a button of the instruments panel
- 2) Keithley6221.ui
 - A QtDesigner file that will contain the design of the graphical interface of the panel

Let's start with the interface

For a quick start let's copy-paste SR830.ui, then rename it Keithley6221.ui

Double-click on it to open it with QtDesigner



The panel design appears in the center.

First let's update the window title: Property editor > QWidget>windowTitle

Panel : QWidget	
Property	Value
▸ QObject	
objectName	Panel
▸ QWidget	
windowModality	NonModal
enabled	<input checked="" type="checkbox"/>
▸ geometry	[(0, 0), 486 x 162]
▸ sizePolicy	[Preferred, Preferred, 0, 0]
Horizontal Policy	Preferred
Vertical Policy	Preferred
Horizontal Stretch	0
Vertical Stretch	0
▸ minimumSize	0 x 0
▸ maximumSize	16777215 x 16777215
▸ sizeIncrement	0 x 0
▸ baseSize	0 x 0
palette	Inherited
▸ font	A [MS Shell Dlg 2, 8]
cursor	Arrow
mouseTracking	<input type="checkbox"/>
focusPolicy	NoFocus
contextMenuPolicy	DefaultContextMenu
acceptDrops	<input type="checkbox"/>
▸ windowTitle	Keithley 6221
▸ windowIcon	
windowOpacity	1.000000

Which updates the window title

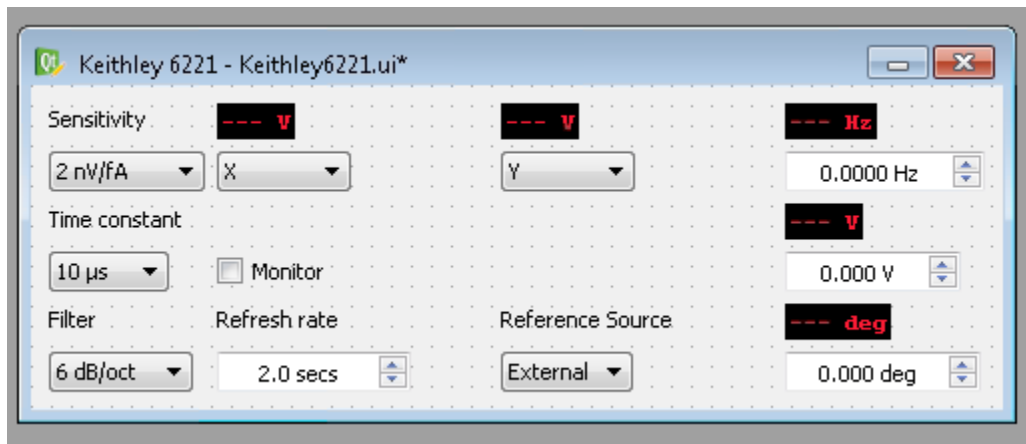


For now, here is a summary of the simple design we will try to make

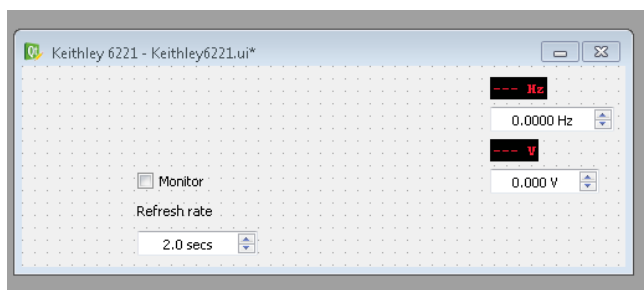
- Display
 - current setpoint
 - voltage compliance
 - on/off state
- Set
 - Current setpoint
 - Voltage compliance
 - On/off state
 - Reset instrument

So we will need three display fields and four input fields

First, select the widget (click on its window titke) then break the layout of the widget

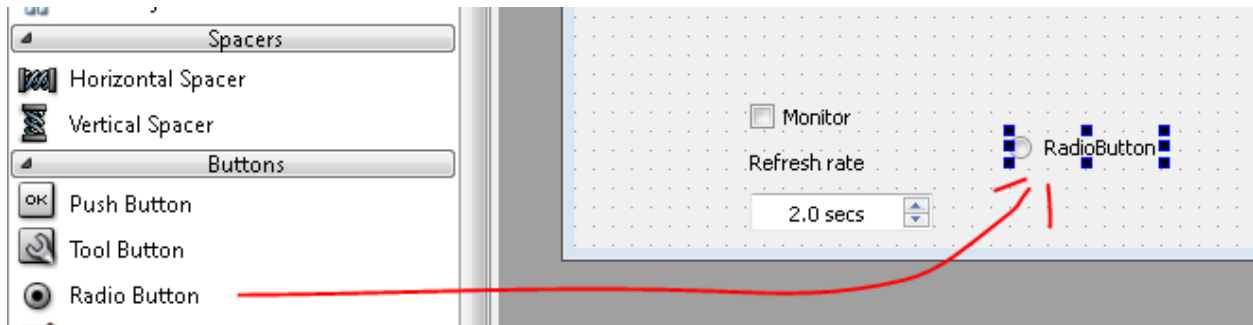


Now let's remove Sensitivity, Time constant, Filter, X, Y, reference source and deg: select each element and its label and press "Delete".

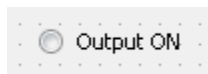


We keep Monitor and refresh rate as well as two entries and two displays, so we just need one on/off button and a Reset button

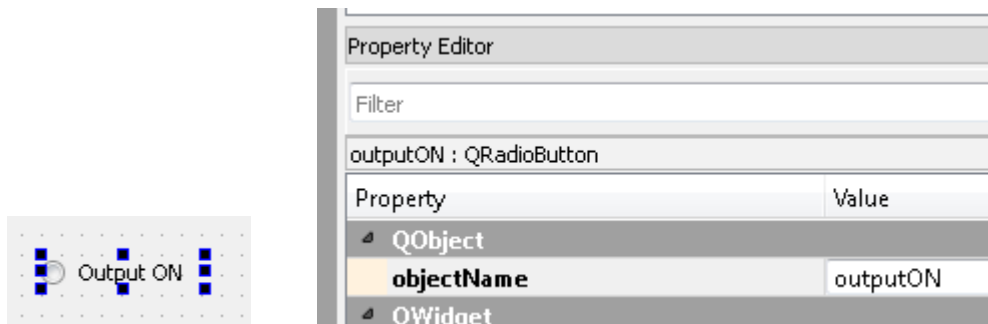
Drag and drop a radio button



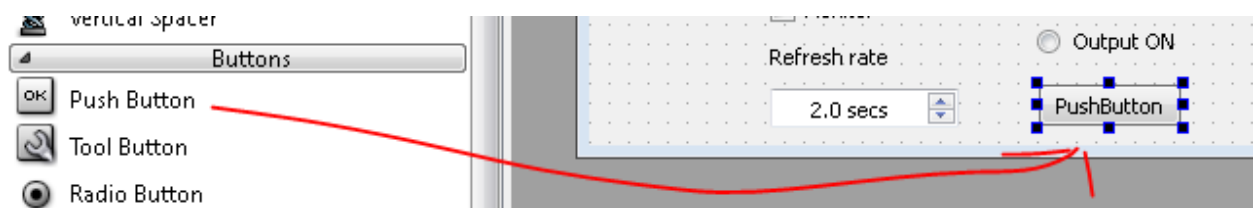
Rename the label Output ON, by double clicking on it



Then select it (one click) and change the objectName to outputON (a valid Python name)



Now drag and drop a push button



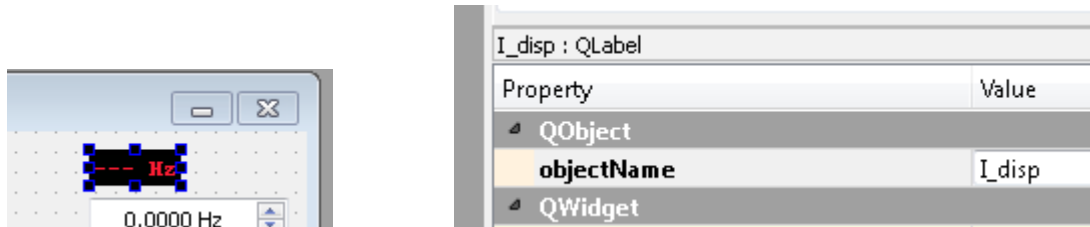
Double click on it to change the label to "Reset"



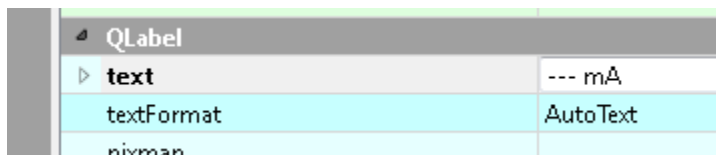
In the property editor, change its objectName to "resetbutton"

Now, for the current setpoint and voltage compliance, let's make our entry fields with displays

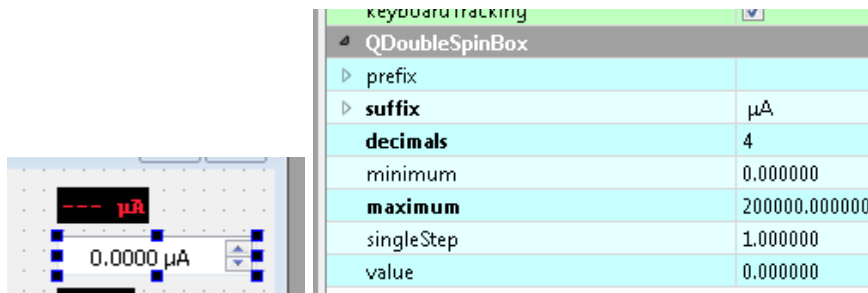
Select the first red/black display, change objectName to I_disp



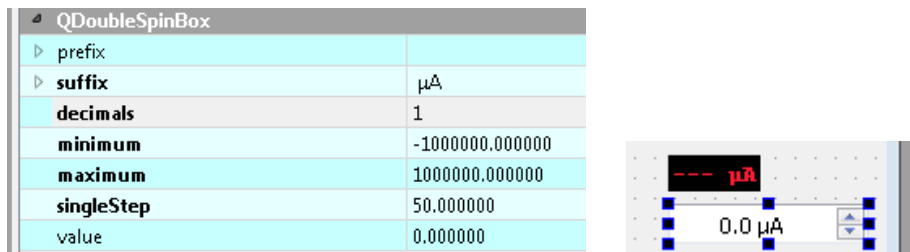
Scroll down the property editor to text, and change it to "--- μ A" (any unicode character should work)



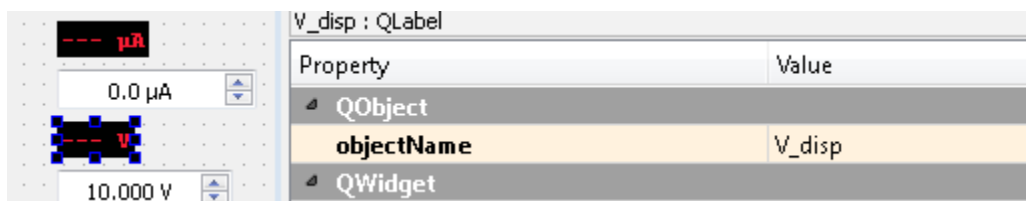
Select the first double spin box, scroll down the property editor to suffix, and change it to " μ A"



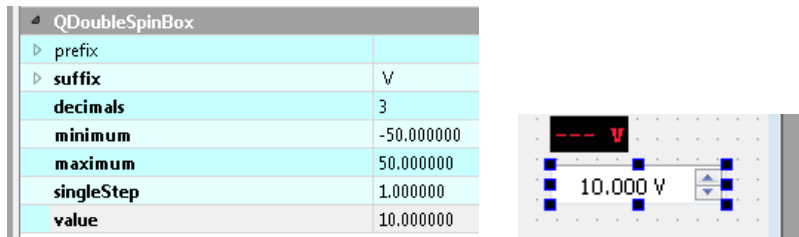
Now let's edit the limits for the values that can be entered



Then do the same for the second display: change its objectName to V_disp



And edit the properties of its corresponding double spin box

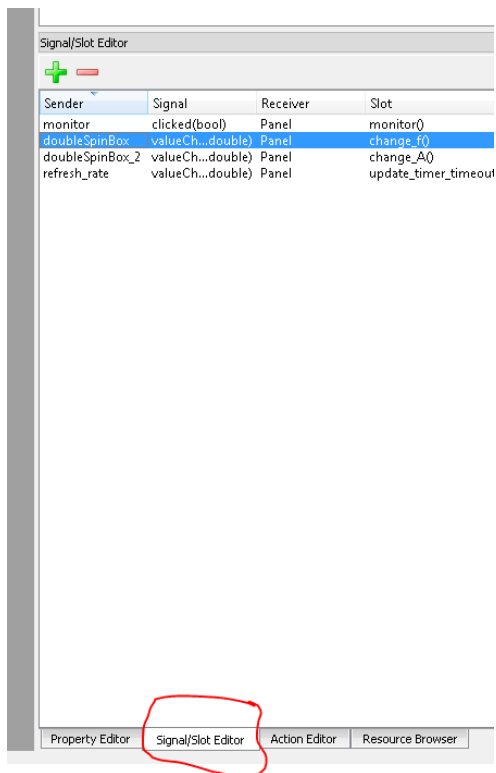


With this, we have all the buttons and displays we wanted. So, now we must connect them to the logic of the program.

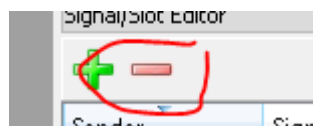
Click on Edit Signal/Slots



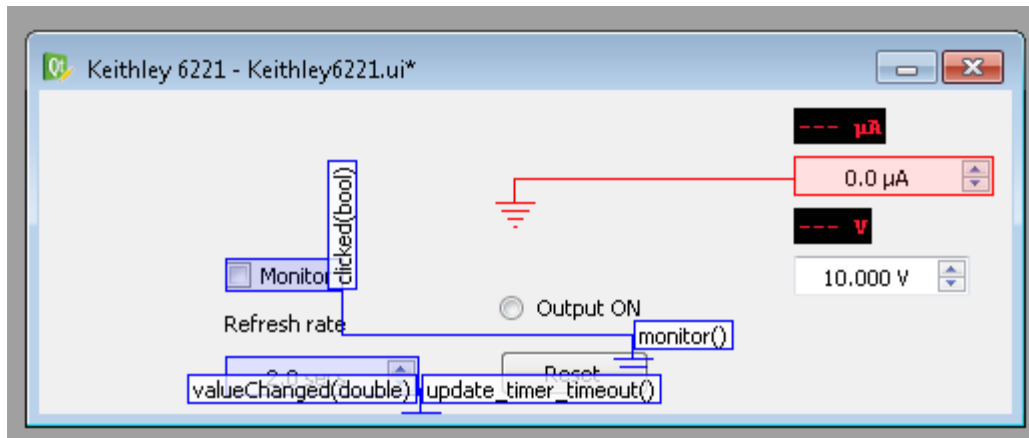
Go to the signal/slot editor



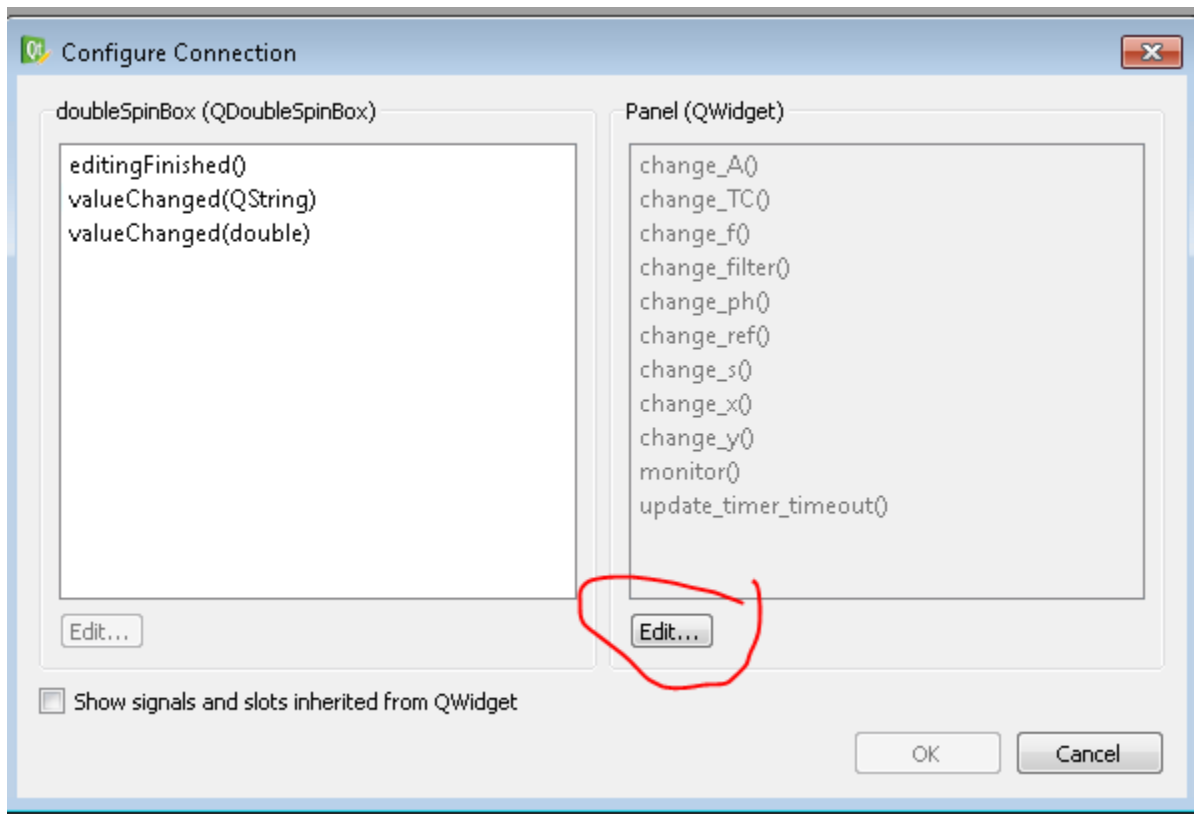
Select doubleSpinBox and press the minus sign to delete it, do the same with doubleSpinBox_2.



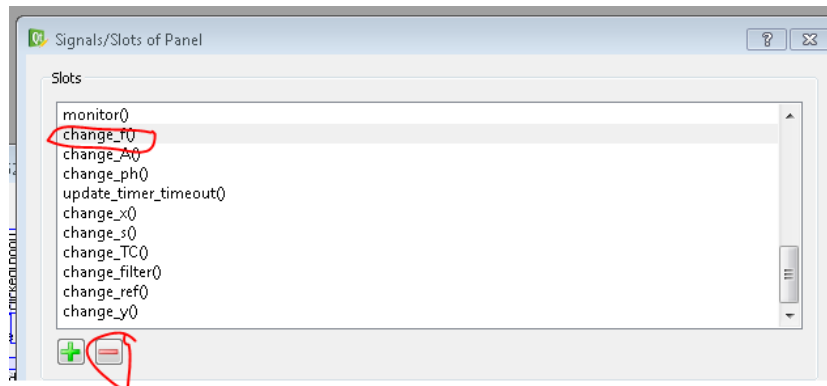
Now select the doublespinbox for the current setpoint, then hold the click and drag your mouse to the side: a red line with a “ground-like” extremity appears.



Release the click and a pop-up menu appear, choose Edit...



A new popup menu appear, let's remove all the functions that we will not implement nor use, i.e. all the ones that start with “change_...”



Then let's add our own functions we will have :

`change_I()`

A function to update the current setpoint

`change_V_comp()`

A function to update the voltage compliance

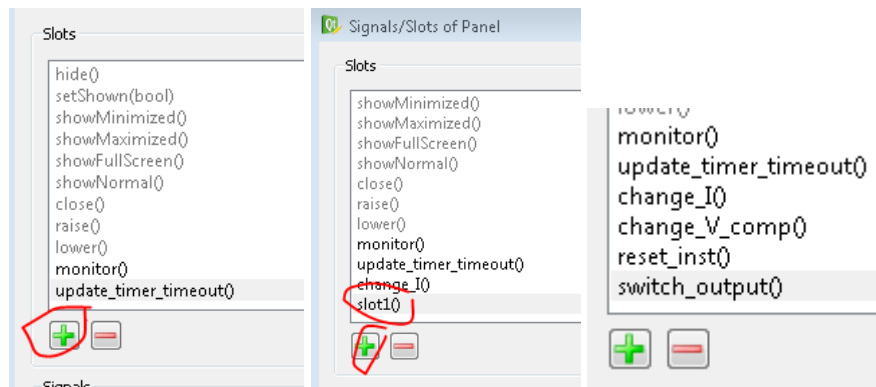
`reset_inst()`

A function to reset the instrument to factory default parameters

`switch_output()`

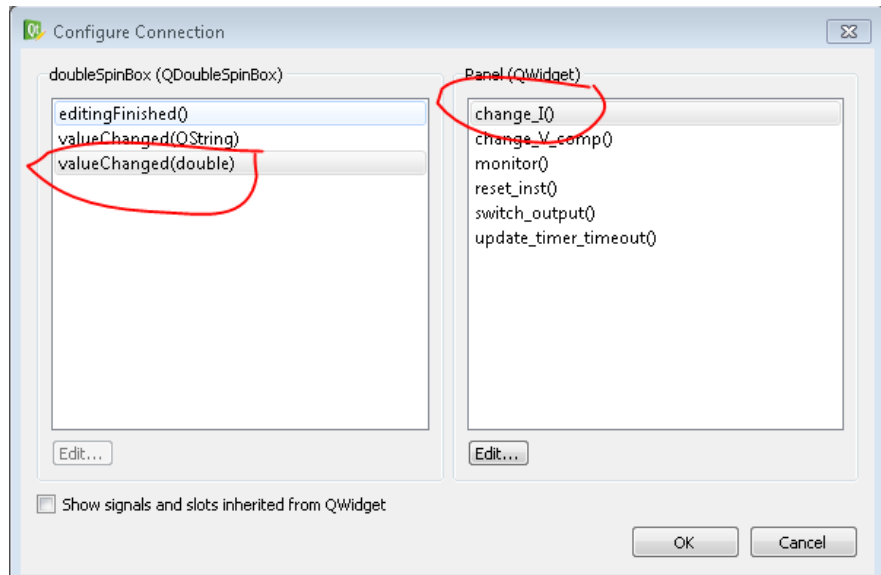
A function to change the output state

Click on the green plus sign to add these four new functions (double click on a function name to edit it) and don't forget the double parentheses at the end "()"



Then press OK, and we are back to the previous menu. Now we need to pick one action in the left column and choose what functions it will trigger in the right column. For instance, we want the function (for the serious reader, read "method") `change_I()` to be called when the value of the `doubleSpinBox` is



changed by the user. For this, we select “valueChanged(double)” on the left and “change_I()” on the right



Then press OK

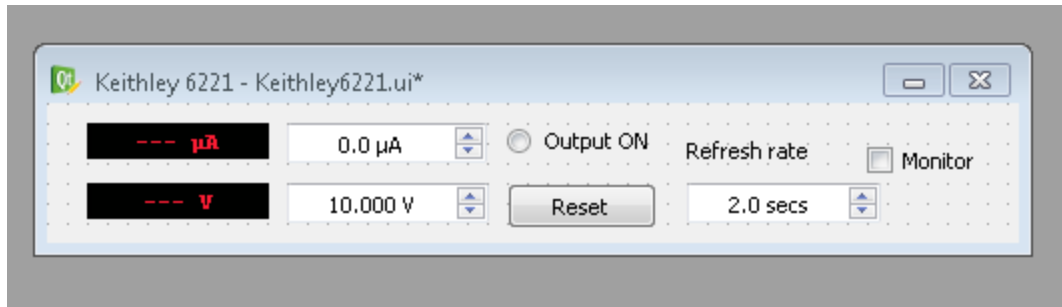
That's it, our first connection to the logic of the program is done.

Now we have to repeat these steps for the other boxes, so that we end up with the following connections in the signal/slot editor

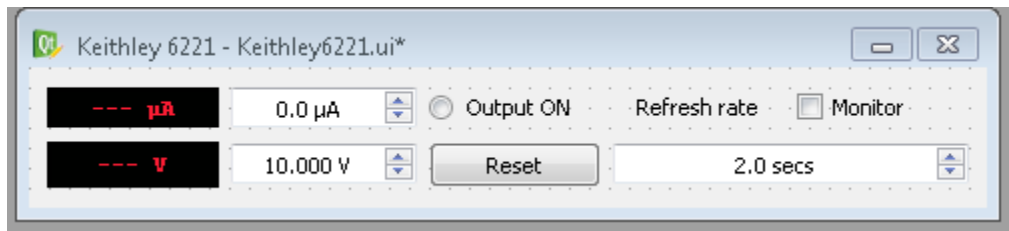
Signal/Slot Editor			
 			
Sender	Signal	Receiver	Slot
monitor	clicked(bool)	Panel	monitor()
refresh_rate	valueChanged(double)	Panel	update_timer_timeout()
doubleSpinBox	valueChanged(double)	Panel	change_I()
doubleSpinBox_2	valueChanged(double)	Panel	change_V_comp()
outputON	clicked(bool)	Panel	switch_output()
resetbutton	clicked()	Panel	reset_inst()

As a final cosmetic step, we go back to Edit Widgets mode and move and resize the elements a little bit





Then select the widget window itself and press Lay Out in a Grid



Now the elements sizes will be updated when the window is resized. Refer to QtDesigner documentation for further tweaking.

That's it for the interface design! Save it and exit QtDesigner.

Now we need to create the logic behind this interface:

- go to \PyGMI_files\Instruments_panels
- Copy-paste SR830.py and rename the copy as Keithley6221.py
- Open Keithley6221.py with your favorite editor

```

File Edit Format Run Options Windows Help
from PySide.QtGui import QWidget,QApplication
from PySide.QtCore import QTimer
#import the interface design generated by Qt designer
import SR830_Ui

class Panel(QWidget):
    def __init__(self,parent=None,instr=None,lock=None,title='Inst:
        # This class derives from a Qt Widget so we have to call
        # the class builder ".__init__()"
        QWidget.__init__(self)
        # "self" is now a Qt Widget, then we load the user interface
        # generated with QtDesigner and call it self.ui
        self.ui = SR830_Ui.Ui_Panel()
        # Now we have to feed the GUI building method of this object
        # with the current Qt Widget 'self', but the widgets from
        # of the object self.ui
        self.ui.setupUi(self)
        self.setWindowTitle(title)
        self.reserved_access_to_instr=lock
        self.instr=instr
        self.monitor_timer = QTimer()
        self.channel=self.ui.channel.currentIndex()
        #the timer would not wait for the completion of the task o
        self.monitor_timer.setSingleShot(True)
        self.monitor_timer.timeout.connect(self.monitor)
        self.firsttime=0
        #bug: if the box is checked in the .ui file, the system fr
        #if self.ui.monitor.isChecked():self.monitor()

```

This is the back panel where we will define the functions that we connected our buttons to in the interface designer, and we also need to update a few references

- 1) First import Keithley6221_Ui

```

from PySide.QtGui import
from PySide.QtCore import
#import the interface des
import Keithley6221_Ui

```
- 2) And correspondingly

```

# "self" is now a Qt widget, then we load the
# generated with QtDesigner and call it self.
self.ui = Keithley6221_Ui.Ui_Panel()
# Now we have to feed the GUI building method

```
- 3) And we also delete the line referring to a now deleted box

```

self.channel=self.ui.channel.currentIndex()

```
- 4) Then we can delete the method "update_boxes"
- 5) We keep monitor(), but edit its content so as to update its displays with the values from the instrument (refer to the Keithley6221.py driver to see what these query functions do)

```

def monitor(self,state=1):
    if state!=1:
        self.monitor_timer.stop()
    elif state and not (self.monitor_timer.isActive()):
        with self.reserved_access_to_instr:
            I=self.instr.query_current_source_amplitude()
            Vcomp=self.instr.query_voltage_compliance()
            outstate=self.instr.query_output_ON()
            self.ui.I_disp.setText(str(I*1e6)+u' μA')
            self.ui.V_disp.setText(str(Vcomp)+' V')
            self.ui.outputON.setChecked(outstate)
            self.monitor_timer.start(self.ui.refresh_rate.value()*1000)

```

(NB: notice the `u' μA'` to declare a unicode string for the mu character)
- 6) Finally, we create the new functions that we have just announced and wired to various buttons in QtDesigner, i.e. : `change_I()`, `change_V_comp()`, `switch_output()` and `reset_inst()`. The content

of these methods is straightforward, as the driver already has all the required methods. Don't forget to protect the commands sent to the instruments with a `self.reserved_access_to_instr` block.

Here is what we told QtDesigner

doubleSpinBox	valueChanged(double)	Panel	change_I()
doubleSpinBox_2	valueChanged(double)	Panel	change_V_comp()
outputON	clicked(bool)	Panel	switch_output()
resetbutton	clicked()	Panel	reset_inst()

And here is what our back panel contains

```
def update_timer_timeout(self,secs):
    #The value must be converted to milliseconds
    self.monitor_timer.setInterval(secs*1000)

def change_I(self,value=0):
    with self.reserved_access_to_instr:
        self.instr.set_current_source_amplitude(value*1e6)

def change_V_comp(self,value=0):
    with self.reserved_access_to_instr:
        self.instr.set_voltage_compliance(value)

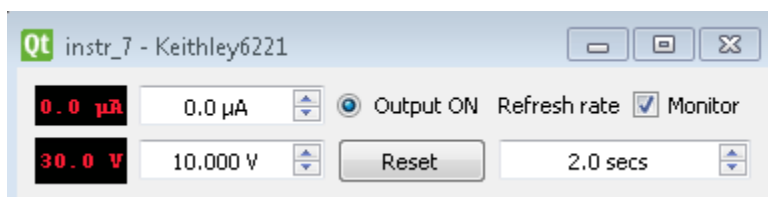
def switch_output(self,value=False):
    if value:
        with self.reserved_access_to_instr:
            self.instr.output_ON()
    else:
        with self.reserved_access_to_instr:
            self.instr.output_OFF()

def reset_inst(self):
    with self.reserved_access_to_instr:
        self.instr.reset()

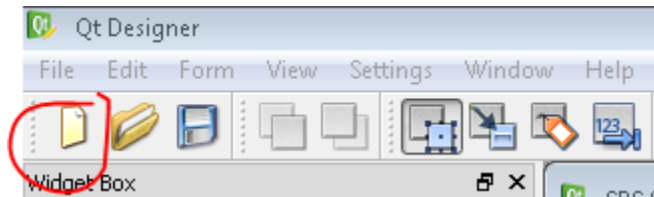
if __name__ == "__main__":
    import sys
    app = QApplication(sys.argv)
    window = Panel(app)
    window.show()
    sys.exit(app.exec_())
```

NB: when the user changes the value in the double spin box connected to `change_I()`, a signal `valueChanged(double)` is emitted, which will be passed to the method `change_I()` along with the corresponding new value "double". So in the definition of `change_I` we announced the expected double value: `def change_I(self,value)`. The same is true of the other types, such as boolean values or text entry.

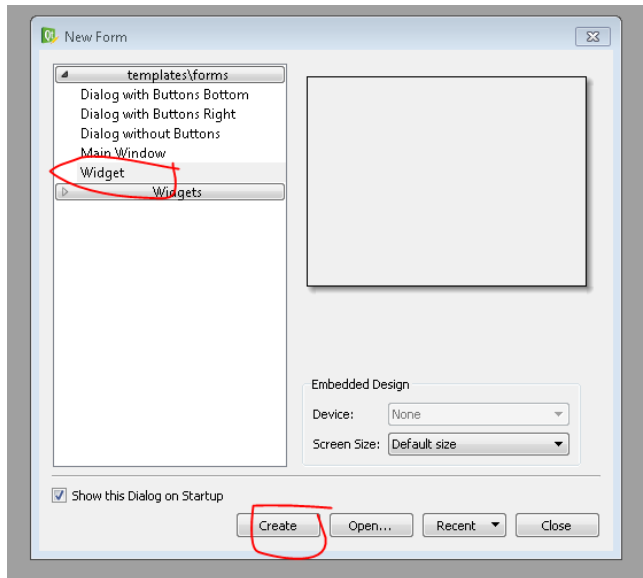
Save, exit and restart PyGMI and that's it we have a new functioning instrument Panel for every K6221 we connect



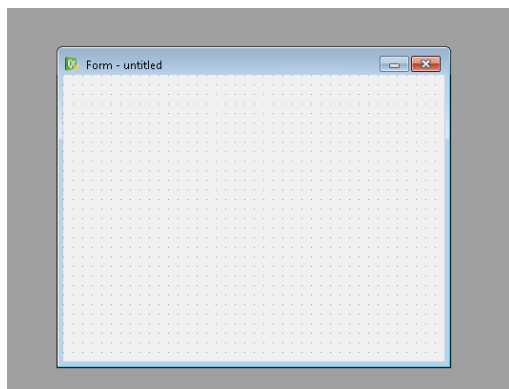
NB: If you prefer to start the design from scratch : Open QtDesigner> click on New



Select Widget, then Create

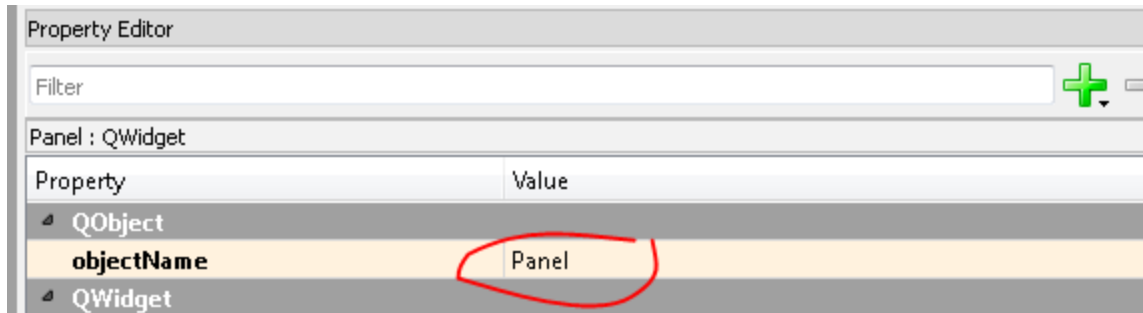


A blank form will appear in the center that just need to be filled with buttons and entries



The only required step is to modify the name of this widget to "Panel"

Go to Property editor>QObject>objectName

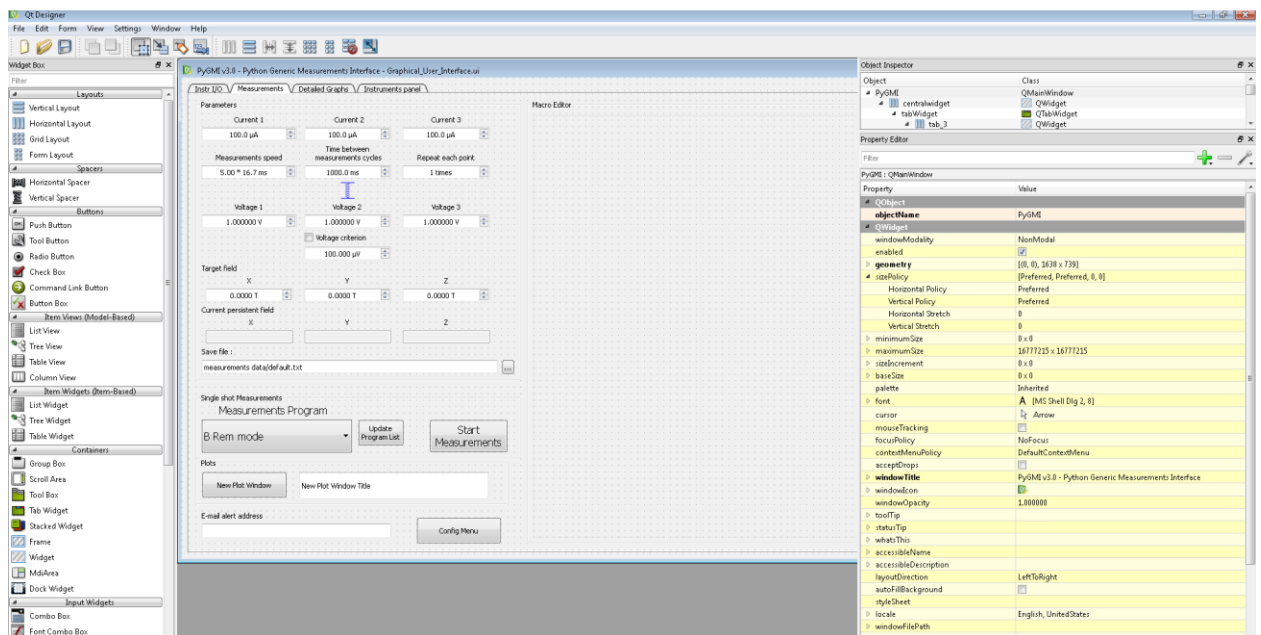


The widget has to be named “Panel” because the program expects the instrument panels files to contain such a class called “Panel”, which will be used to add the instrument panel to the PyGMI interface.

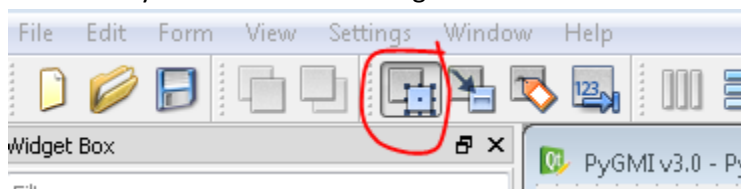
Adding a new element to the User interface

In this section we will add new entries in the graphical interface. For instance we will add three boxes to input a series of angles for a stepper motor.

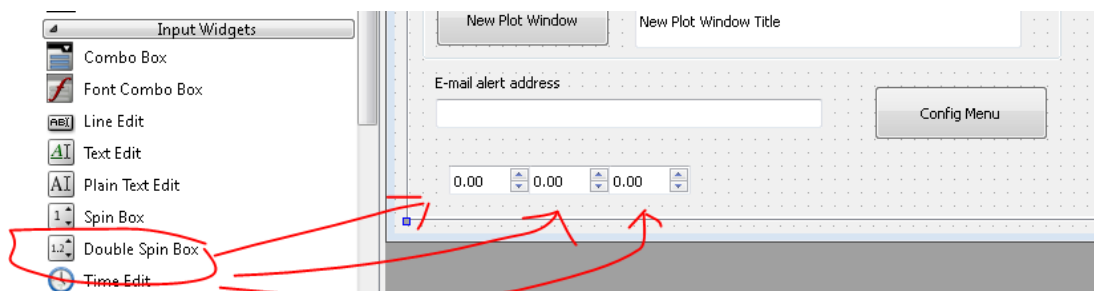
- 1) Go to \PyGMI_files\ and open Graphical_User_Interface.ui with Qt Designer (QtDesigner should be in the PySide folder) if you associated QtDesigner with “.ui” files, then you just have to double-click on the file. An interface with a layout similar to this should appear.



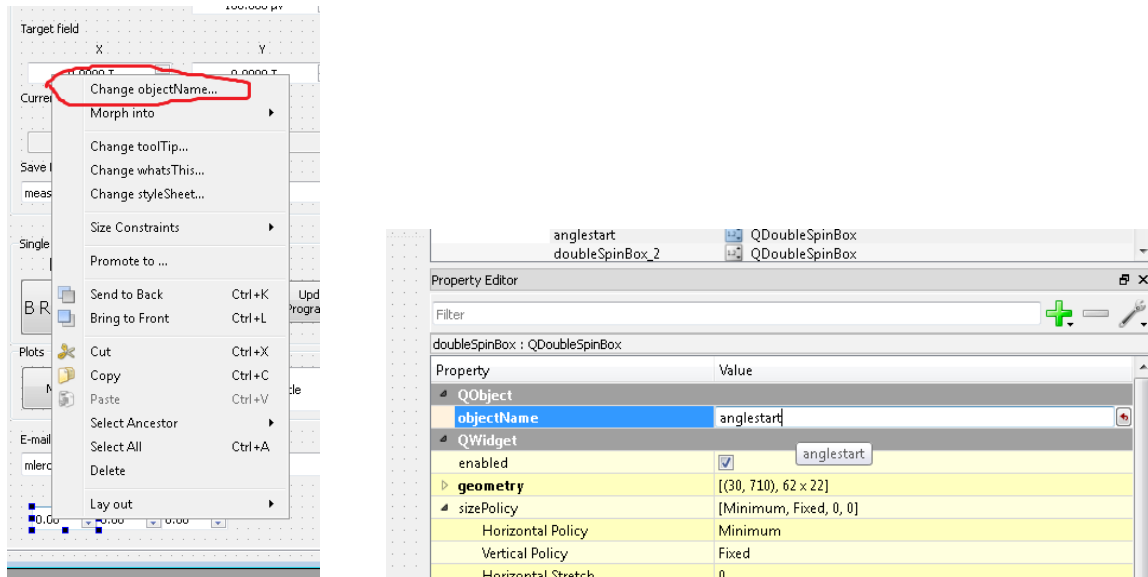
- 2) In the center is the graphical interface as it should look like when you start the program. Now we will add the new elements that we want.
- 3) Make sure you are in the Edit Widgets mode



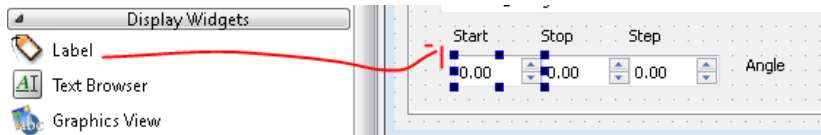
- 4) Let's add three Double Spin Box to enter a float value, simply by dragging and dropping each of them from the list on the left to the interface in the center



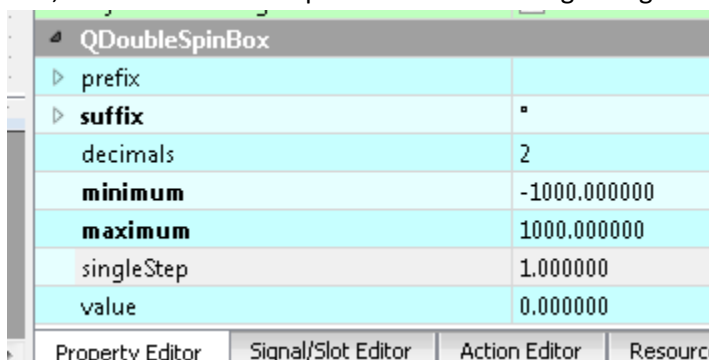
- 5) Now we have to choose a name for each of these boxes, as we will need these names to access the boxes from within the program. For this, right click on a box and select “change objectName”, or in the property editor look for the item “objectName” at the top of the list, and enter the new name for the object (choose something that respects Python variable naming rules: [a-Z][0-9] and the underscore “_”)



- 6) Do this for the three boxes: we will name them anglestart, anglestop and anglestep
7) Now let's add a label above them : each label can be edited by double clicking on it



- 8) Then we need to edit the allowed boundaries for these entries. For each box, we have to select it, go to the property editor and scroll down to “minimum”, “maximum” etc... Here we will allow values from -1000 to 1000, single step of 1 (when pressing the up/down buttons), default value of 0, and two decimals input. Also we add a degree sign as a suffix that will appear in the box.



- 9) Once this is done, we can save the modified interface and exit QtDesigner

- 10) Now, we will make these entries available from within the measurement script. For this, open the file Frontpanel_values.py (also in \PyGMI_files\)
- 11) In the section corresponding to the Measurements tab (actually, anywhere would be fine, but it helps to keep things in order) , we add the following three lines (notice the other entries to see examples for other types such as textboxes)

```
#####  
#"Measurements" tab#  
#####  
#Text entries  
self.email_address=ui.email_address.text()  
self.savefile_txt_input=ui.savefile_txt_input.text()  
  
#Numeric entries  
self.measure_delay=ui.measure_delay.value()/1000.0  
self.measure_speed=ui.measure_speed.value()  
self.repeat_points=ui.repeat_points.value()  
  
self.current1=ui.I_source_setpoint.value()*1e-6  
self.current2=ui.I_source_setpoint_2.value()*1e-6  
self.current3=ui.I_source_setpoint_3.value()*1e-6  
  
self.voltage1=ui.V_setpoint_1.value()  
self.voltage2=ui.V_setpoint_2.value()  
self.voltage3=ui.V_setpoint_3.value()  
  
self.IV_voltage_criterion=ui.IV_voltage_criterion.value()*1e-6  
  
self.B_X_setpoint=ui.B_X_setpoint.value()  
self.B_Y_setpoint=ui.B_Y_setpoint.value()  
self.B_Z_setpoint=ui.B_Z_setpoint.value()  
  
self.anglestart=ui.anglestart.value()  
self.anglestop=ui.anglestop.value()  
self.anglestep=ui.anglestep.value()  
  
#CHECKBOXES  
self.voltage_criterion_on=ui.voltage_criterion_on.isChecked()
```

- 12) As explained in the comments of that file: suppose we just added a text entry in the interface (the .ui file) using Qt Designer and we named that entry "my_text_entry" in Qt Designer. This entry can now be accessed from here using the name "ui.my_text_entry". Next, we want to retrieve the text in that entry and put it in the variable "my_text" which will be available from within the measurement script, for that we just add the line "self.my_text=ui.my_text_entry.text()". After this, self.my_text will be accessible as "frontpanel.my_text" or "f.my_text" in the measurement scripts.
- 13) Restart PyGMI: it will detect a change and automatically recompile the interface
- 14) The value in our new elements is now accessible in the measurements scripts by writing "self.frontpanel.My_new_element_value" or "f. My_new_element_value"

Creating a New Macro Command

In this section we will see how to create a new command that can be recognized in the Macro editor. In particular, following up on the previous section where we added three new entries to the graphical interface, we will create a new command that modifies the values inside these boxes.

Open the file Macro_editor.py located in the folder /PyGMI_files/ with your favorite editor

Scroll down to the class New_command_template()

```
class New_command_template():
    def __init__(self):
        #Text that will appear in the list of commands on the right side of the Macro editor
        self.label=""
        #Regular expression which may or may not catch parameters
        self.regex_str=""
        #Add this to the beginning and end of the regular expression
        #so that whitespaces before and after will not prevent the regex from matching
        self.regex_str="^ "+self.regex_str+" $"
        #instantiate regex
        self.regex=QRegExp(self.regex_str)

    def run(self,main):
        #what to do when the regex defined just above in __init__
        #has matched the current line of the Macro
        #get the captured parameters
        #values=self.regex.capturedTexts()
        #send commands to instruments
        ##>change the temperature setpoint
        #with main.reserved_access_to_instr:
        #    main.temp_controller.switch_ramp(loop,'off')
        #    main.temp_controller.set_setpoint(loop,setpoint)
        #modify stuff in the interface
        ##>set the voltage
        #if values[1] in ['1','2','3']:
        #    V_source_setpoint=eval("main.ui.V_setpoint "+values[1])
        #    V_source_setpoint.setValue(float(values[2]))
        #Finally go to next line of macro...
        self.next_move=1
        #...after 10 milliseconds
        self.wait_time=10
```

Then make a copy of it, and rename it AngleCommand

```

class AngleCommand():
    def __init__(self):
        #Text that will appear in the list of commands on the right side of the Macro editor
        self.label=""
        #Regular expression which may or may not catch parameters
        self.regexp_str=""
        #Add this to the beginning and end of the regular expression
        #so that whitespaces before and after will not prevent the regex from matching
        self.regexp_str="^ "+self.regexp_str+" $"
        #instantiate regex
        self.regexp=QRegExp(self.regexp_str)

    def run(self,main):
        #what to do when the regex defined just above in __init__
        #has matched the current line of the Macro
        #get the captured parameters
        #values=self.regexp.capturedTexts()
        #send commands to instruments
        ##>change the temperature setpoint
        #with main.reserved_access_to_instr:
        #    main.temp_controller.switch_ramp(loop,'off')
        #    main.temp_controller.set_setpoint(loop,setpoint)
        #modify stuff in the interface
        ##>set the voltage
        #if values[1] in ['1','2','3']:
        #    V_source_setpoint=eval("main.ui.V_setpoint_"+values[1])
        #    V_source_setpoint.setValue(float(values[2]))
        #Finally go to next line of macro...
        self.next_move=1
        #...after 10 milliseconds
        self.wait_time=10

```

Now we define what will be the text of the command. We want it to be :

Set **start/stop/step** angle to **VALUE**

The corresponding regular expression, with capturing parameters, for this behaviour is:

“Set (start|stop|step) angle to "+Regexfloat

(For the sake of convenience, Regexfloat is a regex defined at the top of the file and that matches most way of expressing a float value. For more details, refer to regular expression tutorials, such as the online manual for the Python module “re”)

So we add this regular expression to the beginning of the class

```

class AngleCommand():
    def __init__(self):
        #Text that will appear in the list of commands on the right side of the Macro editor
        self.label="Set start|stop|step angle to FLOAT"
        #Regular expression which may or may not catch parameters
        self.regexp_str="Set (start|stop|step) angle to "+Regexfloat

```

Finally we must tell the program what to do when it matches such an expression in the Macro Editor, this is done by editing the run method of the class as follow:

```

def run(self,main):
    #what to do when the regex defined just above in __init__
    #has matched the current line of the Macro
    #get the captured parameters
    values=self.regex.capturedTexts()
    #set the corresponding angle box
    if values[1] in ['stop','step','start']:
        anglebox=eval("main.ui.angle"+values[1])
        anglebox.setValue(float(values[2]))
    #Finally go to next line of macro...
    self.next_move=1
    #...after 10 milliseconds
    self.wait_time=10

```

In details:

Firstly, we get the captured values (values that are inside parentheses in a regex string)

```

#get the captured parameters
values=self.regex.capturedTexts()
" . . . . .

```

We double check that the captured value is start, stop or step (cannot hurt since we are doing an “eval” just after) then we get the object corresponding to that box in the interface: this object is named `main.ui.NAME_DEFINED_IN_QTDESIGNER` (as all others objects of the main interface)

```

#set the corresponding angle box
if values[1] in ['stop','step','start']:
    anglebox=eval("main.ui.angle"+values[1])

```

Then we set the value of that box with the value provided by the user (the methods available for a double Spin Box object are all referenced in the online documentation of PySide. Look up PySide `QDoubleSpinBox` on the Internet for more details) note that the value entered by the user comes as a string, so it must be converted to a float

```

..    anglebox.setValue(float(values[2]))

```

NB: Of course, we could also have written the code below, but I prefer the compact, reusable code above

```

if values[1]=='start':
    main.ui.anglestart.setValue(float(values[2]))
elif values[1]=='stop':
    main.ui.anglestop.setValue(float(values[2]))
elif values[1]=='step':
    main.ui.anglestep.setValue(float(values[2]))

```

The final step is to add this class to the list of valid commands. Scroll up to the `Macro_editor` class and add the name of our new command to the list of valid commands.

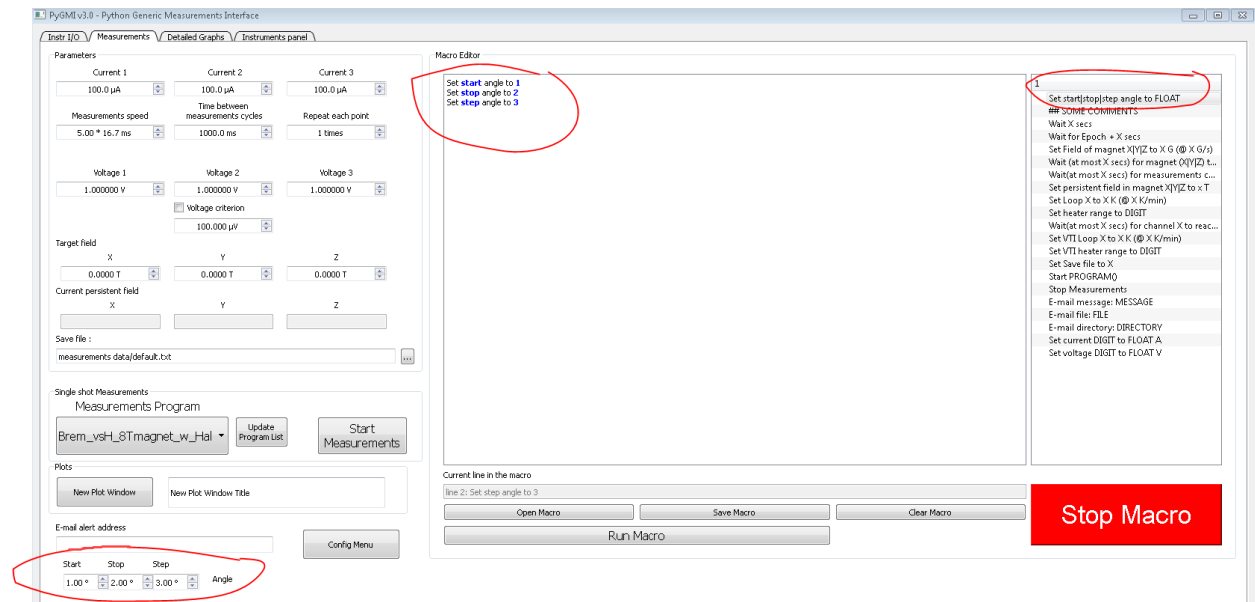
```

class Macro_editor(QWidget):
    def __init__(self, parent=None, title='Macro editor'):
        # This class derives from a Qt Widget so we have to call
        # the class builder ".__init__()"
        QWidget.__init__(self)
        # "self" is now a Qt Widget, then we instantiate the user interface
        # generated with QtDesigner and call it self.ui
        self.ui = Ui_Macro_editor()
        # Now we have to feed the GUI building method of this object (self.ui)
        # with a reference to the Qt Widget where it will appear, i.e. itself
        # but the elements of the GUI will actually be children of the GUI (i.e. of self.ui not of self)
        self.ui.setupUi(self)
        self.setWindowTitle(title)

        # List the Macro command that will be loaded and available
        # into the Macro editor.
        # Once you have created a new Macro command class in this file
        # you must add it in this list for it to be available in
        # the Macro editor.
        # You may also remove commands from this list if you want to deactivate them
        self.valid_list_of_commands=[
            AngleCommand(),
            CommentsCommand(),
            WaitCommand(),
            WaitForEpoch(),
            SetFieldCommand(),
            WaitForHStableCommand(),
            WaitForMeasureCommand(),
            SetPersistentFieldCommand()

```

Restart PyGMI and, that's it, the new command is available in the editor !



Creating a new program of measurements

- 1) Make a copy of the file "Measurements_program_template.py" and place it in the folder

"\PyGMI_files\Measurements_programs"

- 2) Rename your copy of the file with the name that you want while leaving the ".py" extension, but **WARNING** a few rules have to be respected: the name of the file should be a valid Python variable name. It means that the name you choose can contain only the following characters: a-z/A-Z/1-9 and the underscore sign "_". It also must not contain any whitespace " " and must not start with a digit 1-9.

Example of acceptable names:

- a.py
- M_vs_H_with_2_voltmeters.py
- _1.py

Example of invalid names:

- M vs H with 2 voltmeters.py *(whitespace in the middle of the name)*
- 2_voltmeters.py *(starts with a digit)*
- +l_-l_measurements.py *(invalid characters "+" and "-")*

- 3) Once the file is in the folder, modify the template to your needs. Don't forget to use the reserved_bus_access lock to access the instruments (See the section on Technical details on multi-threading/parallel-processing, paragraph Lock)
- 4) When you are done, save the file and press the Update Program List button in the PyGMI interface (restarting PyGMI will also do it)
- 5) Your program should now be accessible for use in PyGMI in the "Measurements" tab: the name of your program should appear in the drop-down menu "Measurements Programs".

Modifying a program of measurements

The script is divided in three parts: a technical part to ensure the connection with the main process of the program, a second part containing some names shortcuts to make the code clearer, and finally the part that communicates with the instruments. Normally, it's only necessary to modify that last part.

Here is the first part; it starts with importing the necessary libraries

```
#Multithreading
import threading
#Time measurement
import time
```

Then we retrieve all the arguments provided by the main process and store them locally

- Mainapp is a reference to the instance of the main process (it's necessary because the objects connecting to the instruments are created in that instance)
- Frontpanel contains a "snapshot", at the time of start-up of the measurement process, of all the values in the entry fields of the frontpanel
- Data_queue is a Queue to transfer data back to the main process
- stop_flag is a Flag that the main process uses to tell the measurement process to stop
- Instr_bus_lock is a Lock that must be used to reserve the access to the instruments, to avoid connection conflicts with other processes

```
#####create a separate thread to run the measurements without freezing the front panel#####
class Script(threading.Thread):
    def __init__(self,mainapp,frontpanel,data_queue,stop_flag,GPIB_bus_lock,**kwargs):
        #nothing to modify here
        threading.Thread.__init__(self,**kwargs)
        self.mainapp=mainapp
        self.frontpanel=frontpanel
        self.data_queue=data_queue
        self.stop_flag=stop_flag
        self.GPIB_bus_lock=GPIB_bus_lock
```

In the second part, a few shortcuts are defined to make the code more compact: for example "f." is short for "self.frontpanel", which allows to access front panel values faster : e.g. "f.instr_on_1" instead of "self.frontpanel.instr_on_1" . And since instrument 1 is supposed to be a lockin for that script, we define lockin as equivalent to "m.instr_1"

```
def run(self):
    #this is the part that will be run in a separate thread
    #####
    #SHORTCUTS
    m=self.mainapp          #a shortcut to the main app, especially the instruments
    f=self.frontpanel       #a shortcut to frontpanel values
    reserved_bus_access=self.GPIB_bus_lock    #a lock that reserves the access to the GPIB bus

    lockin=m.instr_1
    -----
```

Here is the third part that needs to be modified

First, we create a list of character strings (the header), which the main process will use as the title of the columns in the savefile. This way, each time the measurement script sends some data again, the main process will assume that this data is a list of length equal to the header, in the same order, and it will save each element of the list of data in the corresponding column of the savefile.

```
#####
#SAVEFILE HEADER - add column names to this list in the same order
#as you will send the results of the measurements to the main thread
#for example if header = ["Time (s)","I (A)","V (volt)"]
#then you have to send the results of the measurements this way :
#"self.data_queue.put([some time, some current, some voltage],False)"
header=['Time (s)']
header+=['Time (min)']
if f.temp_controller_on:header+=["T (K)"]
if f.instr_on_1:header+=["Radius (V)","theta"]

#####|
#ORIGIN OF TIME FOR THE EXPERIMENT
start_time=time.clock()
#####
#SEND THE HEADER OF THE SAVEFILE BACK TO THE MAIN THREAD, WHICH WILL TAKE CARE OF THE REST
self.data_queue.put((header,True))
```

Then comes the main loop where the measurements occur. Here we measure the temperature and the signal of a lockin twice, then calculate the average. Note how each instruction sent to the instruments is wrapped in a “with reserved_bus_access:” block, so as to avoid connection conflict when discussing with the instruments.

```
if f.instr_on_1:lockin.set_amplitude(f.voltage1)

#####Control parameters loop(s)#####
while True:
    #Check if the main thread has raised the "Stop Flag"
    if self.stop_flag.isSet():
        break
    #reserve the access to the instruments, then discuss with them
    with reserved_bus_access:
        #Measure T
        if f.temp_controller_on:T=m.temp_controller.query_temp('B')
        #Measure R and theta
        if f.instr_on_1:freq,R1,theta1=lockin.query_f_R_theta()
        #Measure R and theta
        if f.instr_on_1:freq,R2,theta2=lockin.query_f_R_theta()
        R=(R1+R2)/2.0
        theta=(theta1+theta2)/2.0
```

Finally, after one series of measurements, we compile all results and put them in a list in the same order as the header, then we send that list of data back to the main process which will automatically take care of the storage and plotting operations.

NB: data_queue expect a tuple (data,flag). If flag is “True” it assumes data is a header for a datafile, if flag is “False” it assumes data is some actual datapoints. And, if flag is the string “newfile”, it assumes that data is the name of the new savefile that it will try to create.

```
#####Compile the latest data#####
t=time.clock()-start_time
last_data=[t,t/60.0]
if f.temp_controller_on:last_data.append(T)
if f.instr_on_1:last_data.extend([R,theta])

#####Send the latest data to the main thread for automatic display and storage into the savefile#####
self.data_queue.put((last_data,False))

#Check if the main thread has raised the "Stop Flag"
if self.stop_flag.isSet():
    break
#####Wait mesure_delay secs before taking next measurements
time.sleep(f.mesure_delay)
```

Technical details on multi-threading/parallel-processing

A. Queues

The great thing of having the measurements running in a separate process, is that it is not blocked by the interactions of the user with the graphical interface, nor does it block the graphical interface. However, since both threads share some variables, some precautions have to be taken to avoid any conflict (meaning: one thread trying to read a variable, meanwhile the other modifies it).

As a good practice in a GUI, only the main thread is supposed to handle the GUI events (user clicking here, typing that, or plotting this). So the measurements thread should not interact with the user interface, i.e : you cannot ask it to plot the data directly. That is why it is necessary to send back the data to the main thread before plotting it.

To handle this data communication between thread I use a standard “Queue” variable. This is a so-called “thread-proof” variable that cannot be accessed by both threads at the same time. If both tries at the same time, then the second thread will wait until the first one is finished (optionally you can tell the second thread to wait until some timeout, after which it will just go on if the first thread is still accessing the Queue).

The way a Queue works in Python is very simple: from one thread you just “.put(some data)” to it, and from the other thread you just “.get()” the data from the Queue. If some data is appended several times in a row to the same Queue, all the data will be buffered, no data will be lost. And the nice thing is that all the buffering and thread lock operations are automatically handled by Python.

NB: the data can be retrieved from the Queue either in the order in which they arrived, a.k.a. First-In-First-Out or FIFO, or the youngest entry can be retrieved first a.k.a Last-In-First-Out or LIFO. Obviously, I use a FIFO Queue, to plot the data in order.

B. Lock

In some very special cases, both the main thread and the measurement thread may try to discuss with an instrument at the same time, which will raise a conflict. To avoid that, a so-called “Lock” is used by each thread to reserve the access to the instruments. A Lock is another type of thread-proof variable that can be accessed by several threads at the same time. As in a Queue, if the second thread finds that the Lock is being used by the first thread, it will wait until the Lock is released.

The way a Lock works in Python is quite simple: suppose “GPIB_bus_lock” is a Lock object, just add the statement “with GPIB_bus_lock:” at the beginning of the part of your code that may access some shared variable (like the instruments) and that’s it: Python will automatically wait for the Lock to be released and reserve the Lock before executing the commands in this block.

Example:

```
with GPIB_bus_lock:  
    T=temp_controller.query_temp(frontpanel.temp_controller_channel) #Measure Temperature
```


(NB: this Lock is used in the main thread anytime it tries to access an instrument)

C. Flag

A Flag is another type of thread-proof variable. A flag can either be set or not, so it's equivalent to either True or False (a boolean variable). The main program ".set()" a flag to tell the measurements program to stop. The measurement program is aware of it by checking periodically whether the Flag ".isSet()" or not. Again, Python make the threads wait automatically to avoid any access conflict.