Handbook for the PillaLab

Continuously updated Version

12.04.2021

Author: Philipp Kagerer

Inhalt

[General remarks on the software 1](#_Toc76859835)

[Program Design 1](#_Toc76859836)

[Functionalities, capabilities 2](#_Toc76859837)

[Programmer and Admin Guide 3](#_Toc76859838)

[GUI creation: 3](#_Toc76859839)

[Program structure 3](#_Toc76859840)

[Styleguide for Developers 3](#_Toc76859841)

[Design Choices 3](#_Toc76859842)

[Importing and executing foreign code/recipes 4](#_Toc76859843)

[Users Guide 4](#_Toc76859844)

[Using the settings menu 4](#_Toc76859845)

[Recipe programming 5](#_Toc76859846)

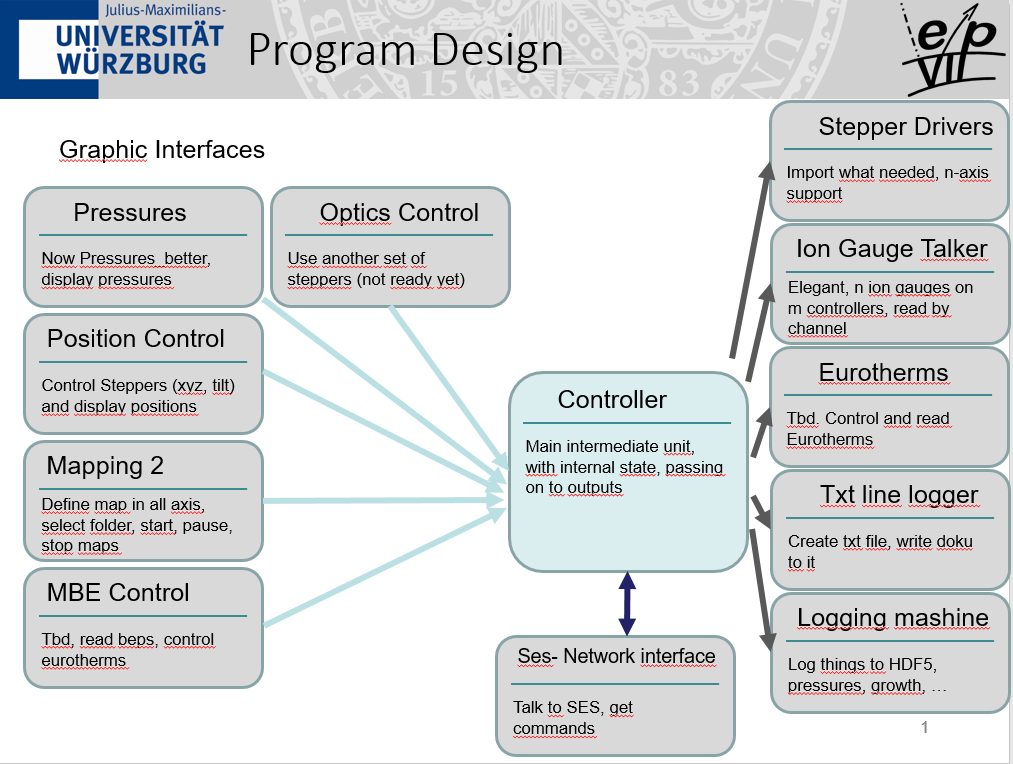
# General remarks on the software

## Program Design

PillaLab is a comprehensive GUI based Lab software meant to assist ARPES and MBE Experiments.

It can control various Stepper-motor types, Eurotherm controllers, read pressure gauges, …

The general design is written such, that there is one main class, the Controller, which can be accessed by GUI’s and which calls all necessary drivers. Everything is bridged through the Controller, such that multiple GUI’s can be used simultaneously without overkilling some controller.



Applications at the time of writing:

Stepper control at the R4000-mashine, EP7, Würzburg

Complete MBE Control, EP7, Würzburg

XAS acquisition (P04 Hamburg)

Published as a private Repo on Github

## Functionalities, capabilities

As grown generically for multiple purposes, Pillalab has a bunch of functionalities, which are combined in one package more as a convenience of having only one universal tool, than actually usefully combining everything. The Toolset can be used as a ARPES manipulator control and hence has various functions combining it with Scienta Omicron SES software.

For mapping /remote control two functionalities are available. Either the Toolset is the boss, mapping is defined in the Pillalab and executed there. Pillalab then pulls SES to front, sends the shortcut for a scan and waits for a file to appear in a predefined foler. Then a step is done and the whole thing is repeated. Make sure, that SES does only do one interation.

2nd way would be network mapping, where a special DLL extension in SES to the manipulator DLL is used. Here the Pillalab is only the receiver and communicates with SES. Position commands are accepted and position queries are answered.

Growth control includes controlling PID controllers, Stepper motors, and reading pressures. In a smart setting this is enough to fully control the growth process.

GUI also include a module to log and display various chamber pressures with live update.

Logging is mostly done to hdf5-files with an easy to understand structure, once opened with a hdf5 viewer of your choice.

# Programmer and Admin Guide

## GUI creation:

GUI modules are created using Winpage 5.04 by a simple drag and drop creation mechanism. Winpage creates two files for a windows, the second one called \_Support is meant for the actual programming. Here this is used to pass the commands on to the controller unit to enable seamless multithreading.

As a design choice the support files are modified to always include the “main” window as topwindow. This leads to only one sign appearing in the taskbar. This is done using the line

top, w = Cell\_control3.create\_Toplevel1(theController.root)

in the startMainGUI call.

## Program structure

Everything in this suite is written as a class, mostly because I found it easier to handle variables in multi-threaded processes. All variables can thus be stored and written from theControler.NAME/FUNCTION.

The only exception here are the GUI modules, which run using global variables.

## Styleguide for Developers

* Use clear commenting, comments are incremented to be at same height for entire document
* Error catching is done using statements like

*Try:*

*…*

*Except Exception as e:*

*Print(“point of error “ + str(e))*

* Everything is strictly object oriented
* Commands should be written as “python” as possible, meaning as compressed as possible to enable good reading

## Design Choices

How steppers are implemented: There are in general two ways of implementing steppers. The standard way is implementing one controller by a driver and controlling it via the stepper control pad. This is the original implementation and is used for ARPES mapping mostly. Given more steppers and more controllers things would become more complicated. For this the motion module is written, which can implement n steppers on m controllers. It basically works as a wrapper around the controllers and motors can then be controlled by calling the motion module class, which passes on movement commands as needed.

How Ion gauges are read: Here a similar approach holds. A wrapper class implements as many controllers as necessary with individual driver files. Reading commands are bridged on. From the controller class this ion gauge talker can be used as a single interface to n ion gauges. For reading new types of gauges the ion gauge talker as well as the driver files will have to be adapted.

## Importing and executing foreign code/recipes

Special function used e.g. for growth programming, it is possible to include python files and run them in separate threads. This is done in three steps: loading the path, loading the code and running.

import importlib.util

spec = importlib.util.spec\_from\_file\_location("automatizer\_programm",apath)

automatizer = importlib.util.module\_from\_spec(spec)

spec.loader.exec\_module(automatizer)

self.AutomatizedModule = automatizer.Programm(self)

This is done using the spec\_from\_file\_location function. In the run here it creates a runnable object and then runs it using exec\_module. Here it is important, that the program to be run is a class, which does not need anything specific upon initialisation and does not do much also. The fun should only begin once the class.start() function is called.

            AutomatizedModule = self.ImportObjectDict[referer]

            mythread = threading.Thread(target=AutomatizedModule.run)

            self.runningThreadDict[referer] = mythread

            mythread.start()

The run button then creates a thread in which the start function of the class is running. This is a elegant way to avoid program crashes/freezes by ill written code.

# Users Guide

## Using the settings menu

Settings are by default loaded from a initialisation file (Settingslib.py) as a single large python dictionary. Dictionaries can be filled by giving the key and entering a value.

Empty dictionary: mydictionary = {}

Filling it: mydictionary[“my-key1”] = “some text”

Reading it: myreading = mydictionary[“my-key1”]

The direct editing of the Settingslib is meant to be used during programming and initialisation, where some extra entries may have to be added, etc.

Pillalab also has the possibility to change settings by clicking on menu-settings. All dictionary entries are then displayed and can be changed and saved. Saving is done to a pickle-file, meaning a compressed python file called settings.pkl. If this file is existent in the main path of Pillalab, loading is by preferably done from this and the Settingslib.py file is ignored. Delete this file, if you want to fall back to the Settingslib.py.

Best practice for key-naming:

The settings-keys are always structured as “main-topic”.”settingsname”, e.g. growthcontrol.controllernames. Most settings are structured as lists [a,b,c,d] or nested lists [[a,b,c],[a,b,c]]. Mostly this refers to multiple devices for each of which a number of settings has to be defined. E.g. multiple controlers, for which names, ports and some other things are defined as .names = [“name1”,”name2”,…] , .ports = [1,2,…], …

## Recipe programming

Pillalab is able to import external python code, compile it and execute it in a Thread on runtime. A Thread is basically a loop running in parallel to the main program. The import is done via a GUI window. Programs need a few main features, the minimal program would look like:

import time

class Programm:

    def \_\_init\_\_(self,theController):

        self.theController = theController

        self.running = False

        self.paused = False

    def run(self):

        if not self.running:

            self.running = True

            self.paused = False

            self.mainProgram()

        else:

            self.paused = False

    def pause(self):

        self.paused = True

    def stop(self):

        self.running = False

    def mainProgram(self):

        while self.running:

            if not self.paused:

                motlist = [0,1,2,3,4,5]

                for mot in motlist:

                    self.theController.SetShutterState(mot,"open")

                time.sleep(3)

                for mot in motlist:

                    self.theController.SetShutterState(mot,"closed")

                time.sleep(3)

The program is a class always called “Programm”. The functions \_\_init\_\_, run, pause and stop are mandatory, everything else is up to you. It is good practice to use the run command to call a main program, all flags like self.running/paused/… should be used as shown to avoid double starting of a routine. This could easily happen, by a simple double-button click elsewise. When importing Pillalab first searches for the path of the file by clicking on a path button. The load button will import the class and call the init function. This function does only (!) take theController as an argument, but this is definitely enough, as the controller is the entire running instance. By calling theController.XXX any function or attribute can be called, e.g. asking for pressures, opening shutters, …

As this is a direct import no boundary is set to your power, you could easily delete the entire PC, burn a few cells, or set the building on fire, if you wanted, so be careful.

The main Program (call it whatever you want, just make sure it is called from the run function) and any additional function can be edited as you like.

The functions pause and stop are called from the GUI and can and should (!) be useable. They will only activate or deactivate the flags self.paused/running. If self .paused is True the program should hang, if self.running is set to false it should terminate as soon as possible.

A good practice is using statements like

while self.running: # for loops

if not self.paused:

# do stuff

else:

time.sleep(1)

This will run as long as nobody pushes abort and only do things if not paused. The else time.sleep part will ensure, that there is no system-overload by the python script running as fast as ist can doing nothing.

The rest is…python…and can be learned by doing.

This would be a more comprehensive program to degas cells checking the pressure on the way, ensuring it does not go too high (skipping the run/pause/stop part, as it is always the same).

    def mainProgram(self):

        evaps = ["Vodka","Rum","Tequilla","Whiskey"]

        tlimits = [600,200,550,400]

        tstarts = np.zeros(len(evaps))

        for i in range(len(evaps)):

            temp = self.theController.ReadTemperature(evaps[i])

            tstarts[i] = temp

        ramptime = 40 \*60 # seconds

        rampstep = 5 # seconds

        pressurelimit = float(0.0000005)

        print("Pressurelimit: " + str(pressurelimit))

        j = 0

        print("Starting!!!")

        while self.running:

            currentpressure = float(self.theController.GetSinglePressure("MBE")[0])

            print("Pressure in loop: " +str(currentpressure))

            if (not self.paused) and (not (currentpressure > pressurelimit)):

                print("stepping")

                for i in range(len(evaps)):

                    evap = evaps[i]

                    temp = tstarts[i] + ((tlimits[i]- tstarts[i])/(ramptime/rampstep))\*j

                    if not temp > tlimits[i]:

                        self.theController.SetTemperature(evap,temp)

            j += 1

            print("rampstep completed")

            time.sleep(rampstep)

A complete growth recipe would look like this:

def mainProgram(self):

        cycles = 5

        TeShutter = 1

        BiTeShutter = 2

        MnTeShutter = 3

        T\_substCtrl = "Beer"

        T\_clean = 350

        T\_growth = 245

        T\_anneal = 350

        time\_clean = 60\*10

        time\_BiteGrowth = 50

        time\_MnTeGrowth = 50

        time\_anneal = 50

        stableTime = 60

        growthTPrecission = 0.2

        if self.running:                                                    # ramp up substrate to clean it, wait, then go on

            self.theController.SetTemperature(T\_substCtrl,T\_clean)

            success = self.WaitForStability(T\_substCtrl,T\_clean,time=stableTime,Precission=1)

            print("Stable at clean: " + str(success))

            time.sleep(time\_clean)

        currentCycle = 0

        while self.running:                                                 # growth loop

            if not self.paused:

                #\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*grow BiTe layer\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

                self.theController.SetTemperature(T\_substCtrl,T\_growth)     # ramp to growth temperature and wait till stable

                success = self.WaitForStability(T\_substCtrl,T\_growth,time=stableTime,Precission=growthTPrecission)

                print("Stable at growth: " + str(success))

                self.theController.SetShutterState(BiTeShutter,"open")      # open shutters and wait the growth time

                self.theController.SetShutterState(TeShutter,"open")

                time.sleep(time\_BiteGrowth)

                self.theController.SetShutterState(BiTeShutter,"closed")      # close shutters and wait the growth time

                self.theController.SetShutterState(TeShutter,"closed")

                #\*\*\*\*\*\*\*\*\*\*\*\*\*\*growMnTe layer\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

                self.theController.SetShutterState(MnTeShutter,"open")

                time.sleep(time\_MnTeGrowth)

                self.theController.SetShutterState(MnTeShutter,"closed")

                #\*\*\*\*\*\*\*\*\*\*\*\*\*\*anneal\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

                self.theController.SetTemperature(T\_substCtrl,T\_anneal)     # ramp to growth temperature and wait till stable

                success = self.WaitForStability(T\_substCtrl,T\_anneal,time=stableTime,Precission=growthTPrecission)

                print("Stable at anneal: " + str(success))

                time.sleep(time\_anneal)

                if currentCycle < (cycles-1):

                    currentCycle +=1

                else:

                    print("finished")

                    break

    def WaitForStability(self,ctrl,aim,time = 60,Precission = 0.2,updatetime = 1):

        counter = 0

        while self.running:

            if not self.paused:

                temp = self.theController.ReadTemperature(ctrl)

                if abs(temp-aim) < Precission:

                    counter += 1

                    if counter >= round(time/updatetime):

                        return True

                else:

                    counter = 0

                time.sleep(updatetime)

A few comments to the growth functions: The individual controllers are addressed with their names, e.g. Beer, Vodka, …. The names can be found in the settings files under the point “growthcontrol.Controlernicknames”. The individual shutters are addressed using their number, e.g. 1,2,3,… The so called “shutternames” s1, s2, … are not important.