Automation of an optical setup for investigation of quantum dots

BACHELORARBEIT

zur Erlangung des akademischen Grades
Bachelor of Science
(B. Sc.)
im Fach Physik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät I
Institut für Physik
Humboldt-Universität zu Berlin

von
Herrn Chris Müller
geboren am 09.07.1989 in Königs Wusterhausen

Betreuung:

1. *Prof. Dr. Oliver Benson*
2. *Prof. Achim Peters, Ph.D.*

eingereicht am: *26. September 2013*

Abstract

The aim of this work is to automate an optical setup for the investigation of quantum dots. Step by step it is explained how to write such a automation in the Python programming language. It is shown how devices can be controlled via the serial port or dynamic link library (DLL) files. To provide a intuitive and simple user interface, it is illustrated how to create graphical user interface with the **traits** package. For that the implementation of buttons, text fields, plots, different menus and event handles are explained in detail. Writing measurement programs occupies lab time and is bound to the physical devices. To overcome these problems, a simulation environmnet for the serial port is created. Finally all different instrument controls are combined in a single quantum dot search program.

Contents

1 Introduction **1**
1.1 General Introduction . . . 1
1.2 Focus of this Work . . . 3

2 Python Classes and Inheritance **6**
2.1 Classes . . . 6
2.2 Inheritance . . . 7

3 traits Introduction **9**
3.1 First Steps with traits . . . 9
3.2 User Interaction . . . 12
3.3 Plotting with Chaco . . . 15
3.4 Handling events in Plots . . . 18
3.5 traits Advanced . . . 18
3.5.1 Drop-Down Menu . . . 18
3.5.2 Menubar . . . 20
3.5.3 Closing handle . . . 21
3.5.4 Import a GUI . . . 23

4 pySerial **24**
4.1 Introduction to pySerial . . . 24
4.2 Buffer and how to handle pySerial . . . 25

5 SimSerial **27**
5.1 How to write a Simulation . . . 27
5.2 How to use SimSerial as a Module . . . 31
5.3 Example Usage of SimSerial . . . 32

6 Controlling the Camera **35**
6.1 ctypes Basics . . . 35
6.2 Using a DLL File . . . 36

7 QDSearch **38**
7.1 Searching the Maximum Intensity . . . 39
7.2 Searching Quantum Dots . . . 40

8 Outlook **42**

9 Summary **43**

References **44**

1 Introduction

1.1 General Introduction

The history of cryptography is several thousand years old. However, with the appearance of the internet and electronic communication, cryptography became very important in daily life. By combining cryptography and quantum mechanics a new method, called the quantum cryptology, was created [1].
Assume the sender, Alice, is sending a message to the receiver, Bob. If the message is not encrypted, the eavesdropper, Eve, would be able to read Alice's message. To prevent this Alice and Bob can use keys to encrypt the message. If Eve does not know the key, she can read the encrypted message, but can not decrypt it. The problem is that the key has to be exchanged between Alice and Bob. If Eve could eavesdrop this exchange, the cipher becomes useless.
Bennett and Brassard showed in their BB84 protocol a possibility of a safe key distribution by using quantum mechanics [2]. Alice sends Bob a number of photons which are polarized randomly and independently. Bob has different bases to measure the polarization. A photon state is disturbed if the wrong basis was chosen during the measurement. Bob determines Alice's photons while he is changing his basis randomly. Afterwards Alice and Bob compare the bases, they have chosen, through a classical communication channel. Out of the photons with the same basis the new key can be generated by using the result of the measurement. If Eve could eavesdrop this key generation, the method would be as unsafe as the classical one. However, the no-cloning theorem of quantum mechanics forbids to copy an unknown quantum state. So Eve disturbs the photon by measuring it. For this reason Alice and Bob compare some results of the measurement with the same basis and calculate the experimental error. If the error is small enough, they know that no one was eavesdropping. Otherwise Alice and Bob repeat the key generation. Even if no one tries to eavesdrop the communication, the error will not be zero because of the noise.
Optical communication is limited by fiber losses [3]. For example, this means for the distance Munich-Berlin (about 500 km) that with a communication speed of 1 Gb/s only 628 bytes would reach the receiver after one day. In classical communication that problem is solved with repeaters, which renew the signal. As a result of the no-cloning theorem the photon state can not be renewed, because it implies complete knowledge about the state.
This seems to render the concept of a quantum repeater impossible. However, it is possible to realize a quantum repeater with entangled photons [4]. In this process several entangled photon pairs are used instead of a single photon. Every pair travels only a part of the distance. The principle of a quantum repeater is depicted in Figure 1. Using a quantum repeater Alice and Bob will share one entangled photon pair. Alice can measure her photon and will know the result of Bobs measurement under the condition that they use the same basis for the measurement.
For the realization of quantum repeaters entangled photon pair sources are needed. Possible candidates are nonlinear-crystals [5] or entangled photons from quantum dots (QDs) [6].

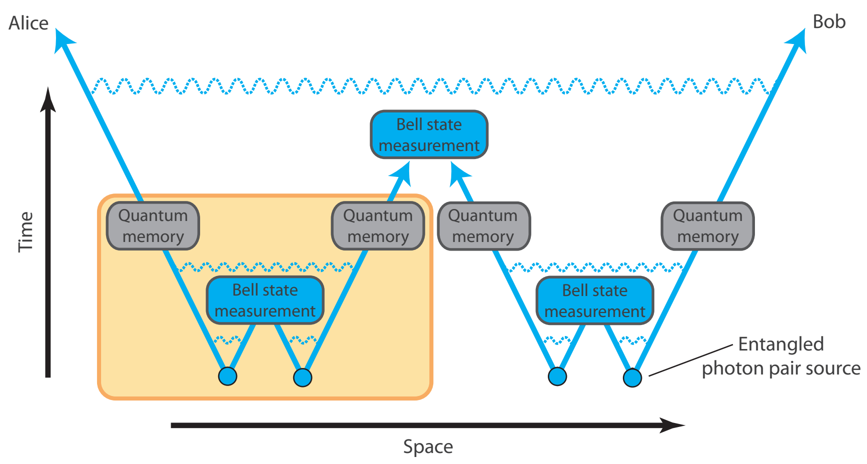


Figure 1: Quantum repeater schema (adapted from [7]). The entangled photon pair source emits two entangled photons. The entanglement is illustrated with the blue dotted line. The inner photon pair of one quantum repeater (orange box) undergoes a joint bell state measurement. As a result of this measurement the two outer photons are entangled. This figure shows two quantum repeater nodes. In principle several nodes can operate between Alice and Bob. The quantum memories save the state if the entanglement swapping was successful.

1.2 Focus of this Work

The Nano Optics group works on the realization of the quantum repeater based on semiconductors. For this purpose suitable QDs are needed [7]. These QDs have to be characterized experimentally. This is done in an optical setup consisting of a cryostat with a motor to move the sample, a spectrometer, avalanche photo diodes (APDs) and an interferometer. With this setup the spectrum, the coherence length and the correlation function of the emitted photons and the life time of the excitons in single QDs can be determined. The QDs are distributed over the entire sample. So it would take weeks to scan one single sample manually. To reduce the time this cumbersome manual search process has to be automated. The part of the setup which can be automated is shown in Figure 2.
An automation could be done with different programming languages. LabVIEW is a visual program language which seems to be very simple in the beginning, i.e. when writing first, simple programs. However, LabVIEW has several disadvantages. The main disadvantage is that small changes in the program may imply major structural problems. Furthermore, it is very difficult to understand existing code.
For these reasons another programing language, Python, was chosen. The main advantage of Python is that the code is very well readable and thus understandable [8]. So others, who have not written the code themselves can grasp the logic of the program relative quickly and can easily extend the program. This is a very important feature for scientific programs because the user changes frequently. Furthermore, Python is independent of the operating system and it is compatible with other program languages. Moreover, it is a free and open source software which results in a fast support. Additional functionality can be added to Python through the so called packages. There are many different packages for many different tasks, such as data analysis and numerics.
In this thesis it is explained how to write a scientific application with Python. The basics of Python are not discussed, since there are already many very useful tutorials and books about it. Some are recommended here. A good beginners book is 'learning Python' written by Lutz Mark [8]. A German book with helpful examples is 'Python: Das umfassende Handbuch' of Johannes Ernesti and Peter Kaiser [9]. The latter one can be accessed freely on the open book homepage. The Python documentation provides a very helpful tutorial as well [10, 11]. Of course there are a lot of other very helpful Python tutorials.
Python is easy to learn. Especially for those who already know another programming language. Since this thesis addresses other students at the bachelor level, it is assumed that the reader is familiar with the concepts of programming. However, many programming languages are not object oriented. Therefore the concept of object oriented programming (OOP) is introduced in Chapter 2. This is important for understanding the other chapters, since the fundamental design of the programs is based on OOP. Chapter 3 presents fundamental elements of writing a graphical user interface (GUI) with the **traits** package. Different items like buttons, text fields and check boxes are introduced. Afterwards plotting with **chaco** and some advanced **traits** topics are discussed. Most devices in this setup can be controlled with the serial port. Chapter 4 shows how this is realized in Python.

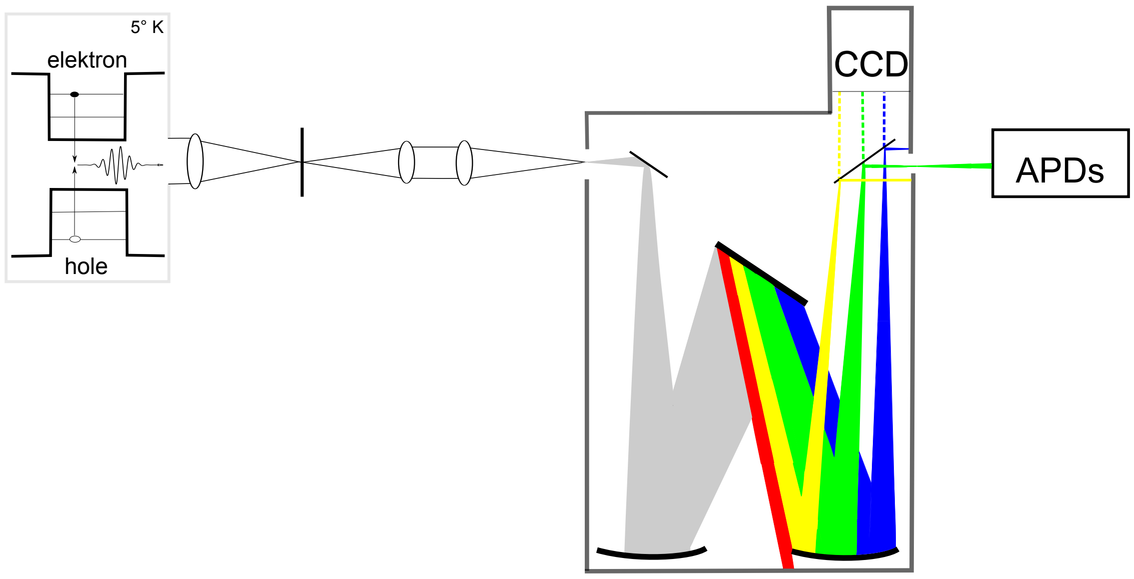


Figure 2: Sketch of the experimental setup. The sample with the QDs is inside a cryostat with motorized stage and is cooled down to 5 °K. The QDs are excited with a Laser. The emitted QD light is collected with an objective and guided through a lens system with a pinhole. Afterwards it reaches a spectrometer (Princton Instruments SP2500), where it is spectrally dispersed. Then there are two different possible light paths. In the first the spectrum is measured with a camera (Andor iDus DU 401A). The second possibility is that the light leaves the spectrometer through the exit slit and is guided onto the APD. The APD count the number of photons in the wavelength band selected with the spectrometer. The counts are converted to a voltage which can be read out with a microprocessor (AVR Atmega32). The motorized stage of the cryostat, the spectrometer, the camera and the microprocessor can be automated.

Since the program and its development is bound to the physical instruments, some time is spend on writing a simulation wrapper for the instruments. So that the program can be written, tested, and debugged without occupying measurement time and lab equipment. How such a simulation with help of the OOP concept can be written is explained in Chapter 5. The camera of the spectrometer cannot be controlled with the serial port. For this reason, Chapter 6 explains the control of the camera with an existing dynamic link library (DLL). Finally the scanning algorithm to find the QDs at the sample is discussed in Chapter 7.

Note these grey boxes. They contain helpful additional information.

2 Python Classes and Inheritance

This chapter explains the basics of OOP and how to create classes in Python. Then a description of inheritance and its advantages follows. An example of overriding methods is given in Chapter 5.
OOP provides a very effective way of programming [8]. Thereby the number of code lines can be reduced, which makes it is easier to grasp the content. Furthermore, by writing a new program the existing code of another program can be used without copying it. In addition any functionality of the program can be overridden without changing the existing code. Taken as a whole, OOP structures the code in a more efficient way than only functions and procedures would do. Python is completely based on OOP, all variables are objects.

2.1 Classes

In simple terms, classes are packages of functions, variables and other objects. A class can be defined with the **class** statement. After that statement the objects of a class can be declared as usual.

```
class CryoGUI():
  x=5
```

This is just a class definition, to use a class one needs to create an instance of it. Just as the instance of the object **int** is created via **i=int()**, an instance of the (class-)object **CryoGUI** is created via **cryo=CryoGUI()**. With that instance all attributes can be used as usual.

```
print cryo.x
>>>5
```

Grouping attributes in objects and addressing them with

```
object.attribute
```

is one of the key principles of OOP. This means that Python searches in **object** for **attribute**. If Python finds **attribute**, it returns the value of **attribute**.

In contrast to other programming languages, Python allows direct accesses and changes of class definitions. One accesses the definition for example

```
CryoGUI.x=6
print CryoGUI.x
>>> 6
```

One has to keep in mind that this should only be used if one wants to change the definition of a class. It is meant to use the methods and attributes directly.

Functions can be defined in a class as well. Within class definition functions get an additional first argument called **self**. With **self** the other attributes of that class can be used.

```
class CryoGUI():
  x=5
  def show(self):
    print self.x
```

The function **show** does not know the attribute **x**. However, **show** can use **x** with **self.x**. This works because **self** refers to the parent class of **show**, namely **CryoGUI**. More precisely it refers to the actual instance of **CryoGUI**.

```
cryo=CryoGUI()
cryo.x=6
cryo.show()
>>>6
```

self in **cryo.show** refers to **cryo**, and therefore **show** shows **cryo.x**(=6) and not **CryoGUI.x**(=5).
Note that **self** is omitted when calling **cryo.show**, this is because Python will replace the function call internally with

```
cryo.show()->CryoGUI.show(cryo)
```

Note that it is not necessary to name the additional argument **self** but it is a very strict convention. So **self** should be used. Otherwise it is very hard for other people to read the code.

2.2 Inheritance

Inheritance means that preexisting classes, which are called parent classes, base classes or superclasses, give theirs properties and attributes to other classes, called child classes, derived classes or subclasses. Classes and inheritances are very useful, because the behavior of one class can be given to another without copying the code. To inherit a class, the parent class will be written in brackets by defining the child class.

```
class CryoGUI(HasTraits):
```

Here **CryoGUI** is the child class and **HasTraits** is the parent class. A child class can have more than one parent class. For that the parent classes have to be listed in the brackets of the child class, separated by commas. The order of the parent classes is important. If two parent classes have the same attribute, the first class overrides the attribute of the second class and so on.
Before it was mentioned that Python will search for **attribute** in **object** when **object.attribute** is called. Now with inherited classes Python will search not only **object** but all parent classes of **object**. There is a strict search order from bottom to top.

In OOP a lot of codes can be reused without copying them. There are a lot of packages which can be used and adjusted to the actual problem. Thereby the original code does not have to be changed. Therefore many problems can be solved with less work. Furthermore, updates become easier because only the relevant class has to be changed and not every file.

3 traits Introduction

traits is a Python package for GUI development. With **traits** GUIs are created quick and easily while the code remains readable, which is one of Pythons main advantages. Many other GUI toolkits have an external program to create the GUI. **traits** focuses on the simple and quick visualization of attributes and rapid prototyping. **traits** is a part of the Enthought Python Distribution (EPD). This Python distribution made available different scientific tools. For data analysis and visualization, for example [12]. The advantages of the EPD is that it presents almost all of the here required packages. However, all packages such as **traits**, **traitsui** and **chaco** can be downloaded and installed without EPD.
In this section the reader should learn what is **traits** and why it is so useful. Therefor buttons, text fields and check boxes will be introduced. It is shown how these elements can be assembled into a GUI. To use the GUI well it is essential to know how the GUI can call functions. For that the methods **fired** and **changed** will be described. Then a short introduction of plotting with **chaco** is following. At the end some advanced topics like event handle and menus are discussed.

3.1 First Steps with traits

traits provides a graphical interface for Python programs and a couple of functions to control this interface. To use them, the **traits** packages have to be imported [13].

```
from enthought.traits.api import *
from enthought.traits.ui.api import *
```

Using **import** * loads all methods of the packages. Instead of using * it is more elegant to list only the required parts of the package and load them.

In **traits** the class **HasTraits** provides the objects to create a GUI. To use these objects **HasTraits** is inherited:

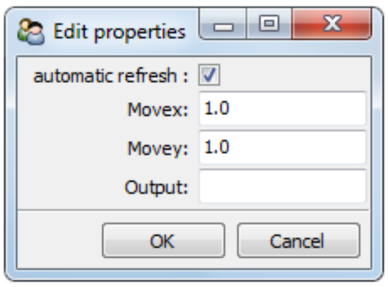
```
class CryoGUI(HasTraits):
  output=Str()
  movex=CFloat(1.0)
  movey=CFloat(1.0)
  move=Button()
  up=Button(label ='^')

  checkbox=Bool(True,label ='automatic refresh ')
```

The **Str** command defines a string and **CFloat** defines a float. These **HasTraits** items will be text fields at the GUI. In the brackets a starting value can be given. Furthermore, a button can be created with **Button**. This button can be clicked at the GUI. The **Bool** command defines a Boolean variable. It is displayed as a check box, which can be checked and unchecked by clicking into the box.
To call the GUI **HasTraits** has the **configure_traits** method. First an instance of the class should be defined. Then the attribute **configure_traits** of this instance can be called.

```
main=CryoGUI()
main.configure_traits()
```

Here the GUI can be displayed with **main.configure_traits**.



automatic refresh : ✓
Movex: 1.0
Movey: 1.0
Output:

Figure 3: GUI without **View**. The items are ordered alphabetically.

This simple example shows the power of **traits** only few lines of code are enough to create a complete user interface. The items of Figure 3 are ordered alphabetically. To change that order an instance of **View** is needed. With **View** the arrangement of the GUI can be determined. Thereto the different objects can be embedded with **Item('object_name')** at the **View** object.

```
traits_view = View(
        Item('movex',label ='x'),
        Item('movey',label='y'),
        Item('up',show_label=False),
        Item('output',style ='readonly'),
        Item('checkbox')
        )
```

Here it is possible to change the **label** of the objects or disable the label with **show_label=False**. In Figure 3 the object output is writable, too. However, to display only the output of the program it is more useful if it would not be writable. Therefore **style ='readonly'** can be used. The result of this **View** is shown in Figure 4.

Usually the button is labeled with its variable name. This can be changed with the keyword **label**. By each click on the check box the value of the Boolean variable is changing, too. A possible start value of **Bool** is **True**. It stands for a checked box. Another possibility is **False** with an unchecked check box.

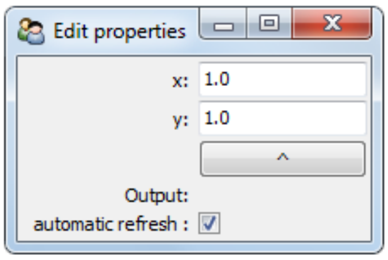


Figure 4: GUI with different items. A **View** is used to arrange the items.

The result of the GUI with this **View** is shown in Figure 4. However, the items cannot be displayed next to each other. That is why different items can be formatted together with **Group**. Thereby the items are handled as one object. For **Group** the subclasses **HGroup** and **VGroup** exist. The difference between them is the orientation. **HGroup** arranges the items horizontally and **VGroup** vertically. By combining these two commands the graphical surface can be determined (see Figure 5).

```
traits_view = View (
  VGroup(
    HGroup(Item('movex',label ='x'),Item('movey',label ='y')),
    HGroup(Spring(),Item('up',show_label=False),Spring()),
    Item('output',style ='readonly'),
    Item('checkbox')
    ),
  resizable = True , width = 300 , height = 150)
```

Usually there shall be space between two items. For that **HasTraits** has the object **Spring**, which works as a placeholder. For this purpose **Spring** is simply used instead of **Item**. The size of the window can be set with the keywords **width** and **height**. Without these statements **traits** defines the size automatically. With **resizable=True** the user can change the size by himself. Furthermore, **item(..., resizable=True)** can be used, too. Then the items grow and shrink with the size of the GUI.

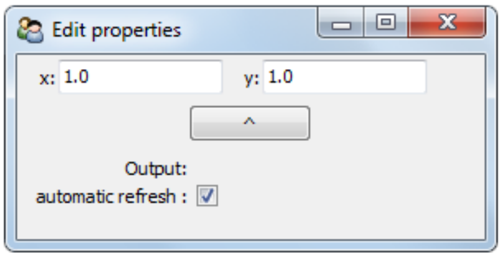


Figure 5: Using **View** with **VGroup**, **HGroup** and **Spring** to arrange the items.

So far, **View** is saved in a variable, which is named **traits_view**. The program still works if the name of this variable is different. However, if in one class more than one **View** is defined, **configure_traits** will use the variable **traits_view**. That is why usually **traits_view** is used for the main window. If there are multiple **View** and none of them is named **traits_view**, **traits** will arrange the objects alphabetically. To choose another **View** as **traits_view** **configure_traits(view='view_variable')** can be used. This is useful if more than one **View** is defined in a class [14].

3.2 User Interaction

So far, a couple of objects and the positioning have been explained. However, the program is not yet interactive. There are two important methods to control the GUI. The first is **_up_fired** which is called always when the user clicks on the button **up**. The second function is **_movey_changed**. It is called every time when the variable **movey** is changing its value.

```
def show(self):
  self.output ='x: '+str(self.movex)+ ' y:'+str(self.movey)

def _up_fired(self):
  self.movex=self.movex+1

def _movey_changed(self):
  if self.checkbox:
      self.show()
```

Whenever the button **up** is pressed or **movey** is changed the corresponding function will be called, and the GUI will be updated.
The complete code of the last two sections (Listing 1) creates the GUI (Figure 6) for controlling the cryostat motor movement.

A **changed** method cannot call itself but another **changed** method can call it. So one has to be careful when changing values in a **changed** method. Note the underscore at the beginning of the function name. Without this underscore the function does only work with normal function calls.

```
from enthought.traits.api import *
from enthought.traits.ui.api import *

class CryoGUI(HasTraits):
  output=Str()
  movex=CFloat(1.0)
  movey=CFloat(1.0)
  move=Button()
  up=Button(label ='^')
  down=Button(label ='v')
  right=Button(label ='>')
  left=Button(label ='<')

  position=Button()
  identity=Button()

  checkbox=Bool(True,label ='automatic refresh ')

  traits_view = View (
    VGroup(
      HGroup(
        Item('movex',label ='x'),
        Item('movey',label ='y')),
      HGroup(
        Item('position', show_label=False),
        Spring(),
        Item('identity', show_label=False)),
      HGroup(
        Spring(),
        Item('up',show_label=False),
        Spring()),
      HGroup(
        Item('left',show_label=False),
        Spring (),
        Item('right',show_label=False)),
      HGroup(
        Spring (),
        Item('down', show_label =False),
        Spring())
        ),
    Item ('output',style ='readonly'), Item ('checkbox'),
    resizable = True , width = 350 , height = 250)

  def show(self):
    self.output ='x: '+str(self.movex)+\
                ' y: '+str(self.movey)

  def _identity_fired(self):
    self.output=' a identify output '

  def _position_fired(self):
    self.show()

  def _up_fired(self):
    self.movex=self.movex+1

  def _down_fired(self):
    self.movex=self.movex-1

  def _left_fired(self):
    self.movey=self.movey-1

  def _right_fired(self):
    self.movey=self.movey +1

  def _movex_changed(self):
    if self.checkbox :
      self.show()

  def _movey_changed(self):
    if self.checkbox:
      self.show()

  def _checkbox_changed(self):
    pass

main=CryoGUI()
main.configure_traits()
```

Listing 1: An executable example of **traits**. Different items are displayed at the GUI. The order of these items is determined in **View**. Furthermore, **changed** and **fired** are used to control the program with this GUI. A picture of the GUI is shown in Figure 6.

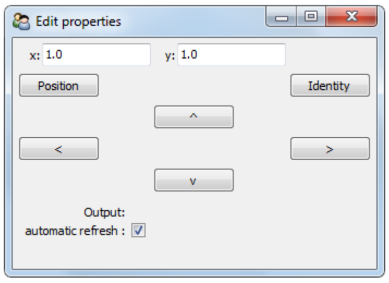


Figure 6: GUI to control cryostat stages. In the two input fields the value of **x** and **y** can be changed. The buttons with the arrows change the value by a constant step. There are two possibilities for a output. The first is to click the **identify** button, which returns an identification string. The second is the **position** button. It returns the current values of **x** and **y**. If the check box is checked, the output is refreshed automatically when **x** or **y** is changing.

3.3 Plotting with Chaco

After introducing **traits**, **chaco** will be explained. **chaco** is a part of the EPD and is build for 2d plots. The advantage of **chaco** is that **traits** integrates smoothly with **chaco**. To use **chaco** additional imports are necessary [15].

```
from enthought.chaco.api import Plot, ArrayPlotData
from enthought.enable.component_editor import ComponentEditor
```

Similar to the last section the class **PlotView** is defined. The difference is, that with **Instance(Plot)** an instance of **Plot** is created. Without this instance there would be no object that can be added to **View**. To display an instance of **Plot** with **View** an editor has to be chosen. Without the right editor **View** would not be working, because **traits** does not know how to handle **chaco** components.

```
class PlotView(HasTraits):
  plot = Instance(Plot)

  traits_view=View(
    Item('plot',editor=ComponentEditor(),show_label=False)
    )
```

Thereby the plot window can be displayed. So far it is only an empty box. First the data which shall be drawn (here **x_data** and **y_data**) have to be grouped together in one array. This is done with the function **ArrayPlotData(x=x_data,...)**. Here an additional dataset **z_data** is used for illustration.

```
plotdata = ArrayPlotData(x=x_data, y=y_data, z=z_data)
tmpplot = Plot(plotdata)
```

ArrayPlotData takes any number of arguments but it always needs an assignment. Here **x** is the variable in which the data **x_data** are saved in the array.

With this array a **Plot** can be created. Therefor **Plot** has to be called with the array. Now one can choose which data shall be plotted.

```
tmpplot.plot(('x', 'y'), type='line', color='blue')
```

Note that **x** and **y** are the variables which are used by **ArrayPlotData**. **z** is not used but it could be instead of **x** or **y**. Furthermore, every permutation would be possible because they are only variable names which were chosen in **ArrayPlotData**. To refresh the GUI with the drawn plot only

```
self.plot=tmpplot
```

is needed, because at the beginning the plot instance was created and added to the **View** method.
The **tmpplot** object saves not only the data which shall be plotted but all of the settings of the plot. These settings, like the title or the label of the axes, can be changed, too.

```
tmpplot.title='Plot of Random Values'
tmpplot.x_axis.title='Wavelength [nm]'
tmpplot.y_axis.title='Intensity [V]'
```

Two useful tools are **ZoomTool** and **PanTool**. **ZoomTool** allows to zoom in and out of the plot. However, zooming is only really useful if the plot can be moved. Therefore **PanTool** exists. It allows to move the plot. To use these tools, the two items have to be imported.

```
from enthought.chaco.tools.api import PanTool, ZoomTool
```

Then they can be appended simply to the **Plot** instance.

```
tmpplot.overlays.append(ZoomTool(component=plot,
              tool_mode='box', always_on=False))
tmpplot.tools.append(PanTool(plot, constrain_key='shift'))
```

The most important aspects of plotting with **chaco** have been shown. In Listing 2 random values with the **sample** command are plotted. To use random functions additional imports are needed. Figure 7 shows the result of Listing 2.

```
from enthought.traits.api import *
from enthought.traits.ui.api import *
from enthought.chaco.api import Plot, ArrayPlotData
from enthought.chaco.tools.api import PanTool, ZoomTool
from enthought.enable.component_editor import ComponentEditor
from random import sample
```

```
class PlotView(HasTraits):
  button=Button('Plot random values')
  plot = Instance(Plot)

  traits_view=View(Item('plot',editor=ComponentEditor(),
                        show_label=False),
                   Item('button',show_label=False))

  def _button_fired(self):
    a=range(1,20)
    b=sample(range(100),19)
    self.draw(a,b)

  def draw(self,x_axis,y_axis):
    plotdata = ArrayPlotData(x=x_axis, y=y_axis)
    tmpplot = Plot(plotdata)
    tmpplot.plot(('x', 'y'), type='line', color='blue')
    tmpplot.title='Plot of Random Values'
    tmpplot.x_axis.title='Wavelength [nm]'
    tmpplot.y_axis.title='Intensity [V]'
    self.plot = tmpplot

plotview=PlotView()
plotview.configure_traits()
```

Listing 2: Example of plotting with **chaco**. By each click at the **button** new random values are created and plotted with **chaco**. A result of this method is shown in Figure 7.

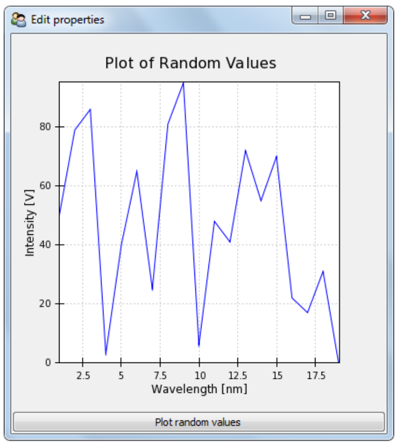


Figure 7: Example use of **chaco**. By clicking on the button random values are generated and plotted (see 2).

3.4 Handling events in Plots

chaco has an amazing possibility for an event handle at its existing Plots [15]. The class **BaseTool** provides a lot of functions, which are called by different events. Here some are listed: **normal_left_dclick** is executed if the user double clicks with the left mouse key on the plot. Similar **normal_right_down** is working. The difference is that one right click executes the function. Futhermore, **normal_mouse_move** is always running if the mouse is moving over the plot. All these functions need, next to the **self**, an additional argument called **event**.

```
from enable.api import BaseTool
class PlotTool(BaseTool):
  def normal_mouse_move(self,event):
    [x,y]=self.component.map_data((event.x,event.y))
    print 'position x: '+str(x)+'y: '+str(y)
```

event.x and **event.y** are the x and y coordinates of the window with the origin of ordinates in the lower left corner. To get the coordinates of the plot **BaseTool** has the function **component.map_data**. The event handle is appended to an existing plot, just like **ZoomTool** and **PanTool**.

```
tmpplot.tools.append(PlotTool(component=plot))
```

3.5 traits Advanced

In this section some advanced topics such as drop-down menus, menu bars and event handles are discussed. Finally it is shown how a GUI can be imported and used.

3.5.1 Drop-Down Menu

At the spectrometer different gratings can be chosen. The gratings determine the resolution of the spectrometer. A grating with more lines can spread different wavelength stronger. Therefore with the same pixel size of the camera one can distinguish smaller wavelength differences. The selection of the gratings shall be realized with a drop-down menu. Therefore the different values of the drop-down menu have to be listed in **List**. The current value of a drop-down list is declared as **Str**.

```
grating_value=List(['100 g/mm','200 g/mm','300 g/mm'])
current_grating=Str('100 g/mm')
```

The drop down menu is displayed by calling **current_grating** as a normal **Item** with **EnumEditor** as **editor**.

```
traits_view=View(Item('current_grating',
        editor=EnumEditor(name='grating_value')))
```

The values of the list are selected by **EnumEditor** with the keyword **name**. The result of the GUI is shown in Figure 8.
If another value of the list is picked, the current value is changing automatically. That is why the **changed** method can be used to recognize that change. An executable example of using a drop-down menu is shown in Listing 3.

It is possible to create a drop-down list with **Enum**, too.

```
grating=Enum(['100 g/mm','200 g/mm','300 g/mm'])
traits_view=View(Item('grating'))
```

The current value of the list can be set and read by using the variable **grating**. This method looks simpler and slightly more intuitive. The disadvantage is that the values of the list cannot be changed simply, while the program is running. The possibility to update the list is preferable. Otherwise the drop-down menu cannot use the values of a device. For that reason this method is not used any further.

It is difficult to give the list a constant start value because the current value of the device could be another than expected. Without a given start value the first value of the list will be empty until another value is chosen. One possibility to solve that problem is to change the current value with one out of the list. This could be done in the **__init__** function, for example. The different values can be read of the device. The marked one of these values is the current value. So the program can find this mark and chose the corresponding value.

```
from enthought.traits.api import *
from enthought.traits.ui.api import *

class spectroGUI(HasTraits):
  grating_value=List(['100 g/mm','200 g/mm','300 g/mm'])
  current_grating=Str()
  traits_view=View(Item('current_grating',
      editor=EnumEditor(name='grating_value')))

  def __init__(self):
    #initializes spectrometer, reads the current grating
    #and saves it in grating
    #dummy grating='100 g/mm'
    grating='100 g/mm'
    self.current_grating=grating

  def _current_grating_changed(self):
    print self.current_grating

main=spectroGUI()
main.configure_traits()
```

Listing 3: Example of using a drop-down menu with **traits**. By every change of the drop-down menu the current value is printed to the terminal. An image of this GUI can be seen in Figure 8.

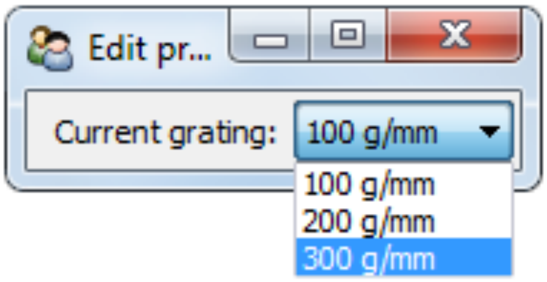


Figure 8: Drop-down menu with different values as generated in Listing 3.

3.5.2 Menubar

By complex programs not all objects should be displayed at the main GUI. Usually a menu at the top of the GUI can open more windows with some advanced settings. Such a menu can be created with the **Menubar** command. However, first an object has to be created, which defines a function call, when the user clicks at the menu item. Therefor **Action** is responsible.

```
menu_item = Action(name='cryo menu',
      accelerator='Ctrl+m', action='item_action')
```

Action has three important keywords. As usual **name** sets the displayed name. With **accelerator** a shortcut of this menu can be chosen. Furthermore, **action** defines which function is called when the menu item is clicked.
To make a new menu item, **Menu** is called. In first place the different objects for the menu can be listed. At the end the keyword **name** defines the name of the menu item.

```
menu = MenuBar(Menu(menu_item, name='Cryo'))
```

Here the embedded object is **menu_item** and the chosen **name** is 'Cryo'.
In **View** the menu can be linked with the keyword **menubar**.

```
traits_view = View(...
    menubar=menu,...)
```

The action, which is performed when **memu_item** is clicked, is defined in the function **item_action**:

```
def item_action(self):
  self.cryo_instance.configure_traits(view='view_menu')
```

Here the function calls another GUI of **cryo_instance**. This GUI and all functions to control the GUI is declared in the cryostat module. Therewith all settings of the second **View** are defined in the cryostat module and could be used by every other module.

The keyword **kind** can be used in **View** to customize the properties of the window. With **kind='modal'** the second window is always in the foreground until it is closed. Modifications are sent to the main window after closing the second window. If the changes should be sent instantly, the window has to be **live**. Consequently **kind='livemodal'** is the correct specification for a window which is **modal** and **live**. If the window should be neither **modal** nor **live**, only **kind='nonmodal'** is used [16].

3.5.3 Closing handle

To reduce thermal noise the camera is cooled down to -75 °C during a measurement. The camera is slowing the cooling and heating process to reduce stress. If the camera is simply shut down, it cannot control this process any longer [17]. That is why the program has to run until the camera is above 0 °C. For that a handler has to be defined. The **Handler** class has a method **close** which will be executed every time if the user tries to close the window. If **close** returns **True**, the window will be closing otherwise it remains open [18].

```
class MainWindowHandler(Handler):
  def close(self, info, isok):
    if camera.gettemperature() >0:
      return True
    else:
      print('Please wait until the temperature'\
    +' is above 0 degrees Celsius.')
```

The method **gettemperature** reads out the current temperature of the camera and returns its value. If the temperature is above 0 °C, the function **close** returns **True**. A handler for a window can be chosen at the **View** command. For that the keyword **handler** is existing.

```
traits_view = View(...
    handler=MainWindowHandler(),...)
```

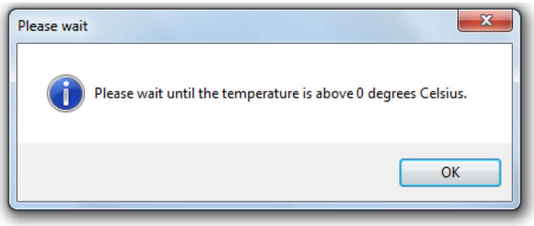
The information, that the temperature is too low, is printed in the terminal. The user will probably not see this information. An additional window, which shows this information, would be more elegant. **pyface** provides such windows.

```
from enthought.pyface.api import information

information(parent=None, title='Please wait',
    message='Please wait until the temperature'\
    +' is above 0 degrees Celsius.')
```

The keyword **message** changes the displayed text and **title** changes the name of the window (see Figure 9).

pyface provides more than an information window. Other possibilities are **error** and **warning**. They have another icon as **information**. To use another icon **information** has to be replaced with **error** or **warning**.



Please wait until the temperature is above 0 degrees Celsius.

Figure 9: An information window created with **pyface**.

```
from enthought.pyface.api import information
from enthought.traits.api import *
from enthought.traits.ui.api import *

class MainWindowHandler(Handler):
  def close(self, info, isok):
    if camera.gettemperature() >0:
      return True
    else:
      information(parent=None, title='Please wait',
          message='Please wait until the temperature'\
          +' is above 0 degrees Celsius.')

class Camera(HasTraits):
  temperature=CFloat(-20)

  traits_view = View(Item('temperature'),
    handler=MainWindowHandler())

  def gettemperature(self):
    return(self.temperature)

camera=Camera()
camera.configure_traits()
```

Listing 4: Handler example which is running when the user tries to close the window. The window closes only if the **close** method returns **True**. Here the function **gettemperature** reads the current temperature of the camera. If the temperature is too low, an information window as in Figure 9 appears.

3.5.4 Import a GUI

So far code examples for addressing single devices were discussed. For the automated search for quantum dots these different devices need to be combined into a single program. To this end, several GUIs can be loaded and combined into a single **View**:

```
spectrometer_instance=Instance(SpectrometerGUI ,())
cryo_instance=Instance(CryoGUI ,())
traits_view = View(
  Item('cryo_instance',style='custom',
      show_label=False,label='cryo'),
  Item('spectrometer_instance',style='custom',
      show_label=False,label='spectrometer'))
```

Of course the same keywords as usual can be used to define the **View** object. It is important to use **style = 'custom'**. Without this, the instances would not be displayed together but two buttons would be shown. One button belongs to one instance and would open the associated GUI.

So far, every single program included its own GUI. At the end of each program **configure_traits** is called to display the GUI. When combining several GUIs the individual GUIs do not have to be launched individually. Therefore it is useful to call **configure_traits** only in the individual files, when it is called independently. To this end, the **configure_traits** call is changed to:

```
if __name__=='__main__':
  main.configure_traits()
```

where **__name__** equals '**__main__**' when the file is executed directly [8]. If the file is imported into another program **__name__** equals its module name.

4 pySerial

Usually in an experiment different instruments are controlled by a computer. Many of them can be controlled by the serial port.
In Python the serial port can access via **pySerial** [19]. **pySerial** is versatile, because it is running on Windows, Linux, Jython [1] and IronPython[2]. The module of **pySerial** is named **serial**.
The first part of this chapter is an introduction to **serial**. It is explained how a serial port can be opened and closed. Furthermore, the communication between the computer and the devices is illustrated. The second part of this chapter introduces buffers.

4.1 Introduction to pySerial

After the import of **serial** a serial port can be opened.

```
import serial
spectro=serial.Serial('COM4', 9600, timeout=1)
```

The command **spectro = serial.Serial('COM4',9600, timeout = 1)** opens the named port 'COM4', with a baud rate of 9600 Bd [22]. The baud rate is a rate of a number of symbols per second. It is important for the communication between the computer and an instrument. Without the same baud rate for sender and receiver the communication cannot work correctly. The baud rate of an instrument is generally given in the manual.
If **serial** does not find enough characters or an end of line (EOL), the keyword **timeout** becomes relevant. **serial** stops the reading process after **timeout** seconds. This can happen at the end of a file, for example. If the keyword **timeout** is not used, no timeout will be defined. So it is possible that the program blocks the serial port while trying to read.
serial can operate on an open port with several commands. Two important commands are **write** and **read** (or **readline**). At first a command is send to the spectrometer ('?NM \r') to query the current wavelength.

```
spectro.write('?NM \r')
```

Then to read its respond wavelength **readline** is used.

```
output=spectro.readline()
print output
```

At the end

```
spectro.close()
```

closes the serial port , so that other programs may access it.

[1]Jython is a Java implementation of Python with the result that Python programs run at Java [20].
[2]IronPyton is a implementation of Python for the .NET Framework [21].

Note that the string has to be formatted properly otherwise the device cannot handle the string. The needed formatting is given in the manual, too. For example, the instruments often need an EOL like a '\r' or '\r\n'.

4.2 Buffer and pySerial

The center wavelength of the spectrometer is adjustable. The command to change the wavelength is 'GOTO'. To change the wavelength to 550 nm the string '550 GOTO \r' has to be send to the spectrometer. If the current wavelength is read out as before (see Listing 5), the result is not the expected wavelength. The output will be '550 GOTO Ok'.

```
import serial

spectro=serial.Serial('COM4', 9600, timeout=1)
spectro.write('550 GOTO \r')
spectro.write('?NM \r')
output=spectro.readline()
print output
>>>550 GOTO Ok
spectro.close()
```

Listing 5: Example which shows that the last requested information is not the first readable information. The output will be a confirmation string of the spectrometer because it changed its center wavelength and not the expected current wavelength.

This behavior can be explained simply. The communication of the device and the computer was described only by writing to and reading from the serial port. However, the serial port saves the whole communication between the computer and the device in a buffer. Often there is more output of the device than expected. This can happen if some information were requested but not read out. Another possibility is that some devices are sending a confirmation string to the serial port when they have finished their task or the reply of the instrument is not well defined. So it is possible that the requested information comes later than expected. The spectrometer sends such a confirmation string. This explains the output of Listing 5.
The problem in Listing 5 could be solved with a **readline** after the first **write**. For more complex programs this is very impractical because it needs the accurate numbers of **readline** to get the correct information. Of course the whole buffer can be read out and analyzed but this might be inefficient.
A more elegant way is to clear the buffer with the serial command **flushInput** [23]. It should be used every time before a command requests an information from the device.
With **flushInput** Listing 5 can be corrected so that the current position is printed.

```
import serial

spectro=serial.Serial('COM4', 9600, timeout=1)
spectro.write('550 GOTO \r')
spectro.flushInput()
spectro.write('?NM \r')
print( readline())
>>>?NM 550 nm Ok
```

flushinput is named input because it clears the input of the computer. Analogical the output of the computer can be cleared with **flushOutput**.

5 SimSerial

While writing a program with **serial** it is useful if the program may run without the device. Then the program can be developed outside of the laboratory. Furthermore, a device, which does not work, could prevent the program from working. Therefore it makes sense to have a simulation environment.
First, this chapter explains how such a simulation can be written. Afterwards an introduction to **SimSerial** is represented. Then an example of using **SimSerial** is shown.

5.1 How to write a Simulation

One elegant way to realize the simulation with the OOP is by overriding the commands of **Serial**, which is the class of **serial**. **SimSerial** has to be a child class of **Serial** so that **SimSerial** can override the behavior of **Serial**. The first method which has to be overridden is **__init__**. **__init__** is a special function of a class. It is executed always if an instance of this class is generated. In **Serial** **__init__** opens the serial port and starts to connect to the device but within a simulation this has to be skipped.

```
import serial

class SimSerial(serial.Serial):
  initargs=str()
  initkwargs=str()

  def __init__(self,*args,**kwargs):
    self.initkwargs=kwargs
    self.initargs=args
```

Besides **self**, the **__init__** method of **SimSerial** has two more arguments here, which are ***args** and ****kwargs** . The * in front of **args** means that all arguments which are used by a function call are saved in **args**. The arguments are saved in a list. In analogy ****kwargs** with ** saves the keywords, which are used by a function call, in a dictionary. Of course ***args** and ****kwargs** can be used by every other function, too.
If the serial port should be opened, the **__init__** of **Serial** has to be called. The **super** function can be used for that [24]. **super** looks at the parent class of a given object and calls its attribute.

```
super(child_object,self).attribute
```

With that method the **__init__** of **Serial** can be called.

```
super(SimSerial,self).__init__(*self.initargs,\
                              **self.initkwargs)
```

To distinguish between simulation and real interaction the attribute **simulation** is added to **SimSerial**. With **simulation** a function **toggle_simulation** can be written. This method opens the serial port if **simulation** is **False** otherwise it closes the port.

```
def toggle_simulation(self):
  if self.simulation:
    self.simulation=False
    print('simulation off')
    super(SimSerial,self).__init__(*self.initargs,\
                                  **self.initkwargs)
  else:
    self.simulation=True
    self.close()
    print('simulation on')
```

Here, the variables **initargs** and **initkwargs** , saved in **__init__**, are used to open the serial port with the right configuration.
SimSerial always starts in the simulation mode. Otherwise it would have to open the serial port like **Serial**. So far the simulation can be toggled on and off. The next step is to override **write** of **Serial**. The new **write** method shall execute the **write** method of **Serial** if **simulation** is **False**. Otherwise it should call a function for the simulation, which simulates the task of the device. As the name of these simulation functions varies, this must be done with **getattr** [24]. This function returns the attribute of an object.

```
getattr(object,attribute)
```

For a function call additional brackets, containing the arguments, are needed. With that concept **write** can be overridden.

```
def write(self,string):
  if self.simulation:
    name=self.search_function_name(string)
    try:
      getattr(self,name)(string)
    except:
      try:
        getattr(self,name)()
      except:
        print 'no simulation function found'
  else:
    super(SimSerial,self).write(string)
```

If the simulation is active, **search_function_name** provides the **name** of the simulation function. At this point it is not important to know how **search_function_name** works. That will be explained later in detail. Then **write** tries to call the simulation method **name**. The **try** statement tests whether the function call is available. If it is not available, it runs **except**. **write** tries to call the function first with a statement and then without a statement. If both do not work, a message is printed in the terminal.
To override **readline** is much simpler.

```
def readline(self):
  if self.simulation:
    return(str(self.buffer))
  else:
    return(super(SimSerial,self).readline())
```

If the simulation is active, **readline** returns **buffer**. Otherwise it executes the **readline** method of **Serial**. **buffer** is an attribute of **SimSerial** and works as a buffer of the serial port.
The complete **SimSerial** is shown in Listing 6. The method **search_function_name** is shown as well. This function separates the string by spaces. To find the command in this string part **search_function_name** uses two extra attributes of **SimSerial**. **commando_position** knows whether the keyword for the device is written at the beginning or at the end of the string. If it is written at the end, the EOLs are important. These are saved in **EOL**. It is important because it defines at which position **search_function_name** looks for the device command. With that information it is defined between which spaces the command is set. When the command is found, a '_' is added in front of the device command, to make it easier for the user to find the functions for the simulation. At the end of **search_function_name** special characters like '?' are replaced by letters in **replace_special_characters**. Summarized it can be said that **search_function_name** searches the device command and returns it with a '_' in front of it.

```
import serial

class SimSerial(serial.Serial):
  commando_position='first'
  EOL=''
  simulation=True
  initargs=str()
  initkwargs=str()
  buffer=str()

  def __init__(self,*args,**kwargs):
    self.initkwargs=kwargs
    self.initargs=args

  def toggle_simulation(self):
    if self.simulation:
      self.simulation=False
      print('simulation off')
      super(SimSerial,self).__init__(*self.initargs,\
                                     **self.initkwargs)
    else:
      self.simulation=True
      self.close()
      print('simulation on')
```

```
def write(self,string):
  if self.simulation==True:
    name=self.search_function_name(string)
    try:
      getattr(self,name)(string)
    except:
      try:
        getattr(self,name)()
      except:
        print('No simulation function found.')
  else:
    super(SimSerial,self).write(string)

def readline(self):
  if self.simulation:
    return(str(self.buffer))
  else:
    return(super(SimSerial,self).readline())

def flushInput(self):
  if not self.simulation:
    super(SimSerial,self).flushOutput()
  else:
    self.buffer=str

def search_function_name(self,command):
  if self.EOL!='':
  #deletes EOL out of the string if EOL not empty
    command=command.replace(' '+self.EOL,'')
  #split the string by spaces
  command=command.split(' ')
  #find name at defined position
  if self.commando_position=='first':
    name=command[0]
  else:
    name=command[-1]
  name=self.replace_special_characters(name)
  name='_'+name
  return(name)
def replace_special_characters(self,name):
  _sign=['?','!','+','-','>','/']
  _replace=['QM','EM','plus','minus','greater','slash']
  for i in range(len(_sign)):
    name=name.replace(_sign[i],_replace[i])
  return(name)
```

Listing 6: The behavior of a device can be simulated with **SimSerial**. Therefor it finds the keyword in the device command and call a function for the simulation.

In this Listing 6 **flushInput** is added, too. The command clears the buffer. If the simulation is running, the simulated buffer is cleared.

5.2 How to use SimSerial as a Module

To use **SimSerial** it has to be imported. If Listing 6 is saved as simserial.py in the same directory as the current module, it can simply be imported. If two different programs have to use the same module, it is not viable to use the same directory. Python will search for modules in the current directory and in all directories given in the **PYTHONPATH** variable.

To add the directory to windows click with right mouse key on *My Computer* and chose *properties*. In this window click on *Advanced system configuration*. For that admin rights are needed. Choose the tab *Advanced* and click on *Environment Variables*. There **PYTHONPATH** can be found and the directory of **SimSerial** can be added. In windows the different directories are delimited by a semicolon. If there is no **PYTHONPATH** it can be added simply, by clicking on *new*. Note that **PYTHONPATH** has to be written in capital letters [25].
In Linux it is easier to add a new directory to **PYTHONPATH**. Open **.bash_profile** in the home directory and add at its end 'export PYTHONPATH=$PYTHONPATH:foo/bar'. Then just save the file and restart the terminal [8].

So the simserial.py is saved in its own directory. This directory is then added to the **PYTHONPATH** variable. Afterwards the module can be imported in every directory.
All changes to the imported module are only refreshed after restarting the compiler. It is not practical to restart the compiler after every change. The **reload** method solves that problem. It ensures that these packages are reloaded, which means that they are compiled again. However, one has to be careful if the same module is imported and reloaded by more than one module. This can result in an error.

5.3 Example Usage of SimSerial

Now **SimSerial** can be inherited to another class. In Listing 7 it is the **Spectro** class, which belongs to a spectrometer.

```
import simserial
reload(simserial)

class Spectro(simserial.SimSerial):
  #for the simulation
  nm=float(0)
  commando_position='last'
  EOL='\r'

  def wavelength_goto(self,target):
    target=round(target,3)
    self.write(str(target)+' GOTO \r')

  def _GOTO(self,string):
    self.buffer=string+' ok'
    a=string.find(' ')
    self.nm=float(string[0:a])

  def output_position(self):
    self.flushInput()
    self.write('?NM \r')
    tmp=self.readline()
    return(tmp)

  def _QMNM(self,string):
    self.buffer=' '+str(self.nm)+' '
```

Listing 7: **Spectro** is a class to control a spectrometer. In this listing the class can change and return the center wavelength of a spectrometer. Furthermore it can simulate these two methods because it is a child of **SimSerial** (see Listing 6).

The **wavelength_goto** method sets a new wavelength and the **output_position** returns the current wavelength. **Spectro** is a child of **SimSerial** so that the overridden methods of **Serial** are inherited for **Spectro**, too. To simulate the behavior of the spectrometer the functions **_GOTO** and **_QMNM** are used.
If **wavelength_goto** is called, it writes the wavelength **target** and the keyword for the spectrometer 'GOTO'. **SimSerial** finds that keyword and calls **_GOTO** with the string as argument if a simulation is running. **_GOTO** looks for the wavelength and saves it in the variable **nm**. The same procedure is done by **output_position**. It writes '?NM' which **SimSerial** finds and calls **_QMNM**.

Here **SimSerial** replaces the '?' with 'QM' because special characters are not allowed in function names. The function_**QMNM** prepares a read out by writing on **buffer**. At the end of **output_position** a **readline** follows. That is why **SimSerial** returns **buffer** which was changed of _**QMNM** before.

In the **wavelength_goto** method **target** is rounded, because the spectrometer cannot handle numbers with more than three decimal places. If more than three decimal places are sent, the spectrometer goes to a wrong wavelength. Then it might happen that it tries to reach a very high or a negative wavelength.

So far a class controlling the spectrometer and a suitable simulation is defined. The aim is to control this class with a GUI. This can be realized with an instance of **Spectro**.

```
spectro=Spectro('COM4', 9600, timeout=1)
```

Remember that **Spectro** is an inheritor of **SimSerial**, which is a child of **Serial**. Therefore **Spectro** is called with the same arguments as **Serial**.

```
from enthought.traits.api import *
from enthought.traits.ui.api import *

import spectrometer
reload (spectrometer)

class SpectrometerGUI(HasTraits):
  spectro=spectrometer.Spectro('COM4', 9600, timeout=1)
  goto=Button()
  position=Button(label='?nm')
  checkbox=Bool(True,
            label='Simulation spectrometer')
  output=Str(label='Output')
  input_goto=CFloat(0.0)

  traits_view=View(HGroup(
                  Item('input_goto',show_label=False),
                  Spring(),
                  Item('goto',show_label=False)),
                  Item('position',show_label=False),
                  Item('output',style='readonly'),
                  Item('checkbox')
                  )
```

```
    def _goto_fired(self):
        self.spectro.wavelength_goto(self.input_goto)

    def _position_fired(self):
        self.output=self.spectro.output_position()

    def _checkbox_changed(self):
        self.spectro.toggle_simulation(self.checkbox)

main=SpectrometerGUI()
main.configure_traits()
```

Listing 8: This code creates a GUI to control a spectrometer. It used an instance of **Spectro** (see Listing 7) to call different functions of the spectrometer. The result of this GUI is shown in Figure 10.

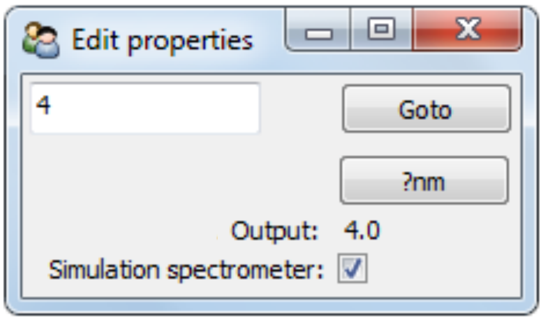


Figure 10: GUI to control the spectrometer. By clicking on the 'Goto' button the spectrometer is changing its center wavelength. The current center wavelength can be read out with the button '?nm'. If the check box is checked, the spectrometer is simulated. The code of the GUI is shown in Listing 8.

6 Controlling the Camera

Not all instruments can be controlled via the serial port. For example, the camera of the spectrometer is controlled via a dynamic link library (DLL). Python can use DLLs with the package **ctypes** [26]. It permits to call functions in DLL and provides C compatible data types.
At the beginning a short introduction of **ctypes** is given. Following, writing a control with a DLL of an Andor camera is explained.

6.1 ctypes Basics

ctypes has its own data types. They can be defined similar to other data types in Python.

```
from ctypes import c_int

number=ctypes.c_int(1)
print number
>>> c_long(5)
print number.value
>>>5
```

The value of **number** cannot be printed directly. Instead **ctypes** has the attribute **value**. Other **ctypes** data types are **c_long**, **c_float** **c_char** and **c_bool**.
An array in **ctypes** works slightly different.

```
from ctypes import c_float

length=5
line=(c_float * length)(1.3,2.5,5.6)
```

Here **length** defines the length of this array. In the second brackets start numbers can be defined. If fewer start values as the length of the array are given, the remaining values of the array are defined as zeros. Values of the array can be printed similar to Pythons list.

```
print line[0]
>>>1.29999995232
print line[:]
>>>[1.2999999523162842, 2.5, 5.599999904632568, 0.0, 0.0]
```

line contains some **ctypes** data but most Python functions cannot handle a **ctypes** object. For that reason it has to be converted to a Python type. For that each value must be converted separately.

```
from ctypes import c_float
line=(c_float * 5)(1,2,3,4)
spectrum=[i for i in line]
print spectrum
>>> [1.0, 2.0, 3.0, 4.0, 0.0]
```

6.2 Using a DLL File

After importing **ctypes** the DLL can be loaded with the command **WinDLL**, with the path of the DLL as argument. The result of this method has to be assigned to an object. Afterwards all functions of the DLL are attributes of this object.

```
from ctypes import *

atm=WinDLL('C:\Programs\Andor\ATMCD32D.DLL')
atm.Initialize(None)
```

Initialize initializes the camera [17]. Some Andor cameras need additional arguments to execute the **Initialize** method. The present camera is initialized with **None**.
To acquire pictures with the camera different methods of the DLL are used.

```
pixel=1024
line=(c_long * pixel)()
atm.StartAcquisition()
atm.WaitForAcquisition()
atm.GetMostRecentImage(byref(line),c_ulong(pixel))
atm.AbortAcquisition()
```

The program starts the acquisition and waits for its end. At the end the acquisition has to be abort, otherwise the camera cannot be controlled any longer.
Some C functions require pointers to variables as arguments. This is done by **byref**.
Listing 9 shows one way to control the camera, realized in a class structure.

```
from ctypes import *

class Camera():
  Cameras=c_long()
  full_vertical_binning_readmode=int(0)
  multi_track_readmode=int(1)
  random_track_readmode=int(2)
  single_scan_acquisitionmode=int(0)
  accumulate_acquisitionmode=int(1)

  exposuretime=c_float(0.1)
  acquisitionmode=single_scan_acquisitionmode
  readmode=full_vertical_binning_readmode

  def toggle_simulation(self,simulation):
    if simulation:
      self.close()
    if not simulation:
      self.atm=WinDLL('C:\Programs\Andor\ATMCD32D.DLL')
      self.atm.Initialize(None)
      self.atm.GetAvailableCameras(byref(self.Cameras))
      self.atm.SetReadMode(self.readmode)
      self.atm.SetAcquisitionMode(self.acquisitionmode)
      self.atm.SetExposureTime(self.exposuretime)

  def acquisition(self):
    pixel=1024
    line  = (c_long * pixel)()
    self.atm.StartAcquisition()
    self.atm.WaitForAcquisition()
    self.atm.GetMostRecentImage(byref(line),c_ulong(pixel))
    self.atm.AbortAcquisition()
    return(line)

  def close(self):
    self.atm.ShutDown()

  def settemperature(self,temperature):
    self.atm.SetTemperature(temperature)

  def gettemperature(self):
    temperature=c_long()
    self.atm.GetTemperature(byref(temperature))
    return(temperature.value)

  def cooler_on(self):
    self.atm.CoolerON()

  def cooler_off(self):
    self.atm.CoolerOFF()
```

Listing 9: Class with some functions to control the Andor camera. For that a DLL is loaded and used. The class **Camera** can acquire the spectrum and control the temperature of the camera.

7 QDSearch

The previous chapters illustrated the generation of different device controls. Now, all these control elements are combined in the measurement program **QDSearch**. **QDSearch** is a program to find and analyze quantum dots on a sample. It uses several devices such as a spectrometer include a camera, a cryostat with motorized stage and a microprocessor to systematically scan the sample. The interaction of **QDSearch** is shown schematically in Figure 11.
In this section an algorithm is discussed, which maximizes the throughput of the spectrometer for a given wavelength range. So the spectrometer can be used as a monochromator. Afterwards the scanning algorithm of a QD sample is illustrated.

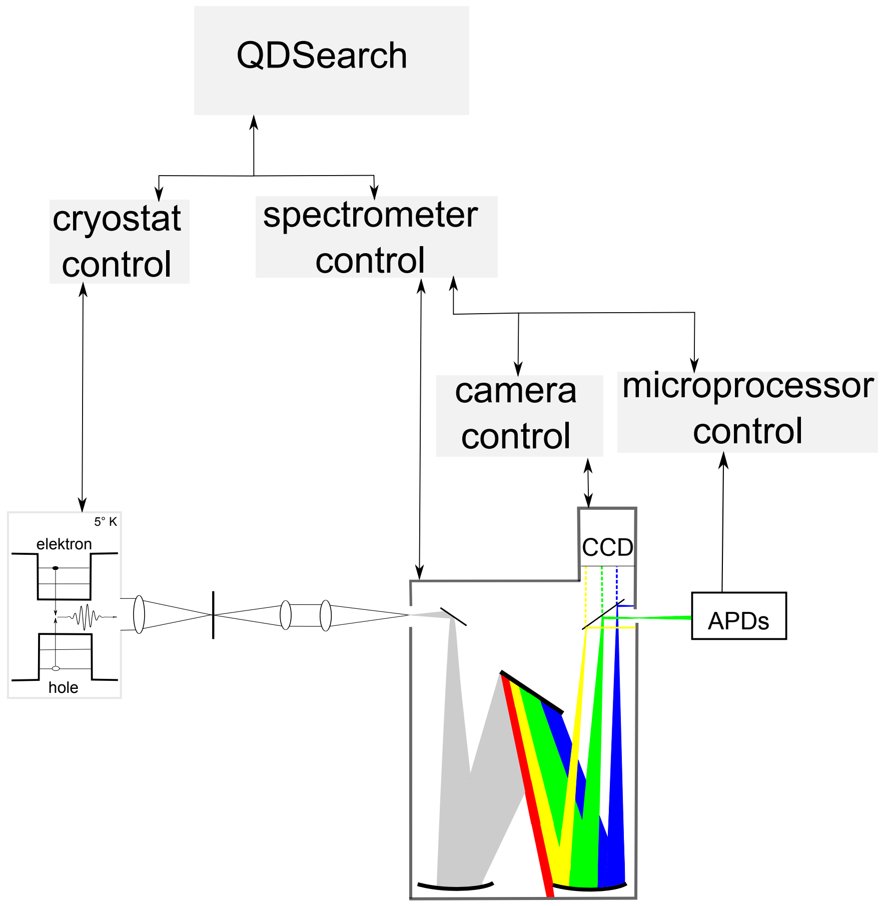
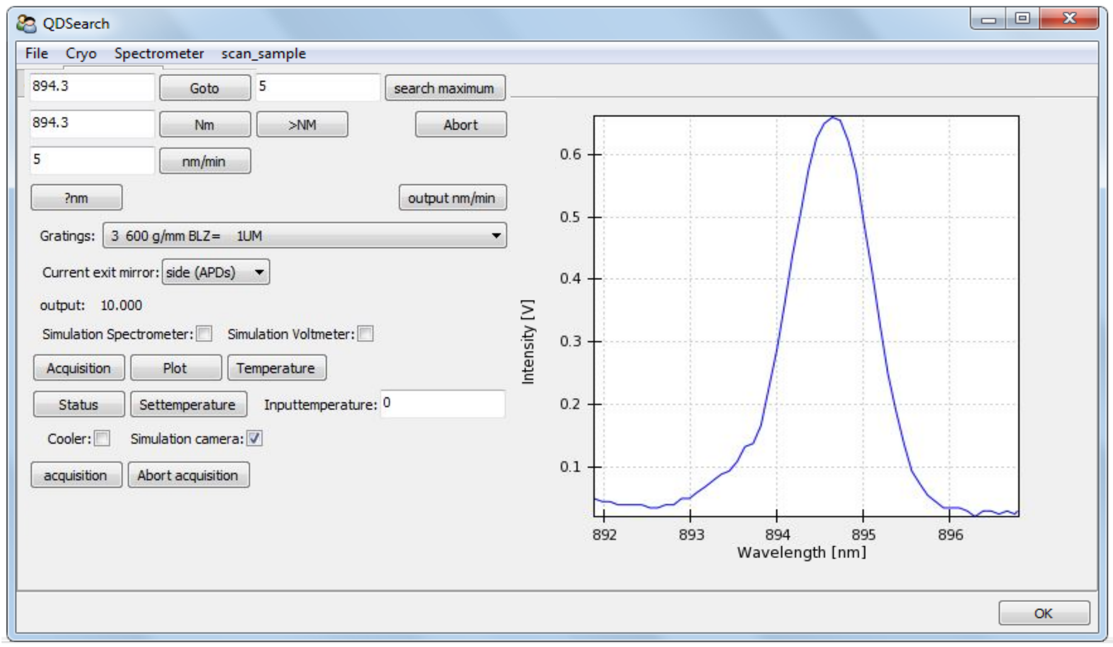


Figure 11: Program structure of **QDSearch**. It handles a spectrometer, a camera, the motor of a cryostat and a microprocessor. The control of the spectrometer is divided into three parts. The first part is the spectrometer itself and the second is the camera, which is controlled with **ctypes** (see Chapter 6) The third part is the microprocessor, which reads out the intensity of one APD.

7.1 Searching the Maximum Intensity

Only light with a small bandwidth leaves the exit slit of the spectrometer. This can be used to measure the spectrum of the light. For that the APDs measure the intensity of the incoming light while the center wavelength of the spectrometer is changing. Then the measured intensity is plotted in dependence of the center wavelength. At the end of the measurement this method changes the center wavelength to the value with the highest intensity. The GUI after a measurement is shown in Figure 12.



894.3 Goto 5
894.3
5
Gratings: 3 600 g/mm BLZ= 1UM
Current exit mirror: side (APDs)
output: 10.000
Simulation Spectrometer: ☐ Simulation Voltmeter: ☐
Inputtemperature: 0
Cooler: ☐ Simulation camera: ✓

Figure 12: Spectrometer GUI. The spectrum of one position on the sample has been measured. The plot is shown on the right site of the GUI. The upper buttons on the left site control the center wavelength and the lower buttons the camera of the spectrometer.

A complex function needs a lot of time to run and the program waits for the result of that function. Meanwhile the GUI is waiting, too. The result ist that the GUI is frozen while such a function is running. For example, this happen when the user is searching the maximum intensity of one position on the sample. Python has the **thread** module to prevent such behavior because it enables parallel computing [27]. **thread** is imported via

```
import thread
```

Afterwards a new thread can be started simply by

```
thread.start_new_thread(function, args[, kwargs])}
```

Note that the function will continue running even when the GUI is closed.

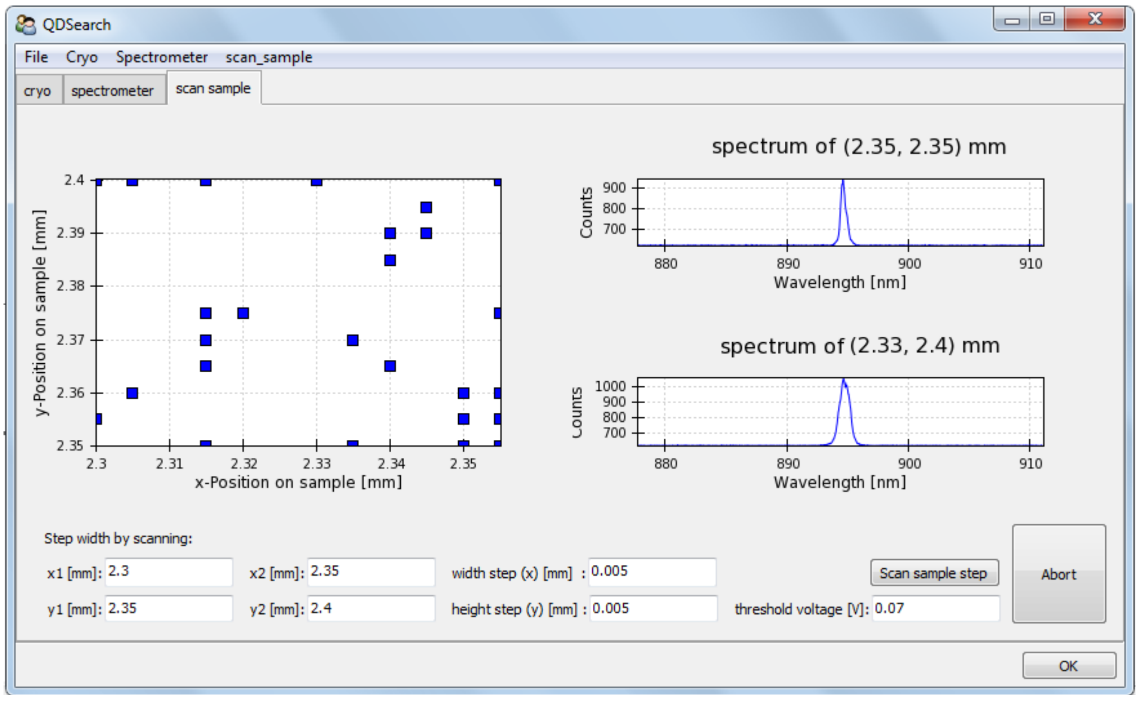
While an automatic measurement is running it is critical if the user gives another command to one device. It is probable that the measurement provides wrong data. An elegant way to prevent this behavior is to disable the involved objects. For that the keyword **enabled_when** can be used in **View**. The condition to enable is given as a string. For example

```
enabled_when='measurement_process==False'
```

Similar **visible_when** is working. The difference is that the items can be hid with that keyword.

7.2 Searching Quantum Dots

QDSearch uses the cryostat to sweep through the sample between two given points, which define the scanning area. The sample moves from the starting point downwards until it reaches the end of the scanning area. Afterwards it moves a step to the left and then upwards until it reaches the start column and so on. The movement is not continuous but step by step. The step size is adjustable in order to change the number of measurement points and the accuracy of the measurement. Depending on the step size it can happen that one QD is not measured or it is measured twice. Some samples have single self-assembled QDs. Then the space between the QDs is random and it is difficult to choose an optimal step size. For example, on other samples a shadow mask is set over the QD layer for equidistant gaps. Therefore the step size has to be adjusted to every sample again.
At the beginning of the measurement the spectrometer is used as a monochromator because usually only a narrow wavelength range is of interest. The monochromatic light of the spectrometer is directed onto the APDs. By every step the signal of the APDs is read out and compared with a given threshold voltage. If the measurement voltage is higher than the threshold voltage, the spectrometer flips a mirror (see Figure 2) and the spectrum of the point can be taken. It is important to adjust the threshold voltage. If the threshold voltage is chosen too low, the spectrometer takes too many spectra. However, if the threshold voltage is too high it can happen that no QDs are found. After the measurement of a spectrum, the mirror is flipped back and the spectrometer can be used as a monochromator again.
The measured points are represented in a map. Every blue point of the map represents a measured spectrum. This spectrum can be displayed when the mouse is moving across the correspondent point of the plot. The GUI of **QDSearch** after a measurement is shown in Figure 13.
For the documentation it is important that the measured map can be saved. Without that feature the sample has to be scanned every time. The saving is realized with Pythons **pickle**[28].


QDSearch
File Cryo Spectrometer scan_sample
cryo spectrometer scan sample
spectrum of (2.35, 2.35) mm
spectrum of (2.33, 2.4) mm
y-Position on sample [mm]
x-Position on sample [mm]
Counts
Wavelength [nm]
Scan sample step
Abort
OK

Step width by scanning:

x1 [mm]: 2.3 x2 [mm]: 2.35 width step (x) [mm] : 0.005

y1 [mm]: 2.35 y2 [mm]: 2.4 height step (y) [mm] : 0.005 threshold voltage [V]: 0.07

Figure 13: The **QDSearch** GUI. On the left hand side the map of the sample is shown. The blue squares correspond to a measured spectrum. On the right side two plots can be seen. The upper plot shows a spectrum of a measured point. It is changed automatically when the mouse is moved across a measured point. To compare spectra of different points, any of the measured points can be selected with a right click. Its spectrum is displayed below the first spectrum. With the buttons and the text fields the searching process can be controlled. Note that not every blue point on the map has to be a single QD. A spectrum is taken, whenever the signal is above the given threshold. Any bright fluorescence source on the sample might cause this. In this measurement a narrow bandpass filter (center wavelength 895 nm, width 1 nm) was used to reduce the low light in the spectra.

The cryostat has two different types of movements [29]. The first is an absolute movement to a given coordinate. The other is a relative move from the current position. Using the relative movement can result in an absolute positioning error. Therefore it cannot be guaranteed that the steps are equidistant. For that reason **QDSearch** always calculates the next step and uses the absolute move.

8 Outlook

A long-term measurement brings the problem that the setup is changing slightly. This might be due to thermal fluctuation or other external perturbance. It is possible to counter act some of these changes actively. This active stabilization of the QDs is not part of this work. However, some ideas for future work will be discussed in this section.

To stabilize the QDs, the program could evaluate the change of intensity of the QDs. If the intensity declines, there are three possibilities to maximize the intensity again, back to its original value. First, the objective in front of the lens system (see Figure 2) can be adjusted with a piezo. The voltage can be set with the digital analog converter of the microprocessor. Second, the sample in the cryostat can be moved. In the process the sample should be moved outgoing of a fixed point in every direction and measures the intensity. Then it goes in that direction with the highest intensity. This is repeated until the intensity of the current point is higher than in each direction. With that point the maximum is found. Third, the center wavelength of the spectrometer can be readjusted. For this purpose, an algorithm was already created to find the wavelength with the highest intensity in a defined range. By iterating these three possibilities the previous intensity level can be recovered.

When moving the sample or the piezo, problems caused by local maxima can occur. When a local maximum is found the searching process could stop without finding the highest intensity. To prevent this behavior the searching algorithm could be executed a few times with different parameters, like the step range or the start point. Then the different maxima found are compared and the one with the highest intensity is presumed to be the global maximum of this QD. It has to be investigated whether this method is feasible. Some algorithms exist to find the global maximum with existing local maxima. For example, the hill climbing algorithm [30]. With such a stabilization a long term measurement becomes more practical and the QDs can be investigated more efficiently.

9 Summary

In this work the automated control of a spectrometer, a cryostat and a microprocessor via pySerial were demonstrated. In this process a simulation program called **SimSerial** for devices with serial port was written. Furthermore a camera was controlled via an existing DLL file. For better handling of the program, a GUI was created with the **traits** packet. At the same time a lot of functions of the GUI were adjusted to the behavior of the devices. An algorithm that searches for the wavelength with the maximum intensity, was created by using the spectrometer and the APDs. Finally, an algorithm for searching a sample of QDs was developed and implemented. With that the sample can be searched effectively. Summarizing the existing setup was successfully automated.

References

[1] Spiller, Timothy P.: Basic Elements of Quatum Information Technology; Lo, Hoi-Kwong (editor); Popescu, Sandu (editor) ; Spiller, Tim (editor). In: *introduction to quantum computation and information*, World Scientific, 2000

[2] Bennett, Charles; Brassard, Gilles: Quantum Cryptography: Public key distribution and coin tossing. In: *Proceedings of the IEEE International Conference on Computers, Systems, and Signal Processing, Bangalore*, 1984

[3] Sangouard, Nicolas; Simon, Christoph; Riedmatten, Hugues de ; Gisin, Nicolas: *Quantum repeaters based on atomic ensembles and linear optics*. 2009. – ar-Xiv:0906.2699v2, 2009

[4] Briegel, H.-J.; Dür, W.; Cirac, J. I. ; Zoller, P.: Quantum Repeaters: The Role of Imperfect Local Operations in Quantum Communication. In: *Physical Review Letters* 81 (1998), p. 5932–5935

[5] Steinlechner, Fabian; Trojek, Pavel; Jofre, Marc; Weier, Henning; Perez, Daniel; Jennewein, Thomas; Ursin, Rupert; Rarity, John; Mitchell, Morgan W.; Torres, Juan P.; Weinfurter, Harald ; Pruneril, Valerio: A high-brightness source of polarization-entangled photons optimized for applications in free space. In: *Optics Express* 20 (2012), p. 9640–9649

[6] Trotta, R.; E.Zallo; Ortix, C.; P.Atkinson; Plumhof, J.D.; J., van den B.; Rastelli, A. ; Schmidth, O.G.: Universal Recovery of the Energy-Level Degeneracy of Bright Excitons in InGaAs Quantum Dots without a Structure Symmetry. In: *Physical Review Letters* 109 (2012), p. 147401

[7] Kroh, Tim: *Charakterisierung von Quantenpunkt-Einzelphotonen für Quantenrepeater-Anwendungen*, Humboldt University of Berlin, masters thesis, 2012

[8] Lutz, Mark; Steele, Julie (editor): *Learning Python*. 4th. O'Reilly Media Inc., 2009. – pages: 3 - 18, 585 - 592, 611 - 623

[9] Ernesti, Johannes; Kaiser, Peter; Stevens-Lemoine, Judith (editor); Scheibe, Anne (editor): *Python Das umfassende Handbuch*. 2. Galileo Press, 2009. – `http://openbook.galileocomputing.de/python/` last checked: 2013.09.25

[10] Rossum, Guido van; Drake, Fed L. (editor): *The Python Standard Library*. Python Software Foundation, 2013. – `http://docs.python.org/2/library/` last checked: 2013.09.25

[11] Rossum, Guido van; Drake, Fed L. (editor): *The Python Tutorial*. Python Software Foundation, 2013. – `http://docs.python.org/2/tutorial/` last checked: 2013.09.25

[12] Enthought, Inc.: *Enthought Tool Suite.* – `http://code.enthought.com/projects/index.php` last checked: 2013.09.25

[13] Pierce, Lyn; Swisher, Janet: The View and Its Building Blocks. In: *Traits UI User Guide*, Enthought, Inc., 2010. – `http://code.enthought.com/projects/traits/docs/html/TUIUG/view.html` last checked: 2013.09.25

[14] Pierce, Lyn; Swisher, Janet: Advanced View Concepts. In: *Traits UI User Guide*, Enthought, Inc., 2010. – `http://code.enthought.com/projects/traits/docs/html/TUIUG/advanced_view.html` last checked: 2013.09.25

[15] Enthought, Inc.: *Interactive plotting with Chaco.* 2012. – `http://docs.enthought.com/chaco/user_manual/chaco_tutorial.html` last checked: 2013.09.25

[16] Pierce, Lyn; Swisher, Janet: Customizing a View. In: *Traits UI User Guide*, Enthought, Inc., 2010. – `http://code.enthought.com/projects/traits/docs/html/TUIUG/custom_view.html` last checked: 2013.09.25

[17] *User's Guide To: Andor Technology SDK.* Andor Technology plc., 2007

[18] Pierce, Lyn; Swisher, Janet: Controlling the Interface: the Handler. In: *Traits UI User Guide*, Enthought, Inc., 2010. – `http://code.enthought.com/projects/traits/docs/html/TUIUG/handler.html` last checked: 2013.09.25

[19] Liechti, Chris: Welcome to pySerials documentation. In: *pySerial v2.6 documentation*, 2010. – `http://pyserial.sourceforge.net/pyserial.html` last checked: 2013.09.25

[20] Pedroni, Samuele; Rappin, Noel; Lewin, Laura (editor): *Jython Essentials.* O'Reilly & Associates Inc., 2002

[21] Mueller, John P.; Reese, Paul (editor); Bridges, William (editor); Mullen, Russ (editor) ; Rapoport, Nancy (editor): *Professional IronPython.* Wiley Publishing Inc., 2010

[22] Liechti, Chris: Short introduction. In: *pySerial v2.6 documentation*, 2010. – `http://pyserial.sourceforge.net/shortintro.html` last checked: 2013.09.25

[23] Liechti, Chris: pySerial API. In: *pySerial v2.6 documentation*, 2010. – `http://pyserial.sourceforge.net/pyserial_api.html` last checked: 2013.09.25

[24] Rossum, Guido van: Built-in Functions; Drake, Fed L. (editor). In: *The Python Standard Library*, Python Software Foundation, 2013. – `http://docs.python.org/2/library/functions.html` last checked: 2013.09.25

[25] Rossum, Guido van: Using Python on Windows; Drake, Fed L. (editor). In: *Python Setup and Usage.* Python Software Foundation, 2013. – `http://docs.python.org/2/using/windows.html` last checked: 2013.09.25

[26] Rossum, Guido van: ctypes A foreign function library for Python; Drake, Fed L. (editor). In: *The Python Standard Library*, Python Software Foundation, 2013. – `http://docs.python.org/2/library/ctypes.html` last checked: 2013.09.25

[27] Rossum, Guido van: thread Multiple threads of control. In: *The Python Standard Library*, Python Software Foundation, 2013. – `http://docs.python.org/2/library/thread.html` last checked: 2013.09.25

[28] Rossum, Guido van: pickle Python object serialization; Drake, Fed L. (editor). In: *The Python Standard Library*, Python Software Foundation, 2013. – `http://docs.python.org/2/library/pickle.html` last checked: 2013.09.25

[29] MICOS: *Handbuch SMC PC - Allgemeine Hinweise Positionierungssteuerung*

[30] Tovey, Craig A.: Hill Climbing with multiple local optima. In: *SIAM Journal on Algebraic and Discrete Methob* 6 (1985), p. 364–193

Danksagung

Zuerst möchte ich mich bei Herrn Prof. Dr. Oliver Benson für die Möglichkeit bedanken, eine Bachelorarbeit in der AG Nanooptik schreiben zu können.

Ich möchte mich außerdem vor allem bei Otto Dietz für seine Unterstzüng bei der Entstehung dieser Arbeit danken. Ohne seinen Einsatz wäre ein erfolgreicher Abschluss dieser Arbeit nicht möglich gewesen. Er hat mich auch dazu ermutigt die Bachelorarbeit in Englisch zu verfassen und mich dann bei diesem Vorhaben unterstüzt.

Weiterer Dank gilt Tim Kroh, der mir bei Fragen zur Verfügung stand und sich immer Zeit nahm diese zu klären. Zusätzlich möchte ich auch Thomas Kreißl danken, mit dem ich einige Zeit im Labor verbracht habe und der das Programm immer wieder getestet hat. Weiterhin danke ich allen anderen MitarbeiternInnen der AG Nanooptik und der AG Optische Metrologie für die sehr angenehme Arbeitsatmosphäre.

Von meinen FreundenInnen möchte ich insbesondere Kirsti danken. Sie hat den Großteil der Arbeit korrigiert und mir die englische Sprache näher gebracht. Des Weiteren bedanke ich mich bei Melanie und Julia, die mich ebenfalls bei der Korrektur unterstützt haben.

Schließlich danke ich meinen Eltern, die mir das Studium erst ermöglicht haben und mich auch auf meinem weiteren Lebensweg weiter unterstützen werden. Bei meinem Bruder bedanke ich mich dafür, dass er mich auf seine Art immer wieder neu ermutigt und mir mit Rat zur Seite steht.

Zuletzt bedanke ich mich bei meinen Freunden, die mir während des Studiums regelmäßig zeigten, dass es noch etwas anderes im Leben gibt.

Eigenständigkeitserkärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ort, Datum	Unterschrift