Automatic Computer Control of the Paul Trap Experimental Setup

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

Jonas Engelmann
geboren am 04.03.1991 in Bergisch Gladbach

Betreuung:

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

eingereicht am: 18. April 2016

Abstract

The aim of this work is the facilitation of experiments conducted on trapped particles in a Paul trap. In this context, a graphical user interface control application for a Paul trap experimental setup has been developed and tested. The control of various devices have been integrated and bundled into one program with the aim of providing the basis for the deployment of feedback systems. In that regard, algorithms and procedures to stabilize the focus and the position of the image of trapped particles have been implemented and successfully tested, therefore guaranteeing that the image of a trapped particle will stay in focus and in the field of view over a long period of time; both are essential preconditions for long-term measurements.

Zusammenfassung

Ziel dieser Arbeit ist die Erleichterung der Durchführung von Experimenten an gefangen Partikeln in einer Paulfalle. In diesem Zusammenhang wurde ein Steuerungsprogramm mit graphischer Benutzeroberfläche für den experimentellen Aufbau einer Paulfalle entwickelt und getestet. Dabei wurde die Ansteuerung verschiedener, im Experimentaufbau beteiligter Geräte integriert und in einem Programm zusammengefasst mit dem Ziel, die Grundlage für die Einsetzung von Feedback-Systemen zu legen. Algorithmen zur Stabilisierung des Fokusses und der Position des Bildes von gefangen Partikeln wurden implementiert und erfolgreich getestet. So kann sichergestellt werden, dass das Bild eines gefangen Partikels über einen längeren Zeitraum fokussiert und im Sichtfeld bleibt; was wesentliche Voraussetzungen für Langzeitmessungen sind.

Contents

List of Abbreviations

CCD	Charge-coupled device camera
DLL	Dynamic link library
DPSS	Diode-pumped solid-state laser
GUI	Graphical user interface
OOP	Object-oriented programming
OpenCV	Open source computer vision library
NV	Nitrogen-vacancy centre

List of Figures

1. Introduction

The Paul trap is an experimental arrangement that has been originally proposed by Wolfgang Paul et al. in 1958 [1]. Wolfgang Paul together with Hans G. Dehmelt shared the Nobel Prize for Physics for the development of electromagnetic ion traps in 1989 [2].

The Paul trap is a quadruple ion trap that makes the confinement of charged particles possible. It acts as an ion store in which charged particles can be trapped for a period of time. With the use of an oscillating quadrupole field a single particle is suspended in free space and is therefore not exposed to interactions with other particles or surfaces. The trapped ion approximately behaves as a free particle over a long period of time, since its interaction with the trap potential is far smaller than with other particles when bound to them. Under these conditions the isolated particle can be easily observed, measured and quickly characterized.

The scope of applications of the Paul trap is wide and versatile. Historically, it evolved as a component of a mass spectrometer and is still used as such [3]. However, it also proved itself beneficial in fluorescence spectroscopy [4], measurement of optical transitions with high accuracy [5] and quantum information processing [6, 7, 8]; paving the way for the trapped ion quantum computer. A quantum computer can be realized with cooled ions confined in an ion trap and interacting with laser beams.

In this project, Paul trap experiments are mainly used to characterize nitrogen-vacancy (NV) defect fluorescence of levitating nano diamond crystals [9]. Using an ion trap for this task is beneficial since optical tweezers, another option, quench the fluorescence emission. The diamonds used in these experiments have an average size of 125 nm, however only clusters of diamonds are trapped which typically have sizes between 500 nm and 5 µm [9].

The nitrogen-vacancy defect is an impurity in the crystal structure of diamonds. One carbon atom is replaced by a nitrogen atom resulting in a defect adjacent to it. The characterization of nitrogen-vacancy centers in diamonds is an interesting field of research since they have a wide scope of possible future applications. NVs can be used as photostable single photon sources operating at room temperature [10]. They are also considered as a promising candidate for the so called qubit [11]; the quantum bit which is a superposition of the classical bits, "1" and "0". Therefore NV centers possibly play an important role in the realization of a quantum computer and its integration into integrated circuitry. Furthermore, NV centers embedded in nano diamonds can be used for nanoscale magnetic field sensing [12] or as fluorescent makers in biology [13].

Focus of this Project

My focus in this project is the implementation of a computer application that controls all necessary devices in the Paul trap experimental setup.

In some cases control applications are already provided by the manufactures, but this project has the aim to bundle their control into one application, making communication among the individual devices possible. In that way, a feedback system can be set up that uses all the information available and automatically adjusts itself accordingly.

Due to the geometry of the trap the optical access is very limited and therefore the emission collection of the spectrometer poor. One can counter this by extending the period of time over which the measurement takes place, but due to imperfections and asymmetries in the geometry of the Paul trap and unwanted stray fields, particles can move out of focus and out of the field of vision of the spectrometer. With such a feedback system, stabilization in all three dimensions can be achieved and therefore the quality of spectrum analysis improved.

In chapter two the fundamentals of the theory of the linear Paul trap will be explained. Additionally, the experimental setup will be explained and depicted in detail. In chapter three an overview over the programming structure is given, while also the programming languages, tools and methods used to realize the Paul trap control application will be introduced and briefly explained. For an optimal understanding basic programming skills in Python are expected. In chapter four, two ways are explained by which communication with the various devices have been established; one using an existing dynamic link library and another being communication via the serial port. In chapter five, algorithms and procedures are discussed that help to improve the quality of spectrum analysis, namely a focus and center stabilization.

2. Linear Paul Trap

In this chapter the fundamentals of the theory of the linear Paul trap are discussed (section 2.1). It is sufficient to gain a basic understanding. Furthermore, the experimental setup of the Paul trap will be explained (section 2.2).

2.1. Theory

How does the confinement of a particle in a Paul trap work? According to Earnshaw's theorem, ions cannot be confined in three dimensions by just using electrostatic forces. Therefore the quadrupole electric field has to oscillate, consequently forming a potential with the shape of a saddle, switching its sign at a high frequency. Over a period of time corresponding to the driving frequency the Paul trap, an ion positioned at the saddle point experiences strong restoring forces, which are created by the potential time-averaged over one period, as it deviates from the saddle point, and is therefore trapped.

The potential of the ion trap can be expressed as:

$$\Phi(x,y,z,t) = \frac{\Phi_0(t)}{2r_0^2}(ax^2 + by^2 + cz^2) \tag{2.1}$$

where r_0 denotes the distance from the trap center to the closes electrodes surface. Any electric field has to fulfill the Laplace condition $\Delta\Phi = 0$, which leads to the condition: $a + b + c = 0$. The easiest two solutions are:

$$i)\ a = 0,\ b = -c = 1 \qquad \text{and} \qquad ii)\ a = b = 1,\ c = -2.$$

Option $i)$ results in a confinement in two dimensions and is referred to as the linear Paul trap whereas option $ii)$ confines in three dimensions, corresponding to the classical Paul trap. In this project a linear Paul trap is exclusively used, therefore I will continue my calculation with option $i)$. Option $i)$ can be achieved using four electrodes extended in the longitudinal direction.

The resulting potential looks as follows:

$$\Phi(y,z,t) = \frac{\Phi_0(t)}{2r_0^2}\left(y^2 - z^2\right) \tag{2.2}$$

with

$$\Phi_0 = U_{DC} + U_{AC}\cos(\Omega t) \tag{2.3}$$

where Ω is the frequency with which the saddle point switches. Using appropriate values for the magnitude U_{AC}, offset U_{DC} and the frequency Ω of the potential, particles with a certain mass-charge ratio can be trapped, that is, their trajectories are stable.

To find these trajectories one can set up the equations of motion using $F = -q\nabla\Phi$:

$$\ddot{y} + \frac{q}{mr_0^2}\left(U_{DC} + U_{AC}\cos(\Omega t)\right) y = 0 \tag{2.4}$$

$$\ddot{z} + \frac{q}{mr_0^2}\left(U_{DC} + U_{AC}\cos(\Omega t)\right) z = 0. \tag{2.5}$$

Through corresponding substitution:

$$\tau = \frac{\Omega t}{2} \tag{2.6}$$

$$a_y = -a_z = \frac{4qU_{DC}}{mr_0^2\Omega^2} \tag{2.7}$$

$$q_y = -q_z = \frac{2qU_{AC}}{mr_0^2\Omega^2} \tag{2.8}$$

one arrives at the canonical form for Mathieu's differential equation:

$$\frac{d^2y}{d\tau^2} + \left(a_y - 2q_y\cos(2\tau)\right) = 0 \tag{2.9}$$

$$\frac{d^2z}{d\tau^2} + \left(a_z - 2q_z\cos(2\tau)\right) = 0. \tag{2.10}$$

An analysis of these differential equations is for example given in [14, 15]. Whether a solution of the trajectory is stable and consequently can be confined within the trap only depends on the parameters u_y and q_y, respectively u_z and q_z. Since in this experiment U_{DC} is equal to zero and thus $u_y = u_z = 0$, stable trajectories can be found for $0 < q_y, q_z < 0.908$ [15], which is equivalent to:

$$\left|\frac{q}{m}\right| < \frac{0.908}{2}\frac{r_0^2\Omega^2}{U_{AC}}. \tag{2.11}$$

An approximate solution can be found by separating the motion into two components:

$$y(t) = \bar{y}(t) + \delta(t). \tag{2.12}$$

where $\bar{y}(t)$ denotes the so-called secular motion with high amplitude and small frequency and $\delta(t)$ the micro motion with small amplitude and high frequency.

The solution is given for example in [15]:

$$y(t) = \bar{y}_0 \sin(\overline{\omega}_y t)\left(1 - \frac{q_y}{2}\cos(\Omega t)\right) \tag{2.13}$$

$$\text{with } \overline{\omega}_y = \frac{q_y}{\sqrt{2}}\frac{\Omega}{2} \tag{2.14}$$

where τ has been resubstituted according to equation 2.6. The solution for the motion in the z-direction can be derived analogously.

2.2. Experiment

The following describes the experimental apparatus and highlights the elements that will be implemented in the control program:

The Paul trap used in this project is a linear segmented ion trap. It is composed of four electrodes arranged/assembled longitudinally at the corners of a square. The whole trap is mounted inside an air tight chamber to protect trapped particles from unwanted air flow.

Confinement in two dimension (y and z) is realized with an oscillating quadrupole potential (see equation 2.2). It is created with a sinusoidal voltage with high angular frequency (typically $\Omega = 2\pi\cdot(2\text{–}100)\,\text{kHz}$) and high voltage amplitudes (typically $U_{AC} = (1-2)\,\text{kVpp}$) between diagonally opposing electrodes. The trap is run without any offset voltage U_{DC} to widen the range in which the stability condition is fulfilled (for a graphical representation of the stability regions see for example: [16] and [2]).

Confinement in the longitudinal trap direction is achieved through segmentation of two diagonally opposing electrodes, as has been used in [17]. To every segment a static voltage of few ten volts can be applied individually. A particle is trapped in the longitudinal direction in a certain segment if a static voltage is applied to its adjacent segments (see figure 2.3). Through appropriate application of voltages to the segments one can move the particle around and store them at the end of the trap or separately isolate them in front of a microscope and thus examine them one after another. The voltages applied to the segments can be turned on and off with relays which are controlled by a micro controller with is connected to a computer.

Additionally, the potential of segment 6 and 8, the ones adjacent to the center segment, can be finely regulated and consequently the position along the x-axis changed on a fine scale. This is necessary to accurately navigate a trapped particle to the front of the objective. The alteration of the potentials of these segments is realized by attenuating the related voltages using a resistive opto-isolator. An opto-isolater consists of an LED and a light-dependent resistor both shielded in an opaque box. In that way, the resistance of the light-dependent resistor can be controlled with the intensity of the light of the LED which can in turn be controlled with the micro controller. Therefore the regulation of the potential can be computer-controlled.

The whole Paul trap is mounted on a linear motor stage that can be operated along two axes: along the z-axis to move the trapped particle in front of the objective, and along the y-axis, allowing alignment along the optical axis and hence navigating it into the focal point of the objective. The controller of the linear motor stage is connected to the computer.

For optical characterization, the trapped particles are excited by a diode-pumped solid-state (DPSS) laser at 532 nm with typical powers between 1 and 30 mW. The DPSS laser is aligned with the longitudinal axis of the trap (x-direction in figure 2.1). The microscopic objective (Olympus, 50x, $N_A = 0.5$) is aimed at segment 7, collecting scattering and fluorescence light from particles positioned in its field of view. The scattered and emitted light is directed onto a charge-coupled device (CCD) camera. For spectral analysis the detected beam of the emitted and scattered light goes through a long pass filter, which passes wavelengths higher than 540 nm and thus suppresses the excitation light. Then the fluorescence can be detected with a spectrometer. A schematic diagram of the optical setup can be found in figure 2.1.

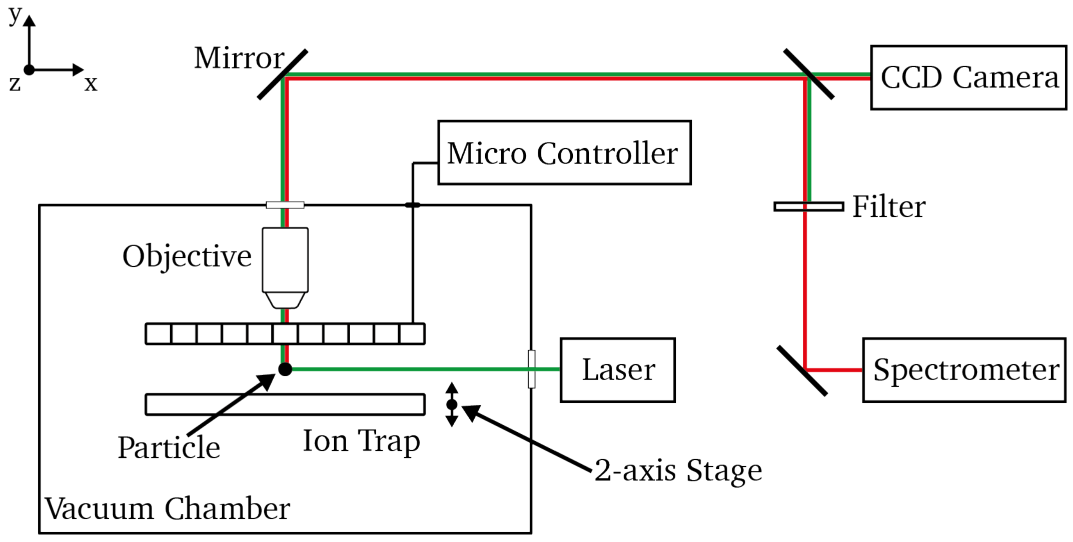


Figure 2.1.: Schematic diagram of the optical setup

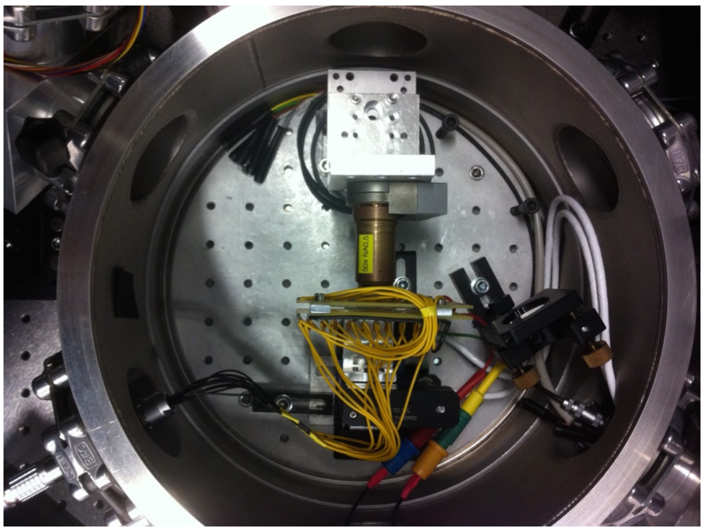


Figure 2.2.: Photo of the experimental setup of the Paul trap

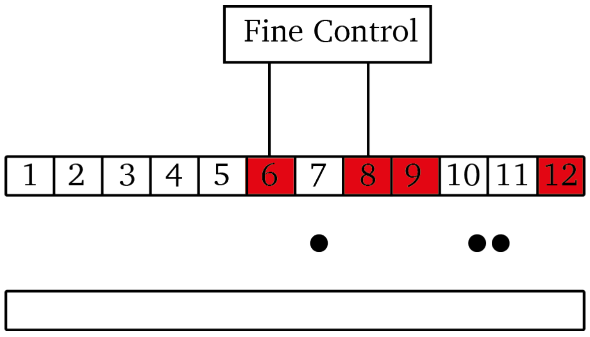


Figure 2.3.: Depiction of the segmentation of the electrodes. Red marked segments are applied with voltage.

3. Programming Structure

In this chapter the main tools and methods will be introduced with which the control application of the Paul trap was realized. It will be briefly motivated why Python has been chosen as the core programming language, its main advantages pointed out and explained how they can be made of use in more detail. Then, Enaml, a framework language that was used to build the graphical user interface (GUI), will be introduced; to provide some ideas of it, some basic examples will be explained.

The main concern in deciding on the tools was to design the software's code to be as understandable and readable as possible. The programming structure should be clear enough that other experimenters can improve and extend the application's functionality.

3.1. Python

Python [18] is a widely used and easy to learn programming language. Its design and syntax emphasize readability, clear code structuring and fewer lines of code, which overall make it easy to grasp and maintain its content. Additionally, Python comes with a wide set of modules as part of the standard library, which can even be extended with many freely available third party modules, thus making Python a very versatile and widely applicable programming language.

To structure the programming code in Python, that is, group certain definitions (functions and variables), different abstraction layers can be used; for example, so-called classes and modules. In that manner, one can encapsulate the code into different groups which provide a better overview and make it easy to trace back errors.

In the following sections, I will explain how this code structuring concept is realized by introducing the programming paradigm object-oriented programming (section 3.1.1) and Python's module concept (section 3.1.2).

3.1.1. Object-oriented Programming

The main idea behind object-oriented programming (OOP) is to package certain data and functions or procedures into a package and encapsulate them. In that way, the data and the functions only reside within that package, and thus are isolated from the rest of the programming code. The data and the functions share a logical connection, that is, the data can only be manipulated by functions residing within the same package. Consequently, code can be modularized and interference of different data and functions can be avoided. The package that bundles the data and functions is called "object". The data of objects is often referred to as their attributes and the functions as their methods.

Classes

To create an object one has to define a class; in some sense it can be understood as an instruction manual for creating objects. The class tells us what attributes and what methods that object will have once it is created; however, the class needs to be instantiated first.

To illustrate that concept, here is an example of a class definition:

```
class My_class():
        my_attribute=1
        def my_method(self):
                print(self.my_attribute)
```

The parameter `class` is used to define a class. The class's name starts standardly with a capitalized letter. It has one attribute called `my_attribute` and one method called `my_method` which prints out the class's attribute. The first argument, standardly named `self`, is a reference to the instance from which it is called.

To create an object from this class definition, we have to instantiate it, thereby we can choose a name to our liking for the object. Sometimes an object is referred to as an instance of a class, but they are in fact the same.

```
instance_name=My_class() # the class is instantiated
```

In this example an object named `instance_name` is created by initiating the class `My_class`. Now we can access the object's attribute through the so-called name space of the newly allocated name, that is, the object's name precedes the name of the attribute and is only separated with a dot. In that regard, the programmer knows where the attribute is defined and can trace it back:

```
>>print(instance_name.my_attribute)
1
```

To invoke a method of that object, we use the same name space: `instance_name.method(argument)`. So in our case:

```
>>instance_name.my_method()
1
```

Inheritance

The concept of inheritance is a powerful tool of object-oriented programming. It is the process by which a so-called child class takes on all attributes and methods of another so-called parent class. That means all attributes and methods from which the child class inherits are available in its own class environment. In that way, code can be reused without copying it, and thus readability greatly enhanced.

To inherit the attributes and methods of a class, the class's name has to be written in the parentheses of the class definition. A class can inherit from more than one parent class.

```
class My_class(Parentclass):
```

By invoking a method with `instance.method(argument)` Python will not only look in the My_class for the stated method but in all its parent classes. No inheritance can be achieved with the parent class's name set to `object`.

3.1.2. Modules

Modules are bundles of functions and attributes that are loaded at the beginning of a script. For example, the built-in module named `math` [19] contains, among many others, the function `sqrt` and an attribute called `pi`.

```
import math
math.sqrt(2)
math.pi
```

In this example the module `math` is loaded and the function `sqrt` is called with "2" passed as an input argument; furthermore, the attribute `pi` is called. As one can see, the function defined in the module (here `sqrt`) is called via the module's name space. If a lot of functions are to be defined, it might be convenient to save them in a separate file and load this file as a module.

Python already comes with an extensive number of modules that cover a wide range of application fields. It is thought of as one of Python's greatest advantages. With the use of modules, the programmer can load the needed tools to solve a problem or handle a task. In that manner, one can rely on standardized solutions to specific problem, such as the calculation of a square root.

Besides Python's build-in modules, there are many third-party modules that greatly extend Python's application. For example, there are modules for scientific computing, computer vision or that facilitate the programming of graphical user interfaces.

3.2. Enaml and Atom

Enaml [20] is an easy yet powerful tool to build professional graphical user interfaces (GUI) with Python. It is a framework language that closely follows Python's syntax. Through the use of a declarative representation and indentation, the structure of the code directly reflects the arrangement of the elements of the GUI and is therefore easier to read and to understand. Enaml supports a strict model-view separation, that is, the model, the core of the application, deals with all the data and functions and is strictly separated from the view, the representation of that model. If data in the model changes, the view is immediately updated.

With the use of such a model-view separation, data synchronization problems can be avoided. To illustrate the importance of data synchronization, here is an example: A toggle button is used to turn a relay on and off. At the same time, procedures running in the background have access to the control of that relay and may change its status. If that happens, the status of the toggle button on the GUI (whether it is toggled or not) stops accurately reflecting the status of the relay. Employing a view-data separation with synchronization corrects this behavior. In that case, the status of the toggle button always reflects the status of the relay defined in the model and is constantly checked and updated. Consequently, this synchronization can be used to reliably read out the hardware's status on the basis of the components of the GUI.

In this project Enaml has been chosen for the development of the graphical user interface since it is easy to understand and emphasizes readability while being able to implement complex GUI behavior, such as data synchronization. The data synchronization is handled by a Python module called Atom; it is supported by Enaml.

An Enaml GUI application consists of two files:

1. The view file ending with `.enaml`: It defines the view of the application, comprising all components of the GUI and layout constraints. It is written in Enaml's own framework language.
2. The model file ending with `.py`: It is written in Python and contains all the data, models, algorithms and procedures. The corresponding view file must be linked. Furthermore, the start up instructions of the application are defined in the Python file, that is, instructions to show a window on the screen and eventually start up the application event loop.

In the following sections the main widgets, that is, the components of a GUI through which a user interacts, will be introduced with illustrative examples. Also, the use of Atom objects and thus data synchronization will be explained.

3.2.1. Toggle Button

In this example a toggle button is created. Unlike a button that simply calls a function when it is pressed, a toggle button allows the user to switch between two states, i. e. it can be used to call two different functions depending on its state, which can be of great use for switching relays on and off.

The corresponding Python file of the implementation of a toggle button looks as follows:

Python File 3.1: toggle_button.py

```
import enaml
from atom.api import Atom, Bool
from enaml.qt.qt_application import QtApplication

class Toggle_Class(Atom):
    toggled=Bool(False)

    def function_1(self,status):
        if status==True:
            print("Button toggled")
            # do something
        else:
            print("Button untoggled")
            # do something else

if __name__ == '__main__':
    with enaml.imports():
        from toggle_button import Main

    toggle_model=Toggle_Class()

    app = QtApplication()
    view = Main(toggle_view=toggle_model)
    view.show()
    app.start()
```

After all the necessary modules have been loaded, a class is defined, inheriting from Atom. The attribute `toggled` is specifically assigned to an Atom object, which is necessary to provide synchronization.

With the function `enaml.imports()` a view file is imported into the Python environment. After instantiating the model (in this case `toggle_button`), the toolkit-specific application object is loaded. While instantiating the view file, the model is linked to it:

```
view=Main(toggle_view=toggle_model)
```

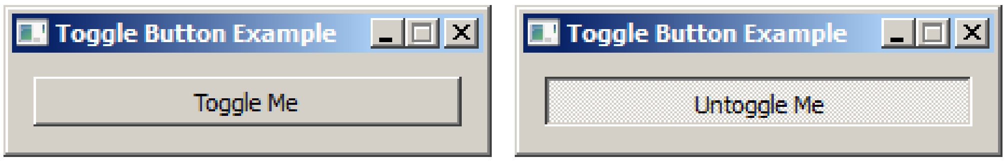
It is important to note that the variable on the left side has to correspond to an attribute set in the view file and the variable on the right side with the instantiated model. Finally, the view object is displayed and the application event loop started.

Enaml File 3.2: toggle_button.enaml

```
from enaml.widgets.api import Window, Container, PushButton

enamldef Main(Window):
    title = "Toggle Button Example"
    attr toggle_view
    Container:
        PushButton:
            checkable = True
            checked := toggle_view.toggled
            toggled :: toggle_view.function_1(toggle_view.toggled)
            text << ('Toggle Me' if toggle_view.toggled is True
                                else 'Untoggle Me')
```

In the first line of the view file, all the necessary widgets are imported analogously to Python's import function. The keyword `enamldef` is used to define an Enaml view called `Main`, which derives from the widget `Window`, creating a window that holds a container which in turn holds a push button. With the help of `container`, other widgets can be arranged and organized within a window. Enaml code follows the indentation concept of Python, where code with a certain indentation level refers to the code line at the next higher indentation level. In this case, the options and variables `checkable`, `checked`, `toggled` and `text` refer to the push button widget. Checkable defines whether the push button is a simple button or a toggle button, and consequently `checked` its status, where the Enaml operator `:=` denotes a bidirectional synchronization between a GUI-element on the left side and a variable on the right side, that is, whenever the toggle button is clicked, the variable on the right side is updated and reversely when the variable on the right side is changed, the GUI-element is updated[1]. The operator `::` is used to execute a piece of code when the GUI-element on the left side changes. To use the Python programming language environment while defining a GUI-element, one can use the operator «.



(a) (b)

Figure 3.1.: A GUI with a button toggled in (a) and untoggled in (b)

3.2.2. Slider and Field

In this example a slider, a field and a button are created. A field is a box displaying an assigned string, float or integer. Here, it will display the slider's current value, which can also be changed, apart from moving the slider button, by directly typing in a number

1 ↑This only works if the variable is an attribute of an Atom object

into the field box. This bidirectional behavior is realized using the Enaml operator := and assigning it to an Atom object, in this case Range. Once a symbol is entered in the field widget, Atom will validate the input, indicating an incorrect symbol by highlighting the field box in a red color. Thereby, it not only checks whether the input is an integer but also whether it is within the determined range.

Python File 3.3: slider_example.py

```
import enaml
from atom.api import Atom, Range
from enaml.qt.qt_application import QtApplication

class Slider_model(Atom):
    slider_value = Range(low=1,high=100,value=50)

    def print_function(self,value):
        print(value)

if __name__ == "__main__":
    with enaml.imports():
        from slider_example import Main
    slider_model=Slider_model()

    view = Main(slider_view=slider_model)
    app = QtApplication()
    view.show()
    app.start()
```

Enaml File 3.4: slider_example.enaml

```
from enaml.widgets.api import (
    Window, Container, Slider, PushButton
)
from enaml.stdlib.fields import IntField

enamldef Main(Window):
    title = "Slider Example"
    attr slider_view
    Container:
        IntField:
            value := slider1.value
            submit_triggers = (["auto_sync", "return_pressed"])
        Slider: slider1:
            tick_interval = 50
            minimum=1
            maximum=100
            value := slider_view.slider_value
            orientation= "horizontal"
        PushButton:
            text = "Print Current Value"
            clicked :: slider_view.print_function(slider1.value)
```

The code structure in the Python and Enaml file is analogous to 3.2.1. However, here, it is worth noting how the value of the slider, set by a user, is passed along: The name of the widget, in this case `slider1`, has an attribute called `value` in which its value is stored.

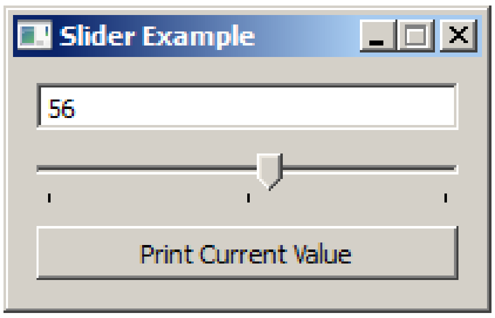


Figure 3.2.: A GUI with a slider, field and button

3.2.3. Layout Basics

In this section, basic concepts for grouping and arranging widgets are introduced. Positioning and arranging widgets such as push buttons, sliders and fields is done with the constraint option of a container. For horizontal and vertical aligning and arranging widgets in a grid view the corresponding modules need to be imported first:

```
from enaml.layout.api import hbox, vbox, grid, spacer
```

In order to refer to certain widgets they have to be assigned to an unique name first. Then, one can arrange them with the constraint option of a container as follows:

```
Container:
    constraints = [hbox(pb1,pb2), pb1.width==pb2.width]
    PushButton:pb1:
        text = "Button 1"
    PushButton:pb2:
        text = "Button 2"
```

For vertical alignment the option `vbox` is used. Arrangement on a grid is done with the grid option. If one position of the grid should be empty, it is simply assigned to `spacer`. To arrange a whole set of widgets, one can group them in different container and then in turn arrange these containers in another container, treating them as widgets and using the same method.

To visually group certain GUI components one can use the `GroupBox` widget which draws a frame around its content. A GroupBox can also be given a title.

3.3. Conclusion

In this chapter, tools and methods for developing a software interface have been motivated and briefly explained. Python has been chosen as the core programming language for its easy and understandable use, providing code structuring and encapsulation methods, such as modules or classes. Furthermore, the fundamentals for designing a basic user interface with complex data synchronization have been explained.

4. Integration of Devices

In this chapter, it will be explained how all the different devices of the Paul trap experimental setup are integrated. Usually, manufacturers provide their own control application. However, in this project the main task will be to establish communication with the devices from within the Python programming environment. In that manner, control over the whole Paul trap can be achieved with one application. Furthermore, information provided by the individual devices, such as image information or the position of the translation stages, reside in this one application. This has the advantage of easily incorporating complex automation processes.

Communication with the individual devices is achieved using two different communication protocols. First, communication using the computer's serial port is introduced (section 4.1). Second, communication via so-called dynamic link libraries (DLL), which are provided by the manufacturers, is explained (section 4.2). Usually, manufacturers provide corresponding instructions to the dynamic link libraries which facilitate the integration into the Python environment.

4.1. Communication with the Micro Controller via the Serial Port

As described in section 2.2, the micro controller in the Paul trap experimental setup controls the individual segments of the trap. Furthermore, two segments can be finely controlled by attenuating the applied voltages on a fine scale. The micro controller is connected via USB, using communication via the serial protocol.

Through the serial port, information is transferred in or out one bit at a time. To access the serial port, the third-party module pySerial [21] is used. pySerial runs on all major platforms and provides versatile and easy use over the serial port from within Python. The Python module of pySerial is called `serial`. In the following example, a port is opened to establish connection with the micro controller:

```
import serial
relay_control=serial.Serial(port="COM5", timeout=1, baudrate=9600)
```

"COM5" designates the port number, which is assigned by the operating system. The designators differ for Windows, Linux and Mac. The parameter timeout determines after how many seconds the sending or reading process terminates. This is only relevant in case serial does not find the so-called end of line character, which designates the end of a command. As soon as the end of line character is read, serial will terminate the reading process. The baud rate specifies the number of symbols transmitted per time unit. It is crucial that receiver and sender communicate with an identical baud rate.

A simple communication with the micro controller to turn on relay on looks like:

```
relay_1_on="l01s\r\n"
relay_control.reset_output_buffer()
relay_control.write(relay_1_on)
```

In the first line, the command to turn on a relay one is assigned as a string; the corresponding command is defined in the program of the micro controller. The command `write` sends the string to the micro controller. In the third line, the output buffer is flushed, discarding all its content to assure that only the defined command is sent.

The micro controller is programmed in a way that one can retrieve its status, that is, query information that tells which relay is set:

```
status="rlst\r\n"
relay_control.reset_input_buffer()
relay_control.reset_output_buffer()
relay_control.write(status)
echo=relay_control.readline()
print(echo)
```

The command `rlst` is sent and the receiving string printed. In line two, the input buffer of the micro controller is flushed to make sure that only the corresponding string is read out; `readline()` actually reads out the output of the micro controller.

It is important not to forget to close the opened port, since communication is blocked for other programs for the entire time the port is opened. One can do so with:

```
micro_control.close()
```

4.2. Communication via DLLs

A dynamic link library (DLL) is a collection of code C functions which handle specific operations. In this project, DLLs are used as device drivers. In order to load and work with DLLs in Python, the built-in module [22] is used. It provides interfacing with DLLs and C compatible data types; here is an example use of it:

```
import ctypes
libc=ctypes.WinDLL("dll_name.dll")
```

In this example, a DLL file is loaded and assigned to the variable `libc`. Now, all C functions contained by this DLL file are callable to the namespace of `libc`. To pass arguments from within Python to C functions, they have to be in C data types required by the function. To achieve this, one can wrap the arguments in their corresponding ctypes type; here is an example:

```
variable_1= ctypes.c_byte(2)
variable_2= ctypes.c_int(1)
libc.a_test_function(variable_1,variable_2)
```

In the documentation accompanying the DLL C data types are specified for every function. They can be translated using the following table:

Table 4.1.: Corresponding Python data types to C data types

C data types	Python data types
BYTE	ctypes.c_byte
WORD	ctypes.c_ushort
DWORD	ctypes.c_ulong
CHAR	ctypes.c_uchar
WCHAR	ctypes.c_wchar
UINT	ctypes.c_uint
INT	ctypes.c_int
HANDLE	ctypes.c_void_p

4.2.1. Two-axis Translation Stage

The whole Paul trap is mounted on a two-axis translation stage (PI M-112.1, PI M-110.1) to navigate trapped particles in front of the objective and along the optical axis in the focal point of it (see figure 2.1). The stages are connected to two DC-motor controllers (PI C-862.0 Mercury II) which are connected to the computer.

To establish connection with the controller, a DLL file is loaded with `ctypes`: [1]

```
import ctypes
stage_control=ctypes.WinDLL("MercLib.dll")
```

Now all the functions of the DLL are accessible through the namespace of `stage_control`. To open a port, the following instructions are given in the manual accompanying the DLL file:

```
MCRS_open(BYTE PortNumber, int baudrate)
```

When passing variables to the function `MCRS_open`, it is important to use the correct data types for `PortNumber` and `baudrate`, which are specified as BYTE and int. The following example shows how to establish connection to the stage controller using port three and a baud rate of 9600 symbols per second:

```
baudrate=ctypes.c_int(9600)
PortNumber=ctypes.c_byte(3)
stage_control.MCRS_open(PortNumber, baudrate)
```

The two stage controllers are actually connected in series and only one is connected to the computer. To be able to choose which of the stages should be moved (y- or z-axis), one has to select the corresponding controller prior to sending a move command. Before different controllers can be selected, the function `MCRS_initNetwork()` has to be called first, which scans all connected controllers and assigns logical addresses to them. In the following example the function will scan the first 5 ports for controller connections and select the first found one:

```
devices=ctypes.c_int(5)
stage_control.MCRS_initNetwork(devices)
```

1 ↑It is important to note that the specific DLL file used, only correctly interfaces with Python when it is run in 32-bit mode

```
axis=ctypes.c_int(1)
stage_control.MCRS_select(axis)
```

Now the selected stage can be moved to a specific position. A shift for 200 steps in one direction can be implemented as follows:

```
stepsize=ctypes.c_int(200)
pos=stage_control.MCRS_getPos()
pos_new=ctypes.c_int(pos+stepsize)
stage_control.MCRS_moveA(axis,pos_new)
stage_control.MCRS_waitStop() # wait until moving is done
```

The opened port should be closed when the whole application is exited, by executing:

```
stage_control.MCRS_close()
```

4.2.2. Charge-coupled Device Camera

The camera used in the experimental setup is a charge-coupled device 12-bit camera system (PCO.1300) with a maximum resolution of 1392x1040 pixels.

To open the camera and establish connection, following instructions are given in the documentation accompanying the DLL file:

```
SC2_SDK_FUNC int WINAPI PCO_OpenCamera(HANDLE* ph)
```

The corresponding ctypes data types can be looked up in table 4.1. The star symbol after `HANDLE` denotes that a pointer, pointing to a variable, has to be passed to the function as an argument. One can create a pointer using the ctypes function `ctypes.byref(var)`, where the `var` denotes the addressed variable.

Opening the camera and assigning the handle parameter of the opened camera device can look as follows:

```
ccd_control=ctypes.WinDLL("SC2_Cam.dll")
ph=ctypes.c_void_p()
ccd_control.PCO_OpenCamera(ctypes.byref(ph))
```

Other functions are called in the same manner. To record an image, a simple implementation can be programmed by calling these functions:

```
PCO_OpenCamera
PCO_SetDelayExposureTime
PCO_ArmCamera
PCO_GetSizes
PCO_Allocate_Buffer
PCO_SetRecordingState
PCO_AddBufferEx
PCO_SetRecordingState
PCO_RemoveBuffer
PCO_CloseCamera
```

4.3. Conclusion

In this chapter the methods by which the various devices of the Paul trap experimental setup have been integrated, are introduced and explained. With the use of two modules (serial and ctypes) it is possible to interface with the devices from within the Python environment. The control of all the devices could be bundled into one control software, which provides the basis for the deployment of feedback systems and complex algorithms.

5. Algorithms and Procedures

One main advantage of bundling the control of the devices used in the ion trap into one program is to provide easy communication among them. Algorithms and procedures can be utilized that have access to the control of the devices and to the data provided by them. In that way one can set up a feedback system that reacts accordingly to certain changes in the data. For example, the feedback system can use an image provided by the camera, evaluate it according to certain specifications and adjust the linear motor stages, resulting in a position or focus change. With the help of feedback loops the program can monitor and track the experimental setup and automatically correct parameters if it sees fit.

Specifically in the ion trap experimental setup and as a part of this project, a focus automation and stabilization (section 5.1) and a center stabilization (section 5.2) are implemented and tested. Their certain specific use will be motivated in the corresponding subsections, but they both have the goal of making long term measurements with the spectrometer possible and thus increasing the collection efficiency of the luminescence light.

In this chapter I will quickly introduce integral tools that were necessary in the implementation of the algorithms. Then I will explain my approaches to a focus automation and a center stabilization.

OpenSource Computer Vision and NumPy

OpenSource Computer Vision [23], more commonly known as OpenCV, is a free library of programming functions for advanced image manipulation and processing and computer vision. The library is cross-platform and was originally written in C++. However, there is a Python wrapper for the original C++ implementation that makes it available to use within Python.

The Python wrapper of OpenCV highly relies on another Python module called NumPy [24], which enables the user to store and manipulate data in a MATLAB-style syntax. NumPy also comes with a very comprehensive math implementation; it is seen as one of the fundamental packages for scientific computing with Python.

In this project I will use OpenCV together with NumPy to determine a sharpness value of the picture and track the position of trapped particles.

5.1. Autofocus

In order to yield convenient results when measuring the luminescence spectrum of particles, the trapped object needs to be in the focal point of the objective. To achieve this the whole ion trap can be moved along the y-axis with a translation stage. (see figure 2.1).

The position of the particle comes about from the forces acting upon the particle and the forces created by the superposition of the ion trap's potential and the potential of interfering and unwanted fields. Therefore the reached equilibrium of all these forces determines the position of the minimum, and therefore the particle's position.

In an ideal Paul trap the position of a trapped particle doesn't change. The following only applies to real Paul traps where imperfections and asymmetries in its geometry are unavoidable.

There are various factors that affect the particle's position:

- The properties of the particle: Differences in mass-charge-ratio (or size) lead to varying positions due to gravity and reactions to stray fields. Therefore the focusing has to be adjusted for every particle.
- The parameters of the ion trap: A change in the frequency and voltage amplitudes of the potential (see equation 2.2) lead to positional changes.
- Slowly changing interfering electromagnetic stray fields in the vicinity of the ion trap.
- Unavoidable misalignment in the experimental setup: The y-axis and the x-axis are correlated, that is, a positional change in the x-axis, caused by a change in the segment voltages or even by the intensity of the laser, yields a positional change in the y-axis as well.

It is a consuming task to constantly readjust the linear motor stage to keep the particle in focus. Therefore an automation of this process would facilitate the work with the ion trap. Additionally, an automated process could assure, through constant tracking, that the particle stays in the focal point throughout a long measurement and thus yields more consistent results.

In the following chapter such an autofocus automation process is explained and implemented.

Brief Overview

The autofocus algorithm scans an image provided by the CCD camera and computes a value that corresponds with the focus level (section 5.1.1). Assuming that the focus level can be improved, the algorithm iteratively moves the linear motor stage along the y-axis to find the position that corresponds with the maximum focus level (section 5.1.2).

5.1.1. Computation of the Focus Level

Computation of the focus level from an image is not a straightforward task. Many different methods to obtain the sharpness information have been proposed in the literature [25]. Usually, a calculation is done that estimates the blurriness or sharpness of an image. For example, one can scrutinize the sharpness of edges in the image, that is, how fast (in terms of distance) pixel intensity values change from low to high.

However, the determination of the focus level in the ion trap experiment is complicated: First, it is worth noting that the focus level changes on a large scale, resulting in a much blurrier and enlarged image of the particle when it is out of focus. Scrutinizing the edges of the image is therefore not a convenient approach, since the image information greatly changes when it is moved in and out of focus. Furthermore, interference results in unwanted edges. Second, the computation is complicated by the motion of the trapped particle which comes about from collision with air particles (Brownian motion) and the characteristics of a real ion trap potential. To counter these difficulties, an algorithm as been chosen that correlates with the size of the image of the particle, indicating the best focus level when the image of the object has minimum size. Additionally, the computation of the focus level is carried out for every picture taken by the CCD camera and averaged over a given period of time.

In the following section such an algorithm is explained and tested using different averaging times.

Focus Measure Algorithm

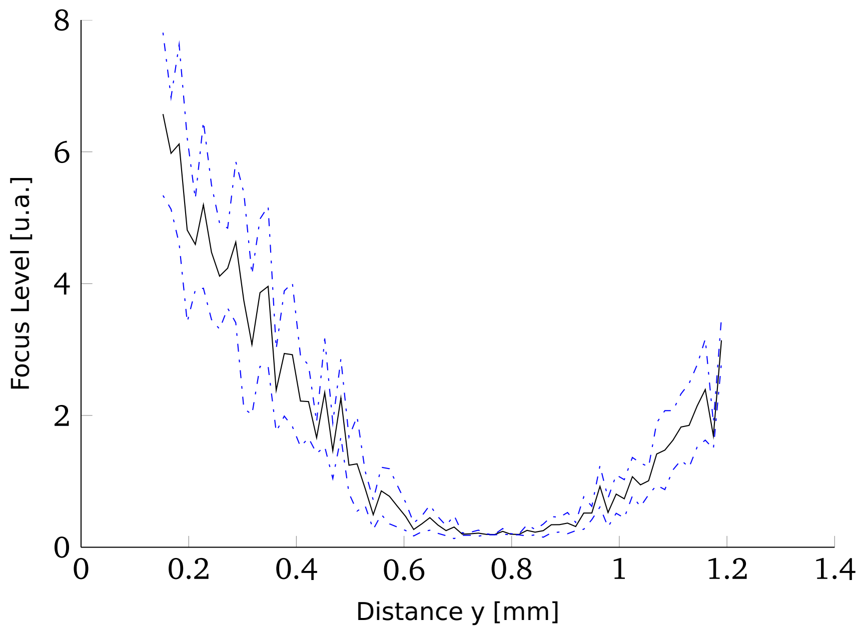
The image $f(x,y,)$ taken by the CCD camera is a gray scale pixel matrix with values between one and zero. The algorithm employed computes a measure correlating to the size of the image of the particle. The image is first normalized to correct different brightness values. This is necessary, otherwise the image of the particle can not effectively be distinguished from its background. Then, it is possible to simply sum over all pixels, using the pixel's intensity value as a weighting value, to obtain a measure of the size of the image of the particle:

$$FocusLevel = \sum_{x=1}^{X}\sum_{y=1}^{Y} f(x,y). \tag{5.1}$$

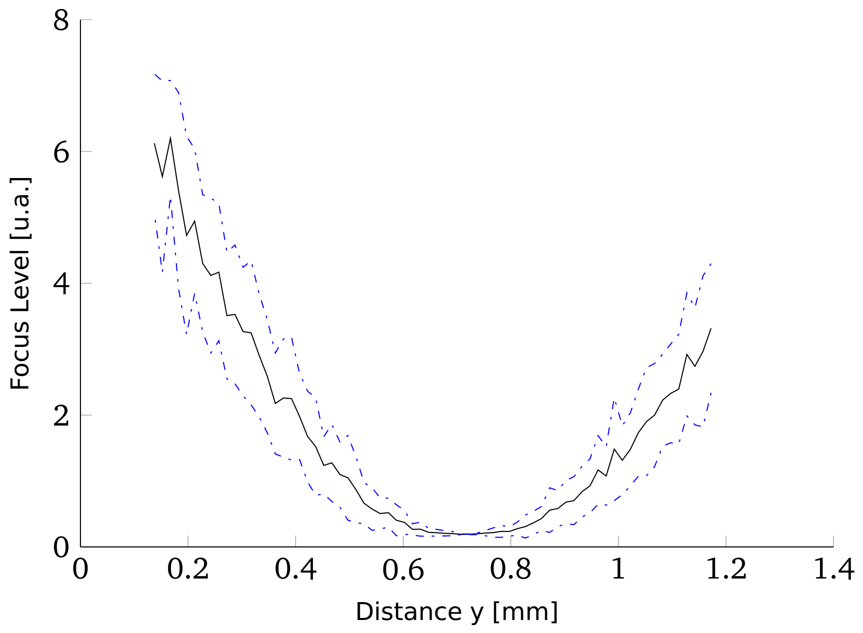
An implementation of this simple algorithm using NumPy looks as follows:

```
#normalizing:
img=img/numpy.amax(img)
#focus level:
focuslevel=numpy.sum(img)
```

To test the efficiency of the introduced algorithm, the translational stage scans over a wide interval with a very fine step size, while the focus level is computed for every position. Also, to test the effect of averaging over a period of time, the focus level is computed for every image and averaged over one second and five seconds. The result is depicted in figure 5.1.



(a)



(b)

Figure 5.1.: Focus level scanned over a distance of ≈ 10 mm with averaging time of (a) one second and (b) five seconds. The black line denotes the focus level and the blue line the standard deviation range.

The best possible result is a smooth unimodal curve with a narrow minimum. One can clearly see the smoothing effect of the longer averaging time.

5.1.2. Search Best Focus Level

The task of the search algorithm is to find the position of the linear motor stage that corresponds to the image with the best focus level. Therefore the linear motor stage is iteratively moved while the focus level is constantly compared to previous positions. There are several search algorithms reviewed in the literature [26], however they all apply to specific arrangements and rely on a very efficient focus measure.

Since the determination of the focus level can sometimes be complicated, an averaging time can be specified by the user, over which several measurements are made and the arithmetic mean and its standard deviation are computed. With that information one can counter a weakness of the focus measure. The standard deviation comes into great use when comparing different focus level measurements of different positions. One can only attest an improvement when the difference exceeds the standard deviation.

In the following section I will outline my search algorithm: Assuming that the focus level can be improved, the focus level is measured and subsequently the linear motor stage moved in an arbitrary direction. Then the focus level is measured again and compared to the previous results whereas the difference has to be greater than the standard deviation of the previous focus level. If this is not the case, the linear motor stage is moved one step further and values are compared again to the first measurement taken until a crucial change can be noticed, and thus it is deduced in which direction the focal point can be expected.

In a second step the algorithm moves the stage in that determined direction with a fixed step size. In this project I will employ an algorithm that incorporates a coarse-to-fine-search, that is the step size is fixed to a coarse value until a peak is found, which is indicated in the changing sign of the comparison focus level. Then the algorithm moves the linear motor stage in the opposite direction while decreasing the step size to a finer value.

5.1.3. Focus Stabilization

Once the particle is positioned in the focal point of the objective, an algorithm can be employed to make sure it stays in the focal point. It will measure the focal level when started and assume that it is the value corresponding to the position of the focal point. Then, it will constantly check the focus level and compare it the previous level. If the difference exceeds the standard deviation it will try to move it back in focus, first, by determining the direction and then iteratively moving the translational stage, described in section **??**. The algorithm will pause once the focus level is within a predetermined range of the best focus level.

The algorithm could be successfully tested: A diamond cluster was moved 4266 steps in one direction over a period of 30 minutes to keep the particle in the focal point. 4266 steps corresponds to a distance of $d \approx 0{,}21\,\text{mm}$.

5.2. Center Stabilization

To yield convenient results when measuring the luminescence spectrum the image of the particle has to be within the slit of the spectrometer which, through proper alignment, corresponds with the center of the image provided by the CCD camera. To navigate the trapped particle to the focal point of the imaging system, the whole ion trap can be moved in the z-direction with the z-axis of the linear motor stage and the minimum of the potential in the x-direction can be changed through adjusting the voltages of segments six and eight (see figure 2.1 and 2.3).

As discussed in 5.1, the particle's position is dependent on various factors. Once the particle is focused and centered, fluctuations in the light pressure of the laser and slowly varying interfering fields[1] can move the particle out of the center and thus out of the field of vision of the spectrometer. Therefore, recording a luminescence spectrum only yields convenient results for the time span the particle is actually positioned in the center of the camera image. To extend the time span and thus improve the quality of the spectrum, an algorithm is employed that constantly tracks the position of the trapped particle and moves the linear motor stage or adjusts the segment voltages to correct the position accordingly.

In the following chapter such a center stabilization process is explained and implemented.

Brief Overview

Initially, the center stabilization process takes into account the particle's own motion. Therefore the particle is tracked over a given period of time and for every image its position is determined (section 5.2.1). After the period of time, the particle's position (arithmetic mean) and the standard deviation are calculated. The standard deviation serves as a radius for a circle in which the particle moves due to its own motion[2]. Once the particle leaves that circle the algorithm knows that its position must be corrected

1 ↑given that the voltage and function generators produce stable output
2 ↑This measure can also be made of use when testing the trap's parameter in regard to stability

(section 5.2.2). In that way the algorithm can distinguish between the particle's own rapid movements and slow drift in position due to interfering electromagnetic fields.

5.2.1. Position Tracking

The image provided by the CCD camera is a grayscale pixel matrix containing values between zero and one. If the pixel's intensity value are interpreted as its mass, one can easily calculate the particle's position using formulas that are known from center-of-mass calculations. With this analogy one can compute the moments as follows:

$$m_{ij} = \sum_{x=1}^{X}\sum_{y=1}^{Y}\left(f(x,y)\, x^i\, y^j\right). \tag{5.2}$$

Determination of the particle's x- and y-position:

$$\bar{x} = \frac{m_{10}}{m_{00}},\ \ \bar{y} = \frac{m_{01}}{m_{00}}. \tag{5.3}$$

The implementation in OpenCV is straightforward, since it already provides such a function. To make the computation more robust against noise, one can simply blur the image without introducing any crucial error before the computation. Additionally, the particle's position is only considered when the "total mass" exceeds a threshold limit. This limitation is sometimes necessary when the particle moves around a lot, to a point where it can even move out of the laser beam and thus result in a dark image. In that case the algorithm would confuse noise with the particle. To avoid this, one can simply set a lower bound.

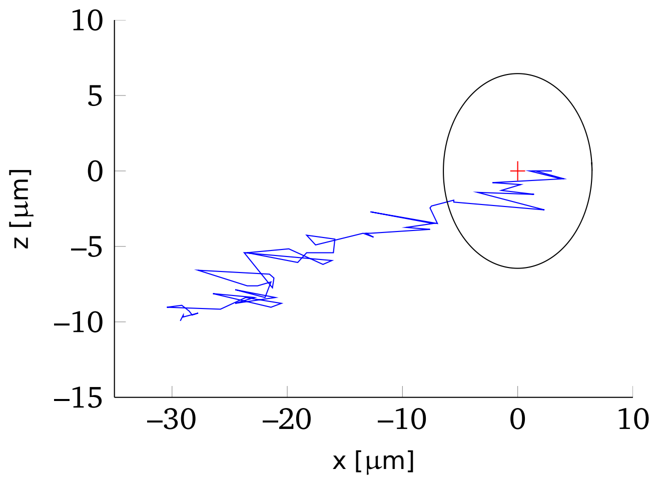
```
#moments:
moments=cv2.moments(gray)
area=moments['m00']
if area>1000:
        x=moments['m10']/area
        y=moments['m01']/area
```

5.2.2. Correction of Position

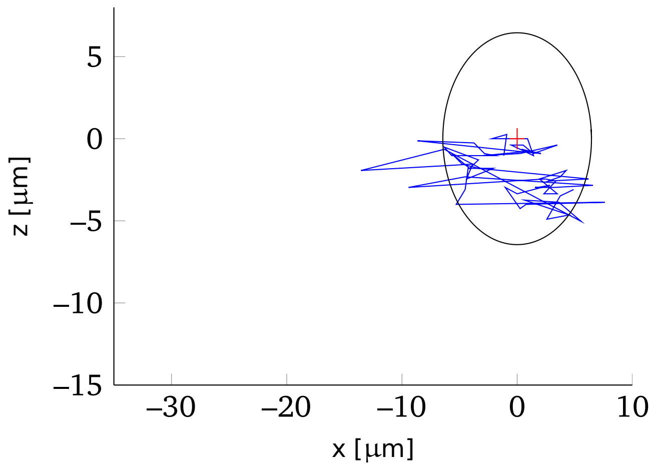
Once the position of the particle has been detected outside of the circle, an algorithm tries to correct its position as fast as possible, using the fine control of the segment voltages in the x-direction and the translational stage in z-direction (see figure 2.1). Since the magnitude of the positional shift in x-direction is dependent on the voltages applied to the segments and on the particle's charge-mass ratio, a single calibration is inappropriate. Therefore the algorithm first determines the direction of the correction by simply comparing the current position to the center position. In a second step, the algorithm tries to correct the position by moving the particle with an initial step size. If the particle is within a smaller set circle around the center, the algorithm stops; if is still outside, the algorithm will establish a correlation between the step size and the changed distance and use it to calibrate. In a last step it uses this calibration to move the particle within a circle in the center.

5.2.3. Test and Comparison

Over the period of 30 minutes the position of a diamond cluster has been tracked and its trajectory depicted in figure 5.2. One can clearly see how the particle leaves the center and therefore the field of view of the spectrometer without center stabilization. Once center stabilization is activated, the positional drift is corrected accordingly.



(a)



(b)

Figure 5.2.: Particle trajectory over a period of 30 minutes. (a) without center stabilization (b) with center stabilization. The radius of the circle is the standard deviation determined while measuring the particle's own rapid movement

5.3. Conclusion

In this chapter algorithms and procedures for focus and position stabilization of the images of trapped particles are motivated, implemented and tested. Focus and position stabilization is necessary to counter the effects of interfering fields. In that manner, it can be ensured that the image of a trapped particle stays in focus and within the field of view of the spectrometer, thus allowing long-term measurement. The implemented algorithms have been successfully tested with a diamond cluster.

6. Summary and Outlook

In this project a control software has been developed to facilitate experiments conducted on trapped particles in a Paul trap. In that context, the necessary programming languages (Python, Enaml) have been motivated and introduced. Basic concepts of those programming languages have been explained and illustrative examples shown to provide the fundamentals in designing a graphical user interface with data synchronization.

The control of the devices in the experimental setup of the Paul trap (CCD camera, 2-axis stage, micro controller) have been successfully integrated into one program, providing the basis for complex feedback algorithms. Algorithms such as a focus and position stabilization have been developed and implemented. Difficult experimental conditions made the computation of the focus level or the determination of the position of the particle complicated and prone to errors. However, through the use of the right algorithm or different approaches, such as averaging over a period of time or normalizing, these difficulties could be coped with. All the implemented algorithm have been successfully tested with a diamond cluster.

In future experiments, diamonds with less NVs will be examined, therefore it is essential that the developed algorithms work properly. It may be necessary to further improve the algorithms and adapt them do different particle and Paul trap characteristics. Furthermore, measurements will be conducted in vacuum. In that case the particle's own motion will decrease. Consequently, the algorithms need to be tested again. However, I assume that they will even work more accurately.

Bibliography

[1] W. Paul, H. P. Reinhard, and U. Von Zahn. „Das elektrische Massenfilter als Massenspektrometer und Isotopentrenner“. In: *Zeitschrift fur Physik* 152.2 (1958), pp. 143–182.

[2] W. Paul. „Electromagnetic Traps for Charged and Neutral Particles (Nobel Lecture)“. In: *Angewandte Chemie International Edition in English* 29.7 (1990), pp. 739–748.

[3] Donald J. Douglas, Aaron J. Frank, and Dunmin Mao. „Linear ion traps in mass spectrometry“. In: *Mass Spectrometry Reviews* 24.1 (2005), pp. 1–29.

[4] S. Arnold and L. M. Folan. „Fluorescence Spectrometer for a Single Electrodynamically Levitated Microparticle“. In: *Review of Scientific Instruments* 57.9 (1986), pp. 2250–2253.

[5] W. H. Oskay et al. „Single-Atom Optical Clock with High Accuracy“. In: *Phys. Rev. Lett.* 97 (2 2006), p. 020801.

[6] J. I. Cirac and P. Zoller. „Quantum Computations with Cold Trapped Ions“. In: *Phys. Rev. Lett.* 74 (20 1995), pp. 4091–4094.

[7] D. Kielpinski, C. Monroe, and D. J. Wineland. „Architecture for a large-scale ion-trap quantum computer“. In: *Nature* 417.6890 (June 2002), pp. 709–711. URL: http://dx.doi.org/10.1038/nature00784.

[8] Ferdinand Schmidt-Kaler et al. „Realization of the Cirac-Zoller controlled-NOT quantum gate“. In: *Nature* 422.6930 (Mar. 2003), pp. 408–411.

[9] Alexander Kuhlicke et al. „Nitrogen vacancy center fluorescence from a submicron diamond cluster levitated in a linear quadrupole ion trap“. In: *Applied Physics Letters* 105.7 (2014).

[10] T. D. Ladd et al. „Quantum computers“. In: *Nature* 464.7285 (Mar. 2010), pp. 45–53.

[11] Jörg Wrachtrup and Fedor Jelezko. „Processing quantum information in diamond“. In: *Journal of Physics: Condensed Matter* 18.21 (2006), S807.

[12] Sungkun Hong et al. „Nanoscale magnetometry with NV centers in diamond“. In: *MRS Bulletin* 38 (02 Feb. 2013), pp. 155–161.

[13] Romana Schirhagl et al. „Nitrogen-Vacancy Centers in Diamond: Nanoscale Sensors for Physics and Biology“. In: *Annual Review of Physical Chemistry* 65.1 (2014), pp. 83–105.

[14] R. F. Wuerker, H. Shelton, and R. V. Langmuir. „Electrodynamic Containment of Charged Particles“. In: *Journal of Applied Physics* 30.3 (1959), pp. 342–349.

[15] H. Winter and H. W. Ortjohann. „Simple demonstration of storing macroscopic particles in a “Paul trap”“. In: *American Journal of Physics* 59.9 (1991), pp. 807–813.

[16] Joachim Zoll. „Konstruktion, Fertigung und Charakterisierung einer Paulfalle zum Einfangen von Nanodiamten“. Diplomarbeit. Humboldt Universität zu Berlin, 2014.

[17] M. G. Raizen et al. „Ionic crystals in a linear Paul trap“. In: *Phys. Rev. A* 45 (9 1992), pp. 6493–6501.

[18] Guido van Rossum. *Python 2.7.11 documentation*. Ed. by Fred L. Drake. Python Software Foundation, 2016. URL: https://docs.python.org/2/ (visited on 02/28/2016).

[19] Guido van Rossum. „math — Mathematical functions“. In: *Python 2.7.11 documentation - The Python Standard Library* (2016). Ed. by Fred L. Drake. URL: https://docs.python.org/2/library/math.html (visited on 02/28/2016).

[20] S. Chris Colbert. *Enaml*. 2015. URL: https://github.com/nucleic/enaml (visited on 02/28/2016).

[21] Chris Liechti. „pySerial“. In: *pySerial 3.0 documentation* (2015). URL: http://pythonhosted.org/pyserial/pyserial.html (visited on 02/28/2016).

[22] Guido van Rossum. „ctypes – A foreign function library for Python“. In: *Python 2.7.11 documentation - The Python Standard Library* (2016). Ed. by Fred L. Drake. URL: https://docs.python.org/2/library/ctypes.html\#module-ctypes (visited on 02/28/2016).

[23] Gary Bradski. „The OpenCV Library“. In: *Dr. Dobb's Journal of Software Tools* 25 (Nov. 2000), pp. 120, 122–125.

[24] S. Chris Colbert Stéfan van der Walt and Gaël Varoquaux. „The NumPy Array: A Structure for Efficient Numerical Computation“. In: *Computing in Science & Engineering* 13 (2011), pp. 20–30.

[25] S. Pertuz, D. Puig, and M. A. Garcia. „Analysis of focus measure operators for shape-from-focus“. In: *Pattern Recognition* 46.5 (2013), pp. 1415 –1432.

[26] H. Mir et al. „An autofocus heuristic for digital cameras based on supervised machine learning“. In: *Journal of Heuristics* 21.5 (2015), pp. 599–616.

[27] Guido van Rossum. „threading – Higher-level threading interface“. In: *Python 2.7.11 documentation - The Python Standard Library* (2016). Ed. by Fred L. Drake. URL: `https://docs.python.org/2/library/threading.html` (visited on 02/28/2016).

A. Multithreaded Programming with Python

When a function is called that needs to run for a longer time, in graphic user interface (GUI) programming the user interface will wait for the function to finish and freeze in the meantime. To circumvent this impractical behavior one has to employ the concept of parallel computing; that is, for the evaluation of that function the program will create a background process (also called thread), so that the GUI, running in the main process, is still responsive in the meantime. Parallel computing in Python can be realized with the build-in module threading [27]. Here is a simple example use of it:

```
import time
from threading import Thread

class Thread_example():
    signal=False

    def worker(self, limit):
        p = 0
        self.signal=True
        while self.signal and p < limit:
            p += 1
            time.sleep(0.2) #program sleeps for 0.2 seconds

    def start_thread(self, limit):
        if self.signal==False:
            thread = Thread(target=self.worker, args=(limit,))
            thread.daemon = True #
            thread.start()

    def stop_thread(self):
            if self.signal==True:
            self.signal = False
```

A thread consists of two definition blocks, one defining the actual task (here called worker) and the other organizing and launching the task in a background process (here called start_thread). To start the background process the class from the above example needs to be instantiated first. If the actual task takes an argument, it needs to be passed to the start_thread function, which internally passes it to the worker function.

```
thread_example=Thread_example()
limit=100
thread_example.start_thread(limit)
```

Usually the background process runs until the task is finished, in this case when it reaches the limit which is passed over as an argument. However, there are cases, where a background process runs until the user wishes to terminate it. An implementation of this functionality can be realized like that: The worker function checks with each iteration, whether an attribute of its class (here called signal) is set to True, if not, it will finish. One can then, with the help of another definition of a function, change the boolean value of this attribute and therefore terminate the background process:

```
thread_model.stop_thread()
```

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne unerlaubte Hilfe angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.
Jonas Engelmann