ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Mose Müller

A flexible python framework for generating pulse sequences for multi-zone operations in ion traps

Master Thesis

Institute for Quantum Electronics, Swiss Federal Institute of Technology (ETH), 8093 Zurich, Switzerland

Supervision

Alfredo Ricci
Dr. Carmelo Mordini
Prof. J. P. Home

December 20, 2022

Abstract

Quantum information processing with trapped ions is a large field of research, exploring different approaches to develop computing devices that can potentially outperform classical computers. In this thesis, we designed and developed the bottom-up python framework `pycrystal` that allows to create a representation of such systems. The framework provides the user with a high-level interface for writing pulse sequences for trapped ion (and potentially cold atom) setups and an interface for executing the pulse sequences while performing scans on its parameters. Single- and multi-zone measurements of the phase coherence of a single ion have been demonstrated with that package using different variations of Ramsey experiments on a surface-electrode trap with integrated optics. Phase fluctuations due to differential laser phase noise in different zones were observed and cancelled by mapping the information on the optical qubit to a Zeeman qubit in the electronic ground state of the ion.

Acknowledgements

First of all, I want to thank Alfredo and Carmelo for their wonderful supervision. I am very thankful for all the physics you have patiently taught me, often not just once, due to my lack of understanding, and all the feedback I received from you. I enjoyed the times we spent together in the lab where we often tried to find bugs and sources of errors that were constantly hiding somewhere but also did cool physics.

I want to thank Martin Stadler for explaining so many things about the control system and helping me to discover new programming tools and how to apply them. I also want to thank Alex for our discussions of pycrystal and the ionpulse sequence generator. It was a pleasure working with both of you.

Thanks to Jonathan for sparking my interest in the area of quantum information processing through the lectures you were giving, and also for allowing me to conduct my master's thesis in the TIQI group.

I want to thank all members of the TIQI group. I really enjoyed the time I could spend with you and appreciated the conversations and discussions with each of you.

Finally, I also want to thank my family, especially my lovely wife Monica, who gave me all the mental and practical support I needed, and my Lord Jesus for creating and loving me as I am. I am so thankful that I know you and can call you my family.

Contents

Abstract i

Acknowledgements ii

Code Listings v

List of Acronyms vi

1 Introduction 1

2 Quantum Information with Calcium Ions 3

2.1 Encoding Qubits in Calcium . . . 3

2.1.1 Calcium Energy Level Structure . . . 4

2.1.2 Qubits in Calcium and Decoherence Effects . . . 4

2.2 Quantum Operations . . . 5

2.2.1 Single-Qubit Gates . . . 6

2.2.2 State Preparation . . . 7

2.2.3 Readout and Tresholding . . . 8

2.3 Cooling Schemes . . . 9

2.3.1 Doppler Cooling . . . 10

2.3.2 EIT Cooling . . . 10

2.4 Ramsey Experiment . . . 11

2.4.1 Phase Noise . . . 12

2.4.2 Phase-insensitive Ramsey Scheme . . . 12

3 Motivation for pycrystal 14

3.1 Control System . . . 14

3.2 Creating and Running Experiments . . . 15

3.2.1 Evaluation . . . 16

4 `pycrystal` 19

4.1 Ionpulse Sequence Generator . . . 20

4.2 Design . . . 20

4.2.1 `pycrystal.beamline` - Defining the Hardware . . . 21

4.2.2 `pycrystal.crystal` - Abstracting Ion Crystals . . . 24

4.2.3 `pycrystal.parameters` - Interface between Database and Experimental Parameters . . . 26

4.2.4 `pycrystal.experiment` - Building Experiment Pulse Sequences . . . 28

5 Experimental Setup 30

5.1 The Cryo Setup . . . 30

5.1.1 Fastinos . . . 32

6 Ramsey Characterisation Experiment 33

6.1 Single-zone Ramsey- and Spin Echo Sequences . . . 34

6.1.1 Code . . . 34

6.1.2 Results . . . 35

6.2 Multi-zone Ramsey Sequence . . . 37

6.2.1 Code . . . 37

6.2.2 Results . . . 38

6.3 Laser Phase-insensitive Multi-zone Ramsey Sequence . . . 38

6.3.1 Code . . . 38

6.3.2 Results . . . 39

7 Conclusion 41

A Ramsey- and Spin-echo Data 43

Bibliography 48

Declaration of Originality 49

Code Listings

4.1 Example of a single-pass `AOM` instantiation. 21
4.2 Example of a `BeamLineLayout` initialisation. 22
4.3 `BeamLineParameterisation` example. 23
4.4 `BeamLine` example. 24
4.5 `Crystal` class example. 25
4.6 Setting the database for the parameter classes. 26
4.7 Defining and using parameters. 27
4.8 Scanning parameters. 28
4.9 Defining experiments. 28
4.10 Performing scans with `pycrystal`. 29
6.1 Ramsey experiment class. 33
6.2 Performing a Ramsey scan. 34
6.3 Single-zone Ramsey experiment sequence. 35
6.4 Multi-zone Ramsey experiment sequence. 37
6.5 Phase-insensitive multi-zone Ramsey experiment sequence. 39

List of Acronyms

AOD acousto-optic deflector

AOM acousto-optic modulator

AWG arbitrary waveform generator

BSB blue sideband

CCW current-carrying wire

CPU central processing unit

DAC digital-to-analogue converter

DC direct current

DDS direct digital synthesis

EIT electromagnetically induced transparency

FPGA field-programmable gate array

GUI graphical user interface

M-ACTION Modular Advanced Control of Trapped IONs

PMT photomultiplier tube

QCCD quantum charge-coupled device

RF radio-frequency

RSB red sideband

RWA rotating wave approximation

SET surface-electrode trap

TIQI Trapped Ion Quantum Information

TTL transistor-transistor logic

UHV ultra-high vacuum

Chapter 1

Introduction

Quantum computer harness the properties of quantum mechanics to solve problems that are thought to be intractable by classical computers. There are different promising platforms for quantum computers, one of which is trapped ions, which utilise individually controlled ions to store quantum information. Since it was first proposed by Cirac and Zoller in 1995[1], much experimental progress has been made to demonstrate the fundamental elements required to build a quantum computer, such as quantum state preparation[2, 3], manipulation[3, 4] and read-out[3].

Scalability and control of such systems is an ongoing challenge. A solution to this problem was put forward by D. Kielpinksi et al. in 2002[5] where it was proposed to use a quantum charge-coupled device (QCCD) architecture, in which information is distributed across a micro-chip by physically moving ions with dynamic electric fields and so linking them together. This architecture provides a building block that can be used to perform extended calculations in a modular and scalable way[6].

Another crucial step in the direction of scaling ion trap systems is the integration of optical interfaces into the trapping chips. This is because the delivery of the laser beams to the qubits becomes increasingly complicated as the size of the system grows. *Integrated optics* in surface-electrode traps (SETs) was first proposed in Ref. [7] as a solution to that challenge. Following that proposal, multiple experiments, including the one I worked on during my thesis, have already demonstrated the integration of optical elements [8, 9, 10, 11].

Interfacing with the flexibility of such systems on the control software side is a big challenge. This thesis describes `pycrystal`, a python library for the generation of control sequences that I developed during my master thesis, as my attempt to create a framework that solves this problem. One of its main goals was to provide an easy-to-use interface allowing for a high-level description of the pulse sequences the user wants to run on his system. The library is flexible and not only allows the description of different trap architectures but also different ion species and their isotopes.
The thesis also includes results of single- and multi-zone Ramsey experiments that we obtained with my package on a setup implementing a QCCD architecture on a SET with integrated optics.

The chapters are organised as follows:

Chapter 2 introduces the basics for quantum information processing with calcium ions.

Chapter 3 gives an introduction to the control system used in our research group and motivates the developments of the python library `pycrystal`.

Chapter 4 introduces the `pycrystal` library by presenting the design choices I made and giving simple implementation examples.
Chapter 5 gives an overview of the setup `pycrystal` has been tested on.
Chapter 6 presents the Ramsey characterisation experiments that we have performed on our setup.
Chapter 7 concludes the thesis with a summary of the results and an outlook for future work.

Chapter 2

Quantum Information with Calcium Ions

Preparation, preservation, manipulation and readout of quantum states are important criteria for a useful implementation of a quantum computer. It must achieve a balance between preventing decoherence effects and allowing interaction with the qubits. This means that we must be able to ensure that both the magnitudes α, β and relative phase ϕ of the general qubit state[12]

$$\alpha\,|1\rangle + \beta\,|0\rangle = |\alpha|\,|1\rangle + |\beta|e^{i\phi}\,|0\rangle\,, \tag{2.1}$$

change in a predictable manner such that we can preserve the information encoded in it. Moreover, to manipulate the qubits, i.e. to apply gates to the qubits, we must be able to change their state by means of external electrical fields and have the qubits interact with each other. Furthermore, the preparation and readout of the qubits require that information can be extracted from them. For state preparation, this means that we must be able to reduce the entropy of the qubit to reach a deterministic initial state at the beginning of the sequence.

In section 2.1 I will give two ways of encoding a qubit in calcium ions. Section 2.2 shows how quantum operations can be performed on those qubits, followed by the cooling schemes used in our experiments in section 2.3. The chapter closes with an explanation of a Ramsey sequence in section 2.4 which will be used later in the thesis.

2.1 Encoding Qubits in Calcium

The ion of choice in our setup is calcium $^{40}Ca^{+}$. To show how qubits are encoded in calcium ions, we will first have a look at the relevant part of their energy level structure and then elaborate on how we can preserve information in a selected pair of states.

2.1.1 Calcium Energy Level Structure

The energy level structure of the lowest eigenstates of a $^{40}Ca^{+}$ ion is shown in Fig. 2.1. Lasers at wavelengths $\lambda = 397\,\text{nm}, 854\,\text{nm}$ and 866 nm can be used to excite electric dipole transitions. They can be driven very fast and with low intensities which makes them unsuitable for storing information as they have very short lifetimes. The $S_{1/2} \rightarrow D_{5/2}$ transition, on the other hand, is an electric quadrupole transition as it is dipole forbidden. This makes the $D_{5/2}$ level meta-stable, i.e. it has a longer lifetime than other excited states. This transition is driven with a 729 nm laser. The $P_{1/2}$ level, used for state preparation, cooling and readout, decays into the $D_{3/2}$ level which is also meta-stable. Thus, we have to repump it as otherwise, the electron will stay there.

Applying a magnetic field to the ion lifts the degeneracy of the levels, giving rise to well-defined quantum states. We can also use those equally spaced energy sub-levels to encode a qubit.

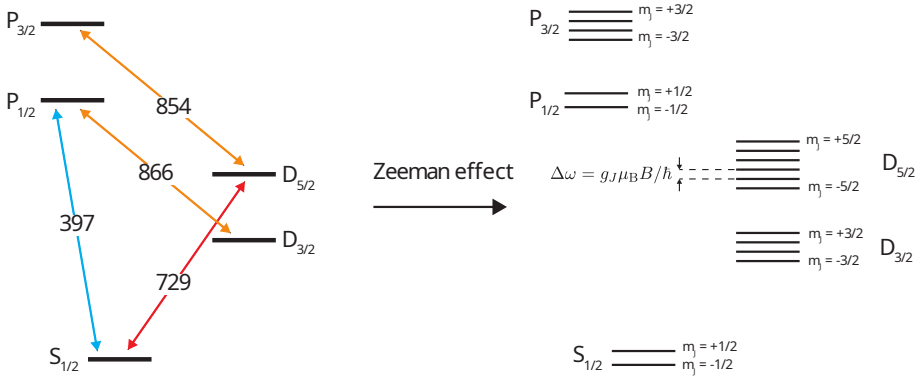


Figure 2.1: Calcium $^{40}Ca^{+}$ ion level structure. The fine-structure energy states are shown on the left, specified by L_J. The blue, red and orange arrows denote the lasers and their corresponding wavelengths for the transition. The Zeeman splitting of the levels is shown on the right side: when applying a magnetic field to the calcium ion the degeneracy of the levels will get lifted resulting in states split into $2J + 1$ equally spaced sub-levels. Figure taken from [11]

2.1.2 Qubits in Calcium and Decoherence Effects

To maximise the coherence time of the qubits, we have to address two issues: the decay rate of the states and the phase de-coherence due to the noise of the quantisation magnetic field (B-field) or phase noise in the laser. The solution to the first problem gives us two ways of encoding a qubit in the electronic levels of a calcium ion.

Due to the long lifetime of the $D_{5/2}$ level, it can be used together with the $S_{1/2}$ level to encode an optical qubit ($|0\rangle_o = |D_{5/2}, m_J = -1/2\rangle, |1\rangle_o = |S_{1/2}, m_J = -1/2\rangle$). Its coherence is limited by both the B-field and laser frequency noise. The magnetic field changes the energy separation of the qubit states, causing a phase difference between them. Fluctuations of it thus cause phase fluctuations of the qubit. Laser frequency noise directly translates into phase noise as the instantaneous frequency is the time derivative of the phase[13, pp. 74-76]. The phase coherence can be probed by, for example, a Ramsey experiment (see section 2.4).

We can also encode a Zeeman qubit in the $m_J = +1/2$ and $m_J = -1/2$ levels of $S_{1/2}$ ($|0\rangle_z = |S_{1/2}, m_J = +1/2\rangle, |1\rangle_z = |S_{1/2}, m_J = -1/2\rangle$). As the energy states are very close to each other

the spontaneous emission is suppressed[14]. The Zeeman qubit also suffers from the decoherence coming from the magnetic field, but single qubit operations are not subject to additional phase noise. This is because they are not performed with a laser but rather with a radio-frequency (RF) pulse through a current-carrying wire (CCW). However, due to the large wavelength, coupling the electronic states of ions to their motion becomes infeasible and therefore two-qubit gates are not possible[11].

2.2 Quantum Operations

Having shown how we can encode qubits into the electronic structure of calcium ions, we want to revisit the other requirements for a useful quantum computer.

Interactions with ions are performed using lasers tuned to the resonance frequency of two levels. We make the assumption that we use monochromatic light with frequency ω that only couples one pair of internal states of the ion. Then, we can think of the ion as an effective two-level system with ground state $|0\rangle$ and excited state $|1\rangle$ with energies E_0 and E_1, respectively. We can model this two-level system with the following Hamiltonian

$$H_0(t) = E_0 \left|0\right\rangle\!\left\langle 0\right| + E_1 \left|1\right\rangle\!\left\langle 1\right| \tag{2.2}$$

and the interaction with the time-dependent Hamiltonian in the dipole approximation

$$H_1(t) = \frac{e\mathcal{E}_0}{2}(\vec{r}\cdot\vec{\varepsilon})\cos(\omega t + \phi), \tag{2.3}$$

where e is the elementary charge, $\vec{r}$ is the position vector, $\vec{\varepsilon}$ the polarisation vector of the electric field, $\mathcal{E}_0$ its amplitude, ω its frequency and ϕ its phase. Expressing $H_1(t)$ in the $\{|0\rangle, |1\rangle\}$ basis, we can rewrite it as

$$H_1(t) = \hbar\frac{\tilde{\Omega}}{2}\left|0\right\rangle\!\left\langle 1\right|\cos(\omega t + \phi) + h.c., \tag{2.4}$$

where $\tilde{\Omega} = \frac{e\mathcal{E}_0}{\hbar}\left\langle 0\right|(\vec{r}\cdot\vec{\varepsilon})\left|1\right\rangle$ and $h.c.$ represents the hermitian conjugate of the first expression[1].

With the transformation $|i\rangle \to e^{iH_0 t}|i\rangle$ we can move H_1 into the rotating frame with respect to H_0. Thus, using $\omega_0 = (E_1 - E_0)/\hbar$ we can write the interaction Hamiltonian in the rotating wave approximation (RWA)[2] as

$$H_I(t) = \hbar\frac{\Omega}{2}\left|0\right\rangle\!\left\langle 1\right| e^{i[(\omega-\omega_0)t+\phi]} + h.c., \tag{2.5}$$

where Ω is called the Rabi frequency. Although the above derivation is carried out in the dipole approximation, eq. 2.5 is valid for both electric dipole and quadrupole transitions. The only quantity that is changing is the Rabi frequency which is given by

$$\Omega_d = \frac{e\mathcal{E}_0}{\hbar}\left|\left\langle 0\right|(\vec{r}\cdot\vec{\varepsilon})\left|1\right\rangle\right| \tag{2.6}$$

[1] The diagonal entries vanish due to atomic parity, $\langle 0|H_1|0\rangle = \langle 1|H_1|1\rangle = 0$.

[2] In the rotating wave approximation we neglect those terms in the Hamiltonian that oscillate rapidly, i.e. at the frequency $\omega + \omega_0$, and only keep the $\omega - \omega_0$ terms.

for the dipole transitions and

$$\Omega_q = \frac{e\mathcal{E}_0 k}{2\hbar}\left|\langle 0|\,(\vec{r}\cdot\vec{\varepsilon})(\vec{r}\cdot\vec{k})\,|1\rangle\right| \tag{2.7}$$

for the quadrupole transitions, where $k = \frac{2\pi}{\lambda}$ is the wavevector pointing in the propagation direction of the plane wave. The expression corresponds to the second-order term of the multipole expansion of the electric field and assumes that the dipole coupling between $|0\rangle$ and $|1\rangle$ is strictly zero[15]. It thus describes the interaction with the electric field gradient[11, pp. 15-18].

We now want to show that we can drive arbitrary single-qubit gates with pulses from a single laser modelled by the interaction Hamiltonian 2.5 (section 2.2.1) and then look at the key operations of state preparation (section 2.2.2) and readout (section 2.2.3).

2.2.1 Single-Qubit Gates

We can decompose any unitary operator on a single qubit into rotations around two axes and a global phase. A rotation is the action of a Hamiltonian of the form $\sigma_x = |1\rangle\langle 0| + |0\rangle\langle 1|\,, \sigma_y = i(|1\rangle\langle 0| - |0\rangle\langle 1|)$ or $\sigma_z = |0\rangle\langle 0| - |1\rangle\langle 1|$ on a qubit. An arbitrary state can be written in this form

$$|\psi\rangle = \cos(\theta/2)\,|0\rangle + e^{i\phi}\sin(\theta/2)\,|1\rangle \tag{2.8}$$

$$= \cos(\theta/2)\,|0\rangle + (\cos(\phi) + i\sin(\phi))\sin(\theta/2)\,|1\rangle\,, \tag{2.9}$$

where θ, ϕ are some angles. Applying a rotation matrix $R_{\hat{n}}(\tilde{\theta}) \equiv e^{-i\tilde{\theta}\hat{n}\cdot\vec{\sigma}/2}$ to that state rotates is by an angle $\tilde{\theta}$ around the axis $\hat{n}$, where $\vec{\sigma} = (\sigma_x, \sigma_y, \sigma_z)$[12, 16, pp. 174-177].

Here, we want to show that the phase of the interaction Hamiltonian 2.5 can be chosen such that it is in the form of either σ_x or σ_y and that the laser modelled by the Hamiltonian can indeed be used to generate qubit rotations, inducing the so-called *Rabi oscillations*. We can see this by calculating the propagator $U_I(t)$ of this Hamiltonian under which the qubit states undergo those oscillations[11, pp. 25-26]. Setting the detuning $\delta = \omega - \omega_0 = 0$ and defining the operators $\sigma_- = |0\rangle\langle 1|$ and $\sigma_+ = |1\rangle\langle 0|$ we can write the time-independent interaction Hamiltonian as

$$H_I = \hbar\frac{\Omega}{2}\left(\sigma_- e^{+i\phi} + \sigma_+ e^{-i\phi}\right). \tag{2.10}$$

Noting that $\sigma_\pm = (\sigma_x \mp i\sigma_y)/2$ we can transform that Hamiltonian into

$$H_I = \hbar\frac{\Omega}{2}\left(\sigma_x\cos(\phi)\ - i\sigma_y\sin(\phi)\right), \tag{2.11}$$

which is in the form of σ_x for $\phi = 0$ and σ_y for $\phi = \pi/2$.

Introducing the vector notation

$$|0\rangle = \begin{pmatrix}1\\0\end{pmatrix} \quad |1\rangle = \begin{pmatrix}0\\1\end{pmatrix}, \tag{2.12}$$

the propagator $U_I(t) = e^{-iH_I t}$ takes the following form

$$U_I(t) = \begin{pmatrix} \cos(\Omega t/2) & -i\sin(\Omega t/2)e^{-i\phi} \\ -i\sin(\Omega t/2)e^{i\phi} & \cos(\Omega t/2) \end{pmatrix}. \tag{2.13}$$

Under the action of that propagator a qubit $|0\rangle$ undergoes resonant Rabi oscillations

$$|\psi(t)\rangle = U_I(t)\,|0\rangle = \begin{pmatrix} \cos(\Omega t/2) \\ i\sin(\Omega t/2) \end{pmatrix} e^{-i\phi}. \tag{2.14}$$

The time after which the whole population is transferred to the other state $|1\rangle$ is called *pi-time*. The probability of finding the system in the $|0\rangle$ state after some time t is given by

$$P(0,t) = |\langle 0|\psi(t)\rangle|^2 = \cos^2(\Omega t/2) = \frac{1}{2}\left(1 + \cos(\Omega t)\right). \tag{2.15}$$

2.2.2 State Preparation

State preparation in calcium is an optical process where the population of the $S_{1/2}$ ground state is transferred to only one of its Zeeman sub-levels. Through this dissipative process, the ion is prepared in a deterministic initial state. For this thesis, we prepare the qubits in $|1\rangle = \left|S_{1/2}, m_J = -1/2\right\rangle$. Fig. 2.2 shows two different ways of achieving state preparation.

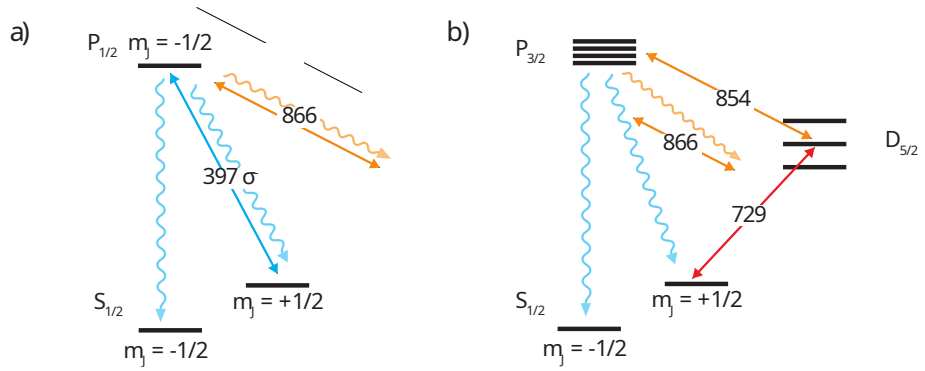


Figure 2.2: Calcium qubit state preparation. There are two schemes that can be used to prepare the qubit in the $\left|S_{1/2}, m = -1/2\right\rangle$ state: a) by optically pumping the $\left|S_{1/2}, m = +1/2\right\rangle \rightarrow \left|P_{1/2}, m = -1/2\right\rangle$ transition with a σ-polarised $\lambda = 397\,\text{nm}$ light while repumping the $D_{3/2}$ population with a laser at $\lambda = 866\,\text{nm}$, and b) by optically pumping the narrow $\left|S_{1/2}, m = +1/2\right\rangle \rightarrow \left|D_{5/2}\right\rangle$ transition while repumping the $D_{3/2}$ and $D_{5/2}$ levels with $\lambda = 854\,\text{nm}$ and $866\,\text{nm}$ light. Figure adapted from [11].

The first way is to drive the electrical dipole transition $\left|S_{1/2}, m = +1/2\right\rangle \rightarrow \left|P_{1/2}, m = -1/2\right\rangle$ with a circularly polarised $\lambda = 397\,\text{nm}$ laser travelling along the quantisation axis. Which levels are coupled is determined by the selection rules, more specifically the handedness of the circularly polarised light[17]. The excited state spontaneously decays with a probability of $\frac{1}{3}$ into the desired state $|1\rangle$. Thus, the qubit should be prepared after more than three scattering events on average. The other times it decays into either the $\left|S_{1/2}, m_J = +1/2\right\rangle$ or the $\left|D_{3/2}\right\rangle$ states which is why we have to use a $\lambda = 866\,\text{nm}$ repump laser. This scheme is limited by the alignment of the 397 nm

light and the purity of its polarisation.

The other state preparation scheme consists of driving the electrical quadrupole transition with a $\lambda = 729\,\text{nm}$ laser followed by 866 nm and 854 nm repumping lasers. The first laser optically pumps the $|S_{1/2}, m = +1/2\rangle \rightarrow |D_{5/2}\rangle$ transition. The coupled sub-level is selected by the frequency sensitivity of the energy levels. The 854 nm laser repumps the population from the $D_{5/2}$ levels to the $P_{3/2}$ levels which then decay into the $D_{3/2}$ and the $S_{1/2}$ levels. The 866 nm laser then repumps the population from the $D_{3/2}$ levels to the $P_{1/2}$ levels which again decay into the $D_{3/2}$ and the $S_{1/2}$ levels. One of the ground state sub-levels can be prepared after a few cycles of this scheme. Although the quadrupole state preparation is only prone to off-resonant excitation errors of the 729 nm laser, it requires much more power than the dipole scheme.

2.2.3 Readout and Tresholding

Readout of the two qubit-encodings has to be performed differently (see Fig. 2.3).

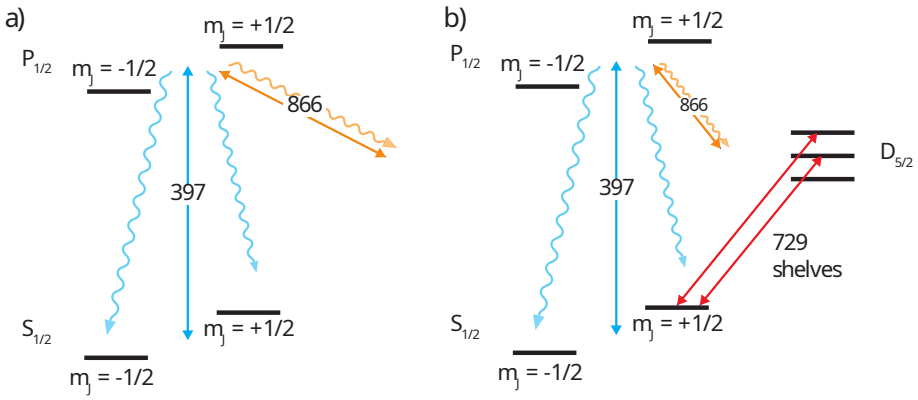


Figure 2.3: Calcium ion readout schemes. The counts of photons emitted by the $P_{1/2} \rightarrow S_{1/2}$ scattering will be used for readout. a) Optical qubit readout. Photons are transferred from the $S_{1/2}$ to the $P_{1/2}$ levels by a $\lambda = 397\,\text{nm}$ laser. Another $\lambda = 866\,\text{nm}$ laser repumps the population decaying into the $D_{3/2}$ levels. b) Zeeman qubit readout. The population of the $m_J = +1/2$ state is shelved into the $D_{5/2}$ levels using a $\lambda = 729\,\text{nm}$ laser, followed by an optical qubit readout. Figure taken from [11].

The optical qubit can be read using a $\lambda = 397\,\text{nm}$ laser driving the cycling $S_{1/2} \rightarrow P_{1/2}$ transition along with a 866 nm repump laser saturating the $D_{3/2} \rightarrow P_{1/2}$ transition. Thus, ideally only the qubits in the bright states, i.e. the $S_{1/2}$ ground state, will fluoresce while the ions in the $D_{5/2}$ state will stay dark. However, the results will always contain a background signal from the 397 nm laser. To correct our measurements for this, we first do a cycle of optical qubit readout (containing the background) and then another cycle without applying the repump laser. The latter will then populate the $D_{3/2}$ levels, making the ion dark. Thus, we only measure the background counts which we use to get the background-corrected counts from the first cycle.

The Zeeman qubit, again, cannot simply be read out by driving a cycling transition. Rather, we have to first transfer the dark-state population in the $|S_{1/2}, m_J = +1/2\rangle$ state to the $D_{5/2}$ level with a couple of $\lambda = 729\,\text{nm}$ shelving pulses. After that, we can use an optical qubit readout as explained above.

After the readout, we perform simple photon-count thresholding for state inference: we detect the number of photons emitted from the ion and compare it to a threshold value to decide whether the qubit was in the dark or the bright state. The threshold value is determined by preparing and measuring a qubit in both $|0\rangle$ and $|1\rangle$ states and building a histogram from that. As the spontaneous emission follows a Poisson distribution, we can fit such a distribution to both the qubit states and get their means/variances. From those values we can calculate the optimal threshold[11, pp. 32-33].
Repeating the same experiment many times, we can use the thresholding technique to obtain probabilities $P(0)$ and $P(1)$ of the ions being in states $|0\rangle$ or $|1\rangle$ before the measurement, respectively. The probabilities are calculated using the sample mean

$$P(0) = \frac{n}{N} \tag{2.16}$$

where n is the number of experiments with outcome $|0\rangle$ and N is the number of repetitions. The uncertainty of those probabilities is known as the *quantum projection noise*, given by[11, p. 33]

$$\sigma_{P(0)} = \sqrt{\frac{P(0)(1-P(0))}{N}}, \tag{2.17}$$

which corresponds to the standard error of the mean of independent Bernoulli trials. This fails when the probability $P(0)$ is close to either zero or one as it underestimates the population variance for small N. *Laplace's rule of succession* solves this: it calculates the probability of success of the next trial based on previous successes and failures and gives a lower bound to the error. The formula that we use is given by

$$\sigma_{P(0),min} = \frac{1}{N+2}. \tag{2.18}$$

2.3 Cooling Schemes

There are many schemes that can be used to cool the ions. Here, we want to look at Doppler cooling and electromagnetically induced transparency (EIT) cooling as we use them in our experimental setup. But before looking at them we will first see how we can model the "heating" of ions and what we can do to reduce it.

In eq. 2.10 we only looked at the motional ground state of the ions. However, to account for the heating of ions (which is just their vibration) we have to add another term that couples their motional state to the external electric field. A good and simple model does this by assuming that the ion moves in a three-dimensional harmonic potential where the laser field can induce transitions that change the motional state with a coupling strength given by the *Lamb-Dicke parameter* η.

For our purposes, we assume $\eta \ll 1$, meaning that the extension of the wavefunction of the ion is much smaller than the laser wavelength λ. In this regime, which is called the *Lamb-Dicke regime*,

the interaction Hamiltonian is given by

$$H_I(t) = \hbar\frac{\Omega}{2}(\sigma_+ e^{i(\delta t+\phi)} + \sigma_- e^{-i(\delta t+\phi)}) \tag{2.19}$$

$$+ i\hbar\frac{\Omega\eta}{2}(\hat{a}\sigma_+ e^{i[(\delta-\omega_m)t+\phi)]} + \hat{a}^\dagger\sigma_- e^{-i[(\delta-\omega_m)t+\phi)]}) \tag{2.20}$$

$$+ i\hbar\frac{\Omega\eta}{2}(\hat{a}^\dagger\sigma_+ e^{i[(\delta+\omega_m)t+\phi)]} + \hat{a}\sigma_- e^{-i[(\delta+\omega_m)t+\phi)]}) \tag{2.21}$$

where ω_m is the motional frequency and $\hat{a}, \hat{a}^\dagger$ are the ladder operators of the harmonic oscillator, and Ω is the interaction strength. The first term is called the *carrier term* (see eq. 2.10) which for $\delta = 0$ corresponds to the resonant coupling between the electric field and the internal states of the ion. The second and third terms are called *blue sideband (BSB)* and *red sideband (RSB)* for $\delta = +\omega_m$ and $\delta = -\omega_m$, respectively, and correspond to the coupling of the electric field with the motional states of the ion. The blue sideband drives transitions of the form $|0\rangle|n\rangle \leftrightarrow |1\rangle|n+1\rangle$ with a Rabi frequency of $\Omega_{n,n+1} = \eta\Omega\sqrt{n+1}$ and effectively adds motional quanta. The red sideband removes motional quanta by driving transitions of the form $|0\rangle|n\rangle \leftrightarrow |1\rangle|n-1\rangle$ with a Rabi frequency $\Omega_{n,n-1} = \eta\Omega\sqrt{n}$. Thus, by choosing the right detuning we can use our lasers to cool the ions in our traps. For a detailed derivation of the theory, we refer to [11, 18].

2.3.1 Doppler Cooling

Doppler cooling is the first step we use to reduce the temperature of the ion. We want to use a dipole transition that has a high scattering rate for faster cooling, however, we cannot directly drive a RSB from the ion as the transition linewidth Γ is usually broader than the motional frequency. We are rather driving carrier, RSB and BSB transitions at the same time. However, with a detuning $\delta < 0$ we make the RSB excitation more likely than the BSB excitation and thus on average cool down the ion. Classically, the atoms counter-propagating the cooling beam see a higher frequency due to the Doppler shift and emit photons with higher energy than the photons they absorbed. The temperature limit is thus dominated by single-photon scattering[11].

We perform Doppler cooling by illuminating the ion with red-detuned 397 nm light and a laser at 866 nm to repump the population in the $D_{3/2}$ states. The hotter the ions, the more should the cooling laser be detuned. We usually first apply a far-detuned Doppler cooling sequence to the ions to cool them down from high-energy states before applying a near-detuned cooling sequence to also remove the lower motional quanta.

2.3.2 EIT Cooling

In addition to Doppler cooling, we can use EIT cooling, which exploits the principle of coherent population trapping, for reducing the motional state of the qubits below the Doppler cooling limit. It requires a three-level Λ system consisting of two ground states $|e\rangle, |f\rangle$ and a short-living excited state $|e\rangle$. Both ground states are driven with detunings δ_1 and δ_2, respectively. When choosing $\delta_1 = \delta_2$, the population is transferred into the dark state of the system which is a superposition of the two ground states. In this state, the system does not scatter photons anymore as the drives destructively interfere with each other[19, 11].

EIT cooling utilises this phenomenon in the following way: one of the ground states is strongly pumped with a detuning δ_1 while the other is weakly probed with detuning δ_2. The detunings are carefully chosen such that with $\delta_1 = \delta_2$ it suppresses the carrier transition while the RSB of the probe beam will be driven and the BSB is only driven weakly. It can cool multiple modes at the same time and thus cool the ion down to the motional ground state[11].

We use both circularly and linearly polarised $\lambda = 397\,\text{nm}$ lasers in addition to an 866 nm repump laser to perform EIT cooling. The strong circularly polarised *pump* beam off-resonantly drives the $\left|S_{1/2}, m_J = +1/2\right\rangle \rightarrow \left|P_{1/2}, m_J = -1/2\right\rangle$ transition with a detuning δ_1 while the weak linearly polarised *probe* beam drives the $\left|S_{1/2}, m_J = -1/2\right\rangle \rightarrow \left|P_{1/2}, m_J = -1/2\right\rangle$ transition with detuning δ_2 and the 866 nm laser repumps the $D_{3/2}$ states. When the detunings match each other $\delta_1 = \delta_2$ the ions will be cooled along the wavevector difference of the 397 nm lasers.

2.4 Ramsey Experiment

To probe the coherence of qubits we can perform Ramsey experiments. The simplest form is shown in Fig. 2.4. This scheme starts with a qubit in the excited state $|1\rangle$ and transforms it into an equal superposition of ground and excited state with a $\pi/2$ pulse $\hat{R}\left(\frac{\pi}{2}\right)$. During a time τ the qubit evolves freely and acquires an additional phase φ that we probe with a second $\pi/2$ pulse with variable phase ϕ followed by a readout of the qubit state.

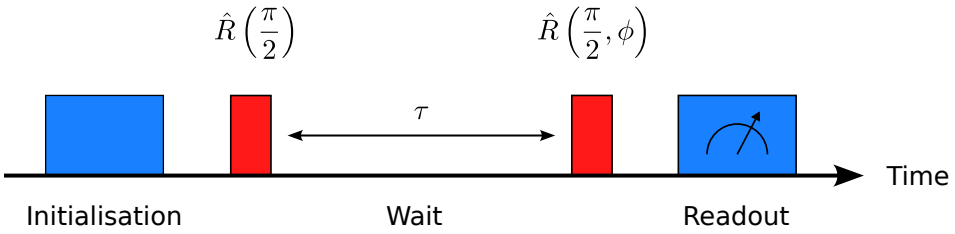


Figure 2.4: Scheme for a Ramsey sequence. The qubit is initialised in the $|1\rangle$ state and brought into an equal superposition of $|0\rangle$ and $|1\rangle$ by a $\pi/2$ pulse. After a free-evolution of length τ another $\pi/2$ pulse with variable phase ϕ is applied and the qubit is read out. By varying ϕ, the coherence of the qubit can be measured.

The probability of finding the qubit in the excited state after the Ramsey sequence is given by a sinusoidal oscillation[11, 20]

$$P(1) = \frac{1 - \cos(\varphi(\tau) - \phi)}{2}, \tag{2.22}$$

where $\varphi(\tau) = \int_0^\tau \delta(t)dt$ is the phase due to the detuning δ of the laser from the qubit frequency. Thus, we can use the Ramsey sequence to perform a frequency calibration of the $|0\rangle \leftrightarrow |1\rangle$ transition with the condition $\varphi = 0$.
However, as the time τ gets longer, the amplitude of the oscillation decays due to phase noise and the probability takes the form[20]

$$P(1) = \frac{1 - C(\tau)\cos(\varphi(\tau) - \phi)}{2}. \tag{2.23}$$

Another Ramsey scheme which reduces part of the low frequency noise is the spin-echo scheme

(see Fig. 2.5). It introduces an additional π pulse between the ground state and the excited state after half of the wait time.

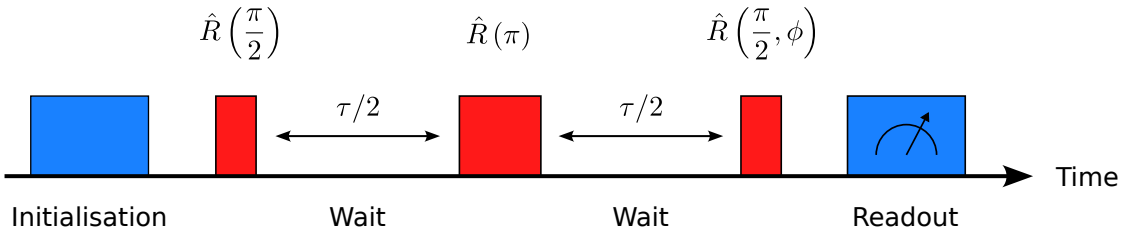


Figure 2.5: Scheme for a spin-echo sequence. It modifies the Ramsey scheme by introducing an additional π-pulse after half the wait time τ.

The probability of finding the qubit in the excited state after the sequence is the same as in the Ramsey sequence, shifted by π. The only thing that changes is the additional phase due to the detuning of the laser from the qubit frequency $\tilde{\varphi}(\tau) = \int_0^{\tau/2} \delta(t)dt - \int_{\tau/2}^{\tau} \delta(t)dt$. Due to this, we expect an echo signal $\tau/2$ after the π pulse, which gives the scheme its name[11].

2.4.1 Phase Noise

The decay of the oscillation amplitude can have different reasons. In this thesis, we to look at two simple models for the phase noise: the *high-bandwidth* or *white noise* and the *low-bandwidth* noise.

The white noise can be treated as having a flat power spectral density and yields a contrast loss following an exponential decay[12, 11]

$$C_{\text{white}}(\tau) = e^{-\tau/T_2}, \tag{2.24}$$

with a decay constant commonly referred to as the T_2 *time*.

The low-bandwidth noise, on the other hand, can be treated as frequency components centred around ω_0 as this noise is very slow and has long correlation times. The Ramsey contrast follows a Gaussian decay[11]

$$C_{\text{slow}}(\tau) = e^{-(\tau/T)^2}, \tag{2.25}$$

where T is the slow noise decay constant. As this noise is so slow, it is partially suppressed by the spin-echo sequence which inverts the direction the phase decoheres into through the π-pulse.

2.4.2 Phase-insensitive Ramsey Scheme

We perform the Ramsey experiments with our 729 nm laser coupling the $\left|S_{1/2}, m_J = -1/2\right\rangle \leftrightarrow \left|D_{5/2}, m_J = -1/2\right\rangle$ transition which is limited by the phase noise of the laser. To circumvent that we can perform the Ramsey sequence with a scheme that is insensitive to the phase of the laser[21, ch. 8.3]. For this, we exploit the hybrid operation on Zeeman and optical qubits as demonstrated in [11, 21]. Operation and manipulation is performed on the optical qubits $|0\rangle_o, |1\rangle_o$ while free-space evolution takes place in the Zeeman qubit $|0\rangle_z, |1\rangle_z$. We can do this by mapping the $|0\rangle_o$ state on the $|0\rangle_z$ state with a π pulse of our 729 nm laser.

With this scheme, we can potentially achieve higher coherence times. This is because the phase shift of a laser only matters when we are in a superposition of that transition. When we are in the Zeeman qubit, any phase of the π pulse will bring us up to the $|0\rangle_o$ state and we won't acquire a phase shift as long as the operations on the optical qubit are coherent.

Chapter 3

Motivation for pycrystal

In this chapter I want to explain the motivations for developing `pycrystal`. To that end, I will give a short overview of the control system in section 3.1 and explain how experiments and experiment sequences can be created and passed to the control system in section 3.2. I will highlight the advantages and disadvantages of the different approaches and point out where `pycrystal` comes in.

3.1 Control System

The control system of the TIQI (Trapped Ion Quantum Information) group (see Fig. 3.1) is called M-ACTION (which stands for *Modular Advanced Control of Trapped IONs*). It is based on a hybrid CPU (central processing unit) and FPGA (field-programmable gate array) centric architecture separating the time-coherent operations on the FPGA from the main application on the CPU. This application, called `ionpulse`, is written in C++ and is executed "bare-metal" (without an operating system) on the CPU. Currently the *Zedboard*[1] is used as the main controller, where the FPGA and CPU meet on a single chip. A PC connects to the Zedboard via USB to debug and program it with the main application, and via Ethernet to configure it in a graphical user interface (GUI) called *Ionizer*.

The Zedboard has multiple digital outputs, often referred to as TTL (transistor-transistor logic), and is connected to DDS (direct digital synthesis) cards which generate RF signals. The digital outputs are used to trigger asynchronous devices during experimental sequences and from the GUI. These can be, for example, shutters, readout devices like photomultiplier tubes (PMTs) or arbitrary waveform generators (AWGs) like the Fastino controller, handling the electrode voltages of a trap. The RF outputs from the DDS cards go to acousto-optic modulators (AOMs) used to control the laser pulses and to shift the frequency of the beams.

There are also some other standalone devices like lasers, cavity locks, shim or cavity DACs (digital-to-analogue converters) or fixed RF sources that are not controlled directly by the M-ACTION system. The control PC uses `tiqi-plugin` to control those devices via raspberry pis.

For a detailed description of the control system we refer to [23, 24].

[1] Commercial Zynq evaluation board by Digilent using a Xilinx XC7Z020-CLG484-1 chip[22]

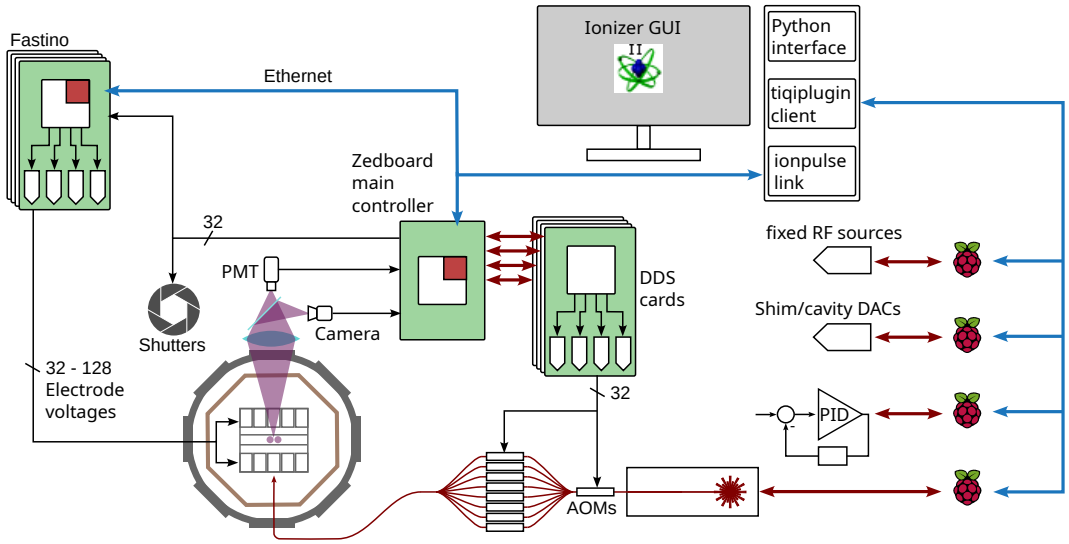


Figure 3.1: A schematic of the M-ACTION control system, along with a control PC and raspberry pi devices controlling asynchronous, high-latency devices. The green-coloured areas show PCBs equipped with an FPGA (white boxes). The white-red box combination depicts a chip with combined CPU and FPGA. The control PC is connected to the main controller, the Fastino controller and a couple of raspberry pis via Ethernet (blue lines). The Zedboard communicates with the DDS cards via the backplane (red arrows) using a custom, low-level interface and handles digital in- and outputs (black lines). Image created by Martin Stadler.

3.2 Creating and Running Experiments

At the moment, there are two ways of executing pulse sequences through our control system, as depicted in Fig. 3.2. In the original implementation, experiments are written in C++ using the `ionpulse SDK` and compiled and linked into the `ionpulse` executable on the control computer. It is uploaded onto the Zedboard's CPU, which executes the experiment on demand, in which case it passes instructions and parameters to the FPGA. The FPGA executes the sequence concerning digital in- and outputs sequentially and forwards instructions for the RF generation to the DDS cards. The experiments are arranged in "pages" on the Zedboard. They can be triggered and configured from Ionizer on the control computer. This allows the user to change "remote parameters" through the GUI which the experiments reference in their sequences. This can, for example, be the number of shots (which is the number of repetitions to average over for a single data point), or other parameters such as the amplitudes, frequencies or phases of the AOMs. They can be updated while the experiment is still running which makes experiments more flexible. The results of each data point of an experiment or scan get sent back to the control computer where Ionizer plots it in a separate window.

Recent work has been done which provides an alternative way of creating pulse sequences. The new framework makes use of a specific experiment already compiled within the `ionpulse SDK`. It accepts as a remote parameter JSON strings, a file format storing data objects in terms of attribute-value pairs and arrays. Those strings encode the sequences we want to run on the control system which uses a custom *JSON parser*, created by Martin Stadler and further developed by

Alexander Ferk, to create the corresponding experimental sequence on the control system. This work is complemented by the `ionpulse sequence generator` python library created by Marco Stucki[25] and further developed by Martin Stadler, Alexander Ferk[26] and me which can be used to create a JSON-encoded pulse sequence. The library defines classes that are directly mapped to instructions for the RF and TTL signals processed by the control system (see section 4.1).
The workflow of creating an experiment is to write a sequence with the `ionpulse_sequence_generator` consisting of RF and TTL events which is then represented as a JSON string. Upon sending it to the Zedboard, the JSON parser translates it into hardware instructions which are then executed. The result of the experiment is returned from the Zedboard to the control PC afterwards.

3.2.1 Evaluation

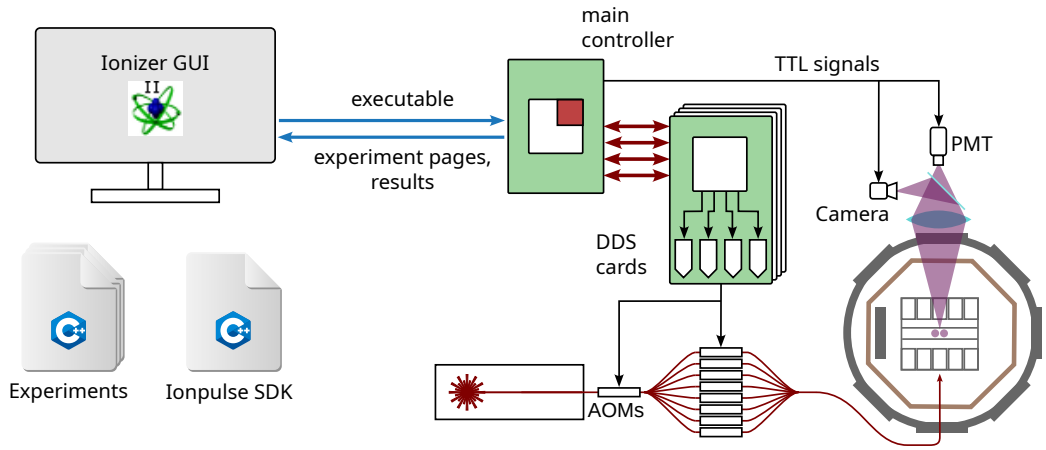
There are advantages and disadvantages to both approaches of creating and running experiments on the control system. I now want to motivate the reason behind the use of the second method and describe how `pycrystal` comes in.

An advantage of writing the experiments in C++ is that we do not need to know the exact values for the parameters when compiling them as they can be configured from Ionizer. However, C++ is not a beginner-friendly language and as such not accessible to all users. Furthermore, adding new pulse sequences requires writing new experiments and to recompile them together with the `ionpulse SDK` which can take up to a couple of minutes. Interfacing with high-level libraries for creating pulse sequences, like Qiskit[2] or pyGSTi[3], is not possible at the moment. This is because there is no API to translate the code from those libraries into classes and methods that can be used in C++, and neither is there a straightforward way to implement this. This is not desirable, especially when you want to provide an easy-to-use interface for third parties. Another conceptual consideration is that the Zedboard should only run hardware operations, i.e. trigger TTL and generate DDS card signals. It does not have to know about all the physics behind the scenes, i.e. knowing about abstract pulses and qubit gates, generating waveforms to move the ion or even translating high-level code into hardware operations.

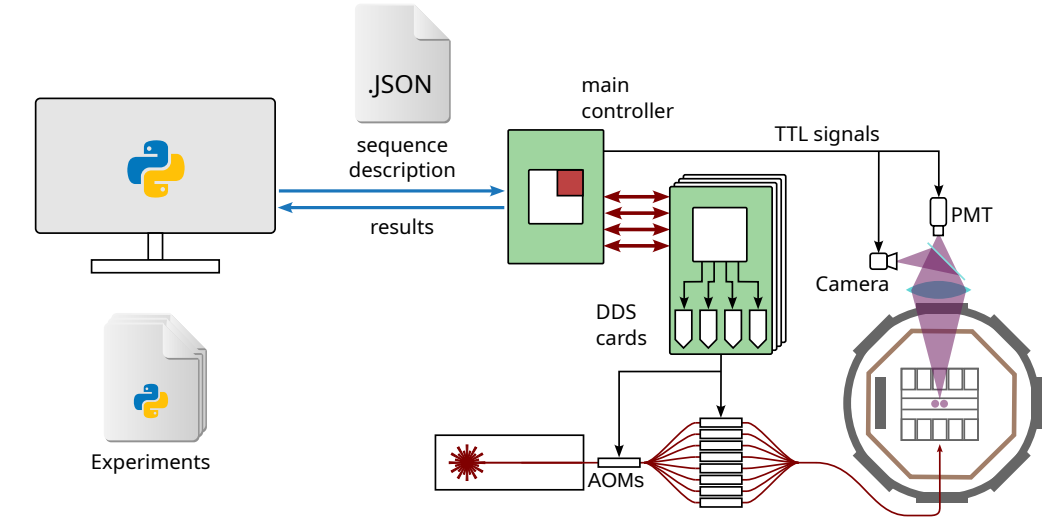
A python program that uses the `ionpulse sequence generator` has to know the values for all parameters before translating the sequence into a JSON string. This means that we have to generate new JSON strings to change the parameters of a pulse sequence. However, it is unrestricted in defining the sequences that will be run on the control system and does not require to recompile the software of the main controller. Following the conceptual considerations mentioned above, it separates the hardware instructions from the high-level pulse sequence description by providing an interface to low-level instructions that the Zedboard understands. The pulse sequences are generated on the control PC rather than stored on the Zedboard. Thus, it is easier to integrate the library with asynchronous devices, like Fastinos, and third party packages that provide a high-level API to create pulse sequences as communication with those devices can happen at the same layer as the pulse sequence generation, which was not the case before. This allows to keep the code for an experiment well-structured and easier to understand and maintain.

[2] "Qiskit is an open-source SDK for working with quantum computers at the level of pulses, circuits, and application modules."[27]

[3] "pyGSTi is an open-source software for modeling and characterizing noisy quantum information processors (QIPs), i.e., systems of one or more qubits."[28]



(a)



(b)

Figure 3.2: There are two ways of creating and running experiments on our control system. (a) The experiments are written in C++. They are compiled and linked into ionpulse to create an executable which is written onto the CPU of the main controller. Experiments are triggered and controlled from a GUI (called *Ionizer*) on the control PC. (b) Experiments are written in python and translated into a JSON string. The software for the control system is already deployed and provides an interface for those strings. New pulse sequences require the generation of the sequence description. Those can directly be passed to the executable on the main controller which will return the results of the experiment. Adapted from image created by Martin Stadler, idea: Alexander Ferk, Mose Müller.

While the ionpulse sequence generator exposes the low-level classes and makes use of the JSON interface to the Zedboard, it is left to the user to build pulse sequences with meaningful physics. This is where `pycrystal` comes in: it serves as a high-level python interface for creating pulse sequences following a bottom-up approach. The users can define their hardware, implement an abstract representation of the trapped ions and directly write sequences that request pulses on them. Chapter 4 will elaborate on the requirements of the package and the design choices I made for it.

It is also worth mentioning other work conducted with the JSON interface of the control system: complementary to the bottom-up approach, a top-down approach was used to directly use Qiskit and its custom JSON encoding of pulse sequences to define abstract operations at the gate level and translate them into a format that could be parsed by `ionpulse`[29, 25]. This work, however, turned out to be not as flexible in the representation of more generic experimental setups.

Chapter 4

pycrystal

This chapter now introduces `pycrystal`, the library that I have been working on for the biggest part of my thesis work. Its main goal is to unify the following of requirements into one framework. It should

1. use the `ionpulse sequence generator` to create the JSON strings that can directly be executed. This should allow for flexibility in creating different pulse sequences.
2. be able to model different traps and architectures like QCCD or long chains with single-ion addressing. In view of the scalability of those architectures, the library should also transparently implement simultaneous operations on multiple ions. Testing the package in the Cryo setup of TIQI, which I will describe in Chapter 5, allowed me to develop the package with a special attention to this feature.
3. be able to model different configurations of ion species and their isotopes. This and the last point also requires that the framework basis remains very generic, leaving the users to implement most of the specific details themselves.
4. connect the generation of experiments to a database storing the values of the experimental parameters and update them after performing calibration scans.
5. allow for re-usability of parameters in form of `Parameter` class instances. We want to use the same parameter for all the variables with the same physical meaning instead of having a parameter instance for each possible variable. For example, we should use the same parameter to query the $\pi/2$ time for both the $\left|S_{1/2}, m = -1/2\right\rangle \rightarrow \left|D_{5/2}, m = +1/2\right\rangle$ and $\left|S_{1/2}, m = -1/2\right\rangle \rightarrow \left|D_{5/2}, m = +3/2\right\rangle$ transitions and only change the arguments we pass to it.
6. provide a module for quickly defining and scheduling experiments on the Zedboard and executing parameter scans. Although this is useful for seeing `pycrystal` in action, a sequence scheduler is not intended to be part of the library, and this functionality will be later implemented somewhere else.

I will first go through the `ionpulse sequence generator` API in section 4.1 and explain what functionality it offers before presenting the design decisions of pycrystal in section 4.2.

4.1 Ionpulse Sequence Generator

The `ionpulse sequence generator` is a python API to create pulse sequences that can be executed on the `ionpulse` application running on the M-ACTION system. It uses the JSON format to communicate information to the control system (which was originally inspired by the JSON format of Qiskit's `Schedule`)[25]. The control system parses the JSON string to create the experimental sequence that will be run.

The API provides a couple of classes which we can use to create pulse sequences:

- *RF events* (`RFEdge`, `RFWait`, `RFPulse`) are used to abstract fundamental pulse objects acting on a single RF channel. `RFEdge`s describe single rising/falling edges with a wait time of specified length before the edge while `RFWait`s describe a time span during which no instruction will be executed. `RFPulse`s are putting two of those edges together and correspond to a rising and falling edge separated by a wait time of a specified length.

- `TtlEdge`s are handling the *digital I/O signals*. They change the global state of all TTL channels and can be used to synchronously trigger devices (e.g. for transport or readout events) during an experiment sequence.

- `Readout` events register a read-out on the Zedboard's CPU.

- *Qubit events* (`QubitEdge`, `QubitWait`) are used to keep track of the qubit phases to enable coherent operations when taking AC-Stark Shifts into accounts[30]. They work the same way as RF events, but set frequency and phase for the phase accumulator tracking a qubit phase (see [26]).

- *Sequences* (`LinearSequence`, `Loop`, `Fork`, `Path`) are more complex objects making up the experimental sequences. They contain subsequences for RF-, qubit- and readout events and digital I/O signals. `LinearSequence`s describe general containers for pulse and sequence objects, `Loop`s describe loop instructions and `Fork`s describe forked instructions. They take multiple `Path`s (which are just linear sequences themselves) and a condition upon which the following path is decided, allowing to implement experiments with real-time decisions and feedback[31].

The JSON generator implements an *event-based approach* analogous to the control system (as opposed to Qiskits timetable-based approach). That is, each edge, wait instruction and sequence is an event with no fixed starting time. When added to a sequence, they start directly after the last event on the same channel has finished. The time argument passed to the events is the wait time before the new event will take place. If sequences act on multiple channels the instructions will start at the same time on all channels, which is enforced by the generator.

4.2 Design

In this section, I want to show how I approached fulfilling the requirements of pycrystal. The library follows a bottom-up approach to modelling the setups in our labs while abstracting the

parts in an object-oriented way. It consists of four modules with separate functionalities. The `pycrystal.beamline` module (section 4.2.1) models the hardware of the setup and revisits requirements 1. and 2. The `pycrystal.crystal` module (section 4.2.2) deals with the 3. requirement by abstracting the ion crystals in the trap. Points 4. and 5. are fulfilled in the `pycrystal.parameters` module (section 4.2.3) which connects the physical parameters to the parameters in a database. Finally, the last requirement is implemented by the `pycrystal.experiment` module (section 4.2.4) which defines an abstract `Experiment` class. This class is used to build pulse sequences that can then be run on the control system and is used to perform scans on parameters.

4.2.1 `pycrystal.beamline` - Defining the Hardware

This module creates an abstract representation of the lab hardware used to control laser pulses, defining all the beamline objects like AOMs or readout devices like PMTs or cameras, and builds objects that can be used to create pulses from lasers or trigger readout events.

For this, I have abstracted the AOMs and PMTs in an object-oriented way as can be seen in Lst. 4.1. The class instances store the defining attributes, such as the DDS or TTL channels they respond to, or the AOM type, central frequency or order of the AOM, within the instance.

```
1    aom_729_sp_3 = AOM(
2        "729_sp_3",
3        type=AOMTypes.SP,
4        dds_channels=[13, 14],
5        central_frequency=150,
6        order=1,
7        endpoints=[Zones.ZONE_3],
8    )
```

Listing 4.1: Examples of instantiation of a single-pass AOM pointing to zone 3. The initialiser takes multiple arguments that will be stored in the instance and is used to internally address the object. The `type` specifies whether the AOM is a single-pass or a double-pass, the DDS channels are connected to the AOM to drive it in either single-tone (first channel) or multi-tone (both channels) mode. The `central_frequency` is the central frequency specified by the manufacturer while `order` is the Bragg diffraction order that is physically chosen through the setup. The `endpoints` argument takes a list of arbitrary specifiers that can be used to specify where this AOM points to. This can, for example, be used for single-ion addressing.

I decided to create three main classes which use the optical objects to build an abstraction of the experimental setup (see Fig. 4.1): the `BeamLineLayout`, the `BeamLineParameterisation` and the `BeamLine`. Those classes separate different functionalities needed to build a working abstraction.

The `BeamLineLayout` class abstracts the path the beam takes from the laser through optical devices to some endpoints on the trap (e.g. an AOM, a zone, a single ion, ...). It uses a tree data structure to store the beamline objects it consists of and their position in the layout (see Lst. 4.2). The base class of the tree is `BeamLineNode` which is based on anytree's `NodeMixin` class[1]. The nodes of the

[1] It adapts some methods for better integration with pycrystal (like type annotations, instance checking, printing warnings, raising errors, ...)

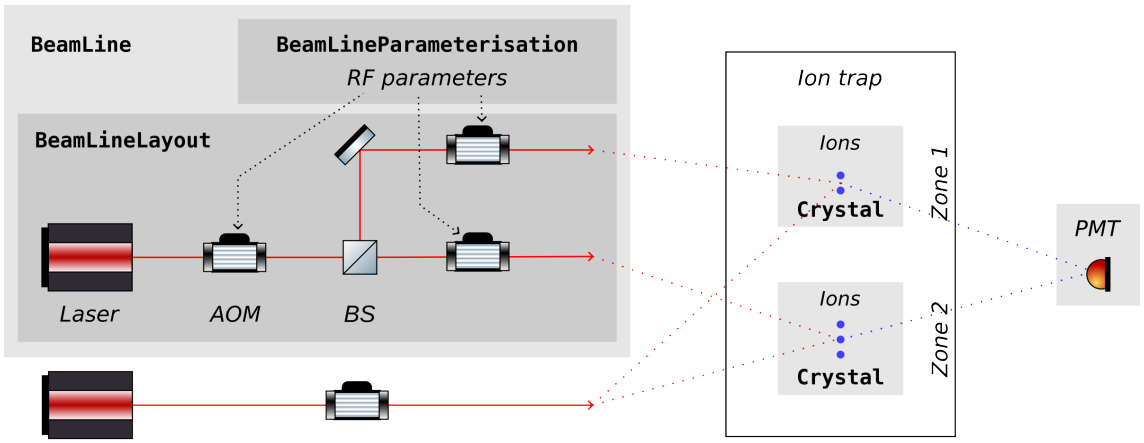


Figure 4.1: A simple setup with lasers, AOMs, ions on a trap with different zones and a PMT. The italic text gives the name of the physical entities or values, the red dotted lines indicate where the laser beams are going to the trap, the blue dotted lines show where the PMT collects photons from and the black dotted lines indicate the RF parameters that have to be supplied to the AOMs. The colour-coded regions highlight the layers of abstraction of the `pycrystal` library. The bold text gives the name of the objects in the code base abstracting the physical entities or values.

layout tree are the beamline layout itself (which is the root) and the beamline objects (like AOMs or acousto-optic deflectors (AODs)).

```
1    layout_729 = BeamLineLayout(
2        "729 Laser",
3        children=[
4            aom_729_dp([aom_729_sp_1, aom_729_sp_2, aom_729_sp_3]),
5        ],
6    )
```

Listing 4.2: Example of a `BeamLineLayout` initialisation for a 729 nm laser with a double pass AOM followed by three single pass AOMs going to three different zones. The `BeamLineLayout` is the root of the tree structure and the `children` argument takes a list of AOMs that will be the branches of the parent node. This happens recursively for each AOM. The AOMs that do not take any children are called the *leaves* of the `BeamLineLayout`.

The layout itself does not know the parameters that the beamline objects receive, e.g. frequency, phase and amplitude shifts of the laser beam passing through an AOM together with the duration the AOM will be turned on during a specified pulse. This is done by the `BeamLineParameterisation`. A beamline parameterisation is a container defining the values of the attributes of all the beamline objects of a layout. The parameterisation can depend on the input arguments like *pulse type* or *endpoints*. In calcium, for example, we can parameterise the 866 nm laser for EIT- and Doppler cooling, sigma-spin-polarisation and detection pulses with the same values. The only thing that changes is how long the AOM is turned on (see Lst. 4.3). The 729 nm laser, on the other hand, could have different parameterisations depending on the pulse type (e.g. sideband or carrier pulses)

and transition. To parameterise a multi-tone AOM one has to pass a list of two parameters or values where the first value will be applied to the first DDS channel and the second value to the second channel.

```
class Parameterisation866(BeamLineParameterisation):
    """This class acts as a container for parameterising the 866 laser."""
    layout = BeamLineLayout("866 Laser", children=[aom_866])

    @classmethod
    def params(
        cls, pulse_type: PulseTypes, **kwargs: Any
    ) -> Tuple[Dict[BeamLineObject, Dict[str, Any]], float]:
        params: Dict[BeamLineObject, Dict[str, Any]] = cls._params_init()

        params[aom_866]["frequency"] = ... # frequency parameter
        params[aom_866]["amplitude"] = ... # amplitude parameter
        params[aom_866]["phase"] = ... # phase parameter

        if pulse_type == PulseTypes.EIT:
            motional_identifier = kwargs.pop(
                "motional_identifier", MotionalIdentifiers.COM
            )
            pulse_time = ... # time parameter using motional_identifier var
        elif pulse_type in {PulseTypes.DOPPLER_FAR, PulseTypes.DOPPLER_MID}:
            pulse_time = ... # time parameter
        elif pulse_type == PulseTypes.SIGMA_SPIN_POL:
            pulse_time = ... # time parameter
        elif pulse_type == PulseTypes.DETECTION:
            pulse_time = ... # time parameter
        else:
            raise Exception(f"Pulse type {pulse_type} has not been parameterised yet!")
        return params, pulse_time
```

Listing 4.3: `BeamLineParameterisation` example. The `Parameterisation866` parameterises an 866 nm laser with one AOM for EIT- and Doppler cooling, sigma-spin-polarisation and detection. The class derives from `BeamLineParameterisation` and implements the `layout` attribute and the `params` class method. The latter returns a dictionary containing the parameterisation of the beamline objects and the pulse duration in dependence on the input arguments like *pulse type* or *endpoints*. The dictionary contains another dictionary for each beamline object of the `layout` assigning values to all attributes of that object. The `layout` defines the BeamLineLayout you want to parameterise and is used internally to initialise the dictionary with `cls._params_init()`. One can use the arguments passed to the `params` method to set the values for the variables as done with the `motional_identifier`.

The `BeamLine` class connects both `BeamLineLayout` and `BeamLineParameterisation` and builds the interface between `pycrystal` and the `ionpulse sequence generator`. Beamline instances

are the central object to create using the `pycrystal.beamline` module as they are essential for creating crystal classes in the `pycrystal.crystal` module. They represent entities that can request pulse sequences from specific lasers through the `request_pulse` method which creates a `LinearSequence`, as demonstrated in Lst. 4.4. This allows us to create arbitrary pulse sequences by adding the requested pulses together, either sequentially or in parallel.
Furthermore, support for different architectures like QCCD is implemented using the `endpoints` specifier. When defining an `AOM`, the *endpoints* keyword argument takes a list with entries of arbitrary type, e.g. an enumeration. It is used to specify the physical locations or objects in the setup the beamline object points at. The endpoints can then be referenced in a requested pulse through either a beamline or a crystal to drive the required AOMs only. One can thus request pulses from beamlines on arbitrary endpoints, as long as they are defined on those objects that are leaves of a beamline layout (setting it for the other objects will not have any effect). This allows, for example, the implementation of single-ion or -zone addressing.
Finally, the `phases` argument implements a way of setting the phases of the beamline's AOMs directly. The order of the specified phases goes from the root of the tree structure to the leaf. Thus, setting the phases of a beamline like the `beamline_729` in Lst 4.4, consisting of a double-pass AOM followed by a single-pass AOM, results in phase shifts of zero for the double-pass and 90 degrees for the single pass.

```
1    beamline_729 = BeamLine(layout_729, Parameterisation729)
2    sequence = beamline_729.request_pulse(
3        pulse_type=PulseTypes.PI,
4        endpoints=Zones.ZONE_1,
5        transition=Transitions.SM12_TO_DM12,
6        phases=[0, 90],
7    )
```

Listing 4.4: `BeamLine` example. The initialiser takes both the layout and the beamline parameterisation. It exposes the `request_pulse` method which takes a `pulse_type` keyword argument and arbitrary other keyword arguments. If `endpoints` are provided, e.g. a beamline object or a list of enumerations, they will be used as the output of the laser. If a list of floats is provided to the `phases` keyword argument, those phases are directly applied to the AOMs of the beamline. All arguments you pass to the pulse request will be passed down to the beamline parameterisation. Using the return values of the parameterisation, the beamline objects are initialised and the RF pulses are generated. The request returns a linear sequence containing information that can be passed to the control system to drive the selected beamline objects.

4.2.2 `pycrystal.crystal` - Abstracting Ion Crystals

Continuing with the bottom-up approach, this module was designed to build upon the `pycrystal.beamline` module. It implements a `Crystal` class whose instances abstract ion crystals in the trap (see Fig. 4.1), fulfilling the fourth requirement of the package.

I decided to first compose a `Crystal` class that meets the requirements for a $^{40}Ca^{+}$ ion as this is the ion we are trapping in the Cryo setup. The idea is that the crystal defines all the pulses we

can apply to the ion crystals as methods, e.g. `detection`, `doppler_cooling`, `pi`, `pi_half`, etc. The pulses themselves come from the beamlines that the user has to define for his setup, while the readout events have to be triggered using a user-defined PMT object. That means that the `Crystal` class has to get the information of which beamlines and PMT he can use to request pulses from. The PMT that can be used in the detection pulse is defined as the Crystal class attribute `readout_pmt` while the beamlines are stored in a dictionary which takes the beam type parameter as the key and the beamline as the value. Thus, knowing that we are manipulating a $^{40}Ca^{+}$ ion, we can already implement the pulses that are generic to the calcium ions. When the users now want to use the `Crystal` class, they inherit from the generic $^{40}Ca^{+}$ crystal class and overwrite the readout PMT and the beamline dictionary (see Lst. 4.5).

```
1  class CryoCrystal(Crystal):
2      """Implementation of a Cryo-specific calcium-40 Crystal class."""
3      beamlines = dict()
4      beamlines[BeamTypes.BEAM_397_SIGMA] = beamline_397_s
5      beamlines[BeamTypes.BEAM_397_PI] = beamline_397_p
6      beamlines[BeamTypes.BEAM_729] = beamline_729
7      beamlines[BeamTypes.BEAM_854] = beamline_854
8      beamlines[BeamTypes.BEAM_866] = beamline_866
9      readout_pmt = pmt
10
11     def individual_pulse(
12         self, endpoints=None, phases=None, **kwargs: Any
13     ) -> LinearSequence:
14         ...
```

Listing 4.5: `Crystal` class example implementing the crystal class for the Cryo setup. The class derives from the generic `Crystal` class and defines the beamlines and the readout PMT the class can use to create the pulse sequences. The beamlines are stored in a dictionary taking the beam type as key and the `BeamLine` instance as value. The class also implements an individual pulse that is not part of the generic class.

This approach also allows for multiple species and other isotopes. The abstract `Crystal` class implements the pulses as abstract methods which can be overwritten for each atom species or isotope. As the abstract pulses are very specific to the different ions used this is not really an overhead for the user as they have to define the pulses either way if they are not yet present.

To make the crystal instances really model the ion crystals, they should also keep track of the ion's position. This is not yet implemented, but the idea is that one can initialise an ion crystal with a specified number of ions at a specified endpoint. When performing a transport operation on it, the endpoint should get updated. When the ion crystal is split, it should create another crystal instance and update the number of ions and their positions.
Another feature from the `ionpulse_sequence_generator` that is not made use of yet is the phase tracker through the qubit events.

4.2.3 `pycrystal.parameters` - Interface between Database and Experimental Parameters

The `pycrystal.parameters` module was designed to provide an interface between a system storing values relevant to our experiments (from now on called *database*) and `pycrystal`. The module contains classes that provide an interface to query and set values in a database (database classes) and classes that abstract sets of values in those databases which can be used in the beamline parameterisations (parameter classes). The parameter class instances (from now on called *parameters*) interface with a database that is set by the user and are designed to be reusable, i.e. to parameterise multiple variables with the same physical meaning. I will first explain my design decisions for the databases and how we can set them for the parameters before going into the reusability aspect of the parameters and how we can scan those parameters in experiments.

The first database class I wrote was the `ZedboardDB`. This database takes the hostname and port of the Zedboard to connect to it. With it, we can query and set the values for the *remote parameters* that are defined in the `ionpulse SDK`. As we are interfacing the values in the SDK, we can also use Ionizer to update them. This was key to our approach as we can use Ionizer and the `ionpulse SDK` to scan the parameters and use them in `pycrystal` afterwards. Furthermore, it allowed us to be flexible in the transition and testing phase in between the two different types of control systems as both `pycrystal` and Ionizer can access the same set of parameters and can thus share experiments. For this to work, we have to provide the hostname of the computer Ionizer is running on and the port it is exposing.
To access the variables with `pycrystal`, I had to implement a naming convention for the Cryo setup, i.e. declare how the name of the remote parameter in the SDK reflects the information about it. For example, the remote parameters always start with the parameter type, i.e. "f" for frequency, "a" for amplitude, "p" for phase and "t" for time, and could continue with the pulse type or transition. This is of course setup-specific, and different setups would either have to stick to the same convention or use a different database class that reflects their own.

The parameter classes call the database instance specified by the user (see Lst. 4.6), or the dummy database `DummyDB` otherwise, to query and set values through their instances. This makes it very easy to change the database. The only thing that new databases have to do is to overwrite the abstract methods of the abstract `Database` class.

```
1  # add this at the beginning of your config code
2  from pycrystal.parameters import Parameter
3  from pycrystal.database import ZedboardDB
4  Parameter.db = ZedboardDB(<optional_connection_details>)
```

Listing 4.6: One can set the database for the parameter classes by changing the `db` attribute of the `Parameter` class which is inherited by all derived classes. Here, we set it to the `ZedboardDB`. One can change its connection details by providing it with the hostname and port of the Zedboard and the hostname and port for the Ionizer host connection.

There are several parameter classes, beginning with the abstract `Parameter` class. The classes differ in their default values for some keyword arguments, e.g. the `FrequencyParameter` defaults the

`param_type` to "frequency", while `AmplitudeParameter`, `PhaseParameter` and `TimeParameter` default to "amplitude", "phase" and "time", respectively. Parameters can be specialised: for instance, there are several frequency parameter classes, each specifying a default value for the `frequency_type`, i.e. `BareFrequency`, `StarkShiftFrequency` and `MotionalFrequency` defaulting to the bare-, stark-shift- and motional frequency, respectively. They all implement a "frequency" parameter which shares the same logical function in different types of beams, e.g. addressing different qubit transitions.

The function of these classes is to reduce the overhead of specifying the same arguments every time a new parameter has to be created. Additionally, fixing the default values makes it easier to debug the code if something does not work. It also enabled me to write more precise warnings and error messages.

One can create a parameter by instantiating one of the parameter classes. By performing a call on it, you can retrieve the value it is parameterising from the database while you can set a value using the `set_value` method, taking the new value as the first argument and arbitrary keyword arguments. Both constructor and call also take additional keyword arguments that are passed to the database for querying and setting the corresponding values (see Lst. 4.7). The arguments passed to the call take prevalence over those passed to the constructor if a keyword argument overlaps. This is done by storing the arguments provided to the constructor as default values. But unlike the default values defined by the parameter classes themselves, they can be temporarily overwritten with a parameter call. (This behaviour can easily be changed to default values that cannot be overwritten).

```
1 >>> from pycrystal.parameters import BareCarrierFrequency
2 >>> from pycrystal.utils import PulseTypes, Transitions
3 >>> freq_full_init = BareCarrierFrequency(transition=Transitions.SM12_TO_DM12)
4 >>> freq_partial_init = BareCarrierFrequency()
5 >>> freq_full_init() == freq_partial_init(transition=Transitions.SM12_TO_DM12)
6 True
```

Listing 4.7: This example shows how we can define and use parameters. It instantiates the parameters `freq_full_init` and `freq_partial_init` which instantiate a bare carrier frequency parameter. The former parameter provides a default value for the transition while the latter does not. We can see, however, that the return values of calling `freq_full_init` without keyword arguments and `freq_partial_init` with the same transition as an argument are the same.

With the example in Lst. 4.7 we can already see how the "re-usability" of the parameters can be exploited. The parameter instance `freq_partial_init` parameterises the bare carrier frequency of pi pulses for different transitions. By just changing the keyword argument `transition` we can change the value queried from the database.

Another feature of the parameters is that they can be used in context managers (starting with a `with` statement), where it is possible to set the value for the parameter without interfacing and changing values in the database. When leaving the context, the value is reset (see Lst. 4.8). This way it is possible to perform scans on parameters, e.g. for calibrating the value of a physical variable.

```
>>> bare_freq = BareCarrierFrequency()
>>> bare_freq.set_value(2, transition=Transitions.SM12_TO_DM52)
>>> print(bare_freq(transition=Transitions.SM12_TO_DM52))
... with bare_freq as param:
...     param[
...         ("transition", Transitions.SM12_TO_DM52),
...     ] = 10
...     print(bare_freq(transition=Transitions.SM12_TO_DM52))
... print(bare_freq(transition=Transitions.SM12_TO_DM52))
2
10
2
```

Listing 4.8: This example shows in a python interpreter how we can easily change the value of a parameter for specific keyword arguments without changing the value in the database. We first instantiate a bare carrier frequency parameter and print out the value for the $|S_{1/2}, m = -1/2\rangle \rightarrow |D_{5/2}, m = -5/2\rangle$ transition that we just stored in the database. Then we change this value inside a context manager (which is really storing the value in a dictionary) and see that the parameter call gets assigned to that value. Printing the same parameter call after the context manager again yields the value stored in the database.

4.2.4 `pycrystal.experiment` - Building Experiment Pulse Sequences

This module allows us to easily create and run pulse sequences on the Zedboard and perform scans over variables, in a way similar to how it is done in Ionizer with the `ionpulse SDK`. I wrote it for testing purposes and for quickly debugging test sequences. The data and results presented in chapter 6 were taken using this module.

With the abstract `Experiment` class, this module provides a skeleton for experiments. It implements an abstract `schedule` method that should return the JSON string of the pulse sequence which is either run or used to perform scans on parameters. The intended usage of this class is to inherit from it to implement a specific experimental sequence, defined by overriding the `schedule` method as shown in Lst. 4.9. With that, one can now use the methods `run`, `scan_1D` and `scan_2D` on instances of this class to run this pulse sequence or perform parameter scans.

```
class CryoExperiment(Experiment):
    def schedule(self) -> str:
        crystal = CryoCrystal("Test exp")
        seq = LinearSequence("Test exp", auto_channel_mask=True)
        seq += crystal.default_cooling(eit_cooling=False, sideband_cooling=False)
        seq.sync()
        seq += crystal.state_preparation(endpoints=Zones.ZONE_3)
        seq.sync()
        seq += crystal.pi(transition=Transitions.SM12_TO_DM52, endpoints=Zones.ZONE_3)
        seq.sync()
```

```
11          seq += crystal.detection(endpoints=Zones.ZONE_3, detect_background=True)
12          seq.sync()
13          json_string = seq.get_json_string()
14          return json_string
```

Listing 4.9: An experiment derives from the `Experiment` class and overrides the `schedule` method. This example shows a typical sequence involving cooling, state preparation, a π pulse and a detection of a `CryoCrystal`. The `sync` statements indicate that the pulses will be scheduled sequentially. The last statement translates the `LinearSequence` into the JSON string.

The `run` method takes the number of shots of the experiment and calls the `schedule` method to create the JSON string before executing it on the Zedboard. The method returns the message sent back from the Zedboard when the data point was taken. It contains the raw PMT counts along with the readout channel results, containing the PMT counts post-processed on the CPU of the Zedboard.
The `scan_1D` and `scan_2D` methods perform scans over one or two variables, respectively (see Lst. 4.10). They use the feature presented in Lst. 4.8 to change the value of the parameters and create a JSON for each new data point of the scan. Using the `run` method, they then execute each of the JSON files consecutively and return a dictionary with the scan parameters as keys and the return value of `run` as the values. For 1D scans, I have also implemented a live-plotting feature. This plots the data points as soon as they are returned by the Zedboard while scheduling the next data point. This helped us a lot during data taking as we could directly see when things went wrong. Additionally, Carmelo Mordini also implemented the functionality for timestamping and saving the experimental data, such that they could be stored and then analysed offline.

```
1  from pathlib import Path
2  exp = CryoExperiment(
3      "Test", data_dir=Path("<results_folder>"), zb=Zedboard(<connection_details>)
4  )
5  value_1 = exp.scan_1D(
6      num_shot=50,
7      parameter=<time parameter used to parameterise the pi pulse>,
8      param_kwargs = {
9          "pulse_type": PulseTypes.PI,
10         "transition": Transitions.SM12_TO_DM52,
11     },
12     scan_range=[1.4, 100, 40],
13 )
```

Listing 4.10: This code snippet shows how to perform a scan with `pycrystal`. First, one instantiates the experiment class with a name, a directory specifying where the data will be stored and a Zedboard instance which is used to connect to the Zedboard and run the sequences there. Then you can call `scan_1D` on the experiment instance and provide the number of shots, the parameter to be scanned, the specific keyword arguments for the parameter to be scanned and the scan range (or a list of scan values). Here, we are scanning the time parameter used to parameterise the pi pulse on a specific transition and we would expect to see a Rabi oscillation.

Chapter 5

Experimental Setup

In this chapter, I will give an overview of the Cryo setup which I used to perform experiments on with pycrystal (see chapter 6). I will first present an overall picture, continue by explaining the surface trap more in detail and then what lasers — and where — we are using in the setup.

5.1 The Cryo Setup

Our setup utilises a two-dimensional Paul surface-electrode trap (SET) to trap and control $^{40}Ca^{+}$ ions. Fig. 5.1 shows a simple schematic of the setup. The trap lies within a 6 K cryostat (see Fig. 5.1 (left)). The chamber is placed in a vacuum vessel and protected from room-temperature radiation by heat shields. This provides the ions with an environment with attenuated magnetic field fluctuations and ultra-high vacuum (UHV). As the ions are trapped using a combination of DC (direct current) and RF voltages applied to the electrodes of our trap, the cryostat chamber also houses various electronics like RC filters. Imaging is performed by collecting the light emitted by the ions with an objective which focuses it on either a PMT or a camera outside the cryostat. The vacuum chamber has three viewports used to deliver free-space lasers to the trap as well as a feedthrough for integrated optics fibres. Two sets of permanent magnets are placed outside the vacuum chamber to create a homogeneous magnetic field of 5.8 G at the position of the trap. The calcium atoms are emitted from a resistively heated oven mounted on a vacuum flange.

The trap is a two-dimensional SET with integrated waveguides. On the upper layer, there are both DC and RF electrodes creating the electric fields that trap the ion. The RF electrodes lie between the inner and the outer DC electrodes and create an effective potential trapping on the radial plane, perpendicular to the axis of the trap. The inner DC electrode is split into two and used to tilt the principal axes of the radial potential with respect to the plane of the trap, which is necessary to cool the ion using beams parallel to the surface. The outer electrodes are separated into nine segments, each 120 µm wide, which allows shaping the electric potential along the trap axis (see Fig. 5.1 (right)). The electrode voltages can all be controlled separately by a Fastino controller, a 32 channel DAC capable to execute real-time voltage waveforms (see section 5.1.1). We use it to position the ions in the trap and to move them from one region to another.
The silicon oxide layer below the electrodes hosts embedded silicon nitride waveguides in which

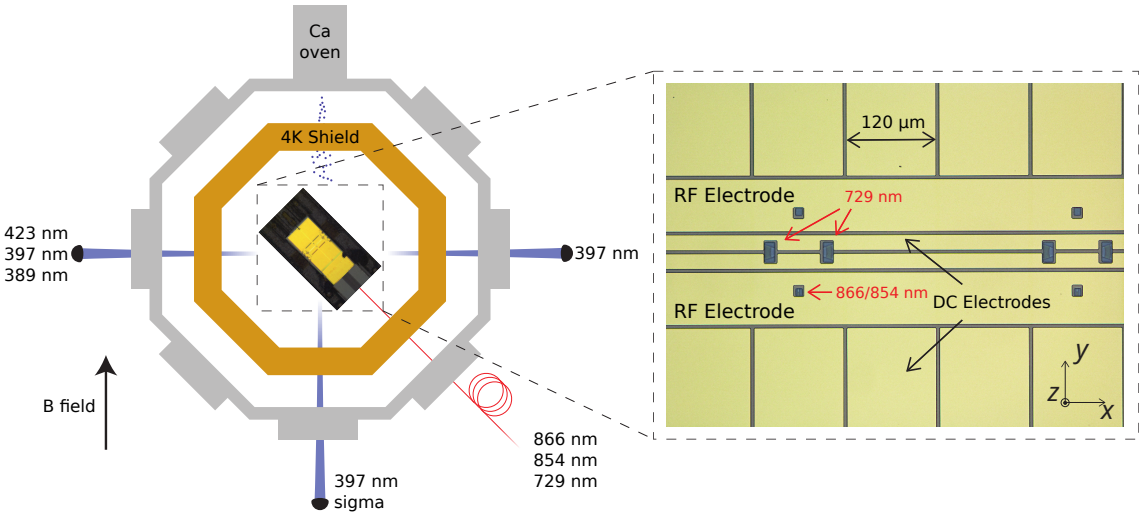


Figure 5.1: Schematic of our experimental setup. (left) A SET with integrated photonics is located in a cryostat copper chamber and is held in UHV. Two permanent magnets outside of the vacuum chamber generate the 5.8 G magnetic field in the region of the trap. Blue lasers are delivered to the trap through three viewports, red and infrared light through integrated waveguides. (right) The SET consists of 18 outer DC electrodes separated by 120 µm, two inner DC electrodes, two RF electrodes and laser outcouplers for the integrated waveguides (highlighted in red). The DC and the RF electrodes create the trapping potential and are used to confine and transport the ions between the *zones*. The trap has three zones, only two of which are shown here, characterised by the laser outcouplers. Image adapted from [32].

light is coupled directly into the chip and which deliver it directly to the ion at the position of interest. In three equally separated regions, the trap has large openings in the middle DC electrodes in correspondence with grating outcouplers terminating the waveguides, which focus the output light at the position of the ion (see Fig. 5.1 (right)). These *zones* define where we can operate and manipulate the ions. We call the second zone, i.e. the one in the middle, the *loading zone* as we use it for generating and trapping ions, and the third zone the *working zone* as it is where we mainly manipulate and read out the ions. We load and work in different zones as the lasers used to ionise the atoms charge up the exposed dielectric on the trap, modifying the electric potential landscape in the vicinity of the ion. The first zone is currently not used.

The integrated waveguides deliver lasers at 729 nm, 854 nm and 866 nm to each zone. The 729 nm laser is used for coherent control and passes a free-space double-pass AOM followed by a fibre-coupled single-pass AOM for each zone before being delivered to the trap. Lasers of wavelength 423 nm, 389 nm and 397 nm are delivered in free-space to the loading zone (see Fig. 5.1 (left)). The first two lasers are used for the photoionisation of the atoms, the last one can be used for both cooling and readout. The only two free-space lasers delivered to the working zone are at 397 nm where one is π polarised and the other is σ polarised. As we only have a 397σ beam in the working zone, we can only perform state preparation and EIT cooling there (see sections 2.2.2, 2.3.2). For further details about the setup and integrated optics we refer to [11].

5.1.1 Fastinos

We use DAC boards, called Fastinos[1], to control the DC voltages of our trapping electrodes on the trap. Each board is capable of outputting synchronous, time-dependent voltage sequences, hereby called *waveforms*. We use waveforms in the experiments described in this thesis to transport the ion between two trap zones. The execution of predefined waveforms can be triggered with a TTL signal.
Waveforms are setup-specific and have to be optimised experimentally for the specific experiment for which they are used. Waveform synthesis is a topic that has been already developed in our lab and is outside of the scope of pycrystal, so I will not go further into details.

[1] `https://github.com/quartiq/fastino`

Chapter 6

Ramsey Characterisation Experiment

In this chapter, I will present the Ramsey characterisation experiments we performed on the Cryo setup using the `pycrystal` library. I will first explain the typical experiment sequence and then show the Ramsey- and spin echo sequences in our working zone in section 6.1. In section 6.2 I will show our efforts in measuring the laser phase stability of multi-zone operations with a simple Ramsey scheme which suffered from differential laser phase noise. Section 6.3 then closes with a demonstration of an improved multi-zone Ramsey scheme that is insensitive to the phase fluctuations of the 729 nm light.

A typical experiment sequence starts with a cooling step, i.e. far- and mid-Doppler cooling, and a state preparation in the working zone, and ends with a state readout of the qubit. Additional sequences can be put in between the state preparation and the readout as shown in Lst. 6.1.

```
1  class RamseyExperiment(Experiment):
2      def schedule(self) -> str:
3          seq = LinearSequence("Ramsey experiment", auto_channel_mask=True)
4          crystal = CryoCrystal("Ramsey experiment")
5          seq += crystal.default_cooling(eit_cooling=False, sideband_cooling=False)
6          seq.sync()
7          seq += crystal.state_preparation(endpoints=Zones.ZONE_3)
8          seq.sync()
9          # ... additional pulse sequence
10         seq.sync()
11         seq += crystal.detection(endpoints=Zones.ZONE_3, detect_background=True)
12         json_string = seq.get_json_string()
```

```
13         return json_string
```

Listing 6.1: The Ramsey experiment class. The generic sequence starts with a cooling step followed by a state preparation in zone 3 and finishes with a read-out in zone 3 with background detection. The placeholder between the state preparation and the detection is the region where we put our ion manipulation sequence.

Lst. 6.2 shows the code to perform the scan. We can instantiate the experiment class with the arbitrary keyword arguments, e.g. `wait_time` or `spin_echo`, which will be added as instance attributes and the `zb` keyword which takes a `Zedboard` instance as discussed in section 4.2.1. We then perform a scan over the phase of a single-pass AOM of the 729 nm laser with the specifications given in `param_kwargs`. We take a scan range from 0 to 360 degrees with 100 data points of 50 shots each.

```
scan_range = [0, 360, 100]
ramsey_exp = RamseyExperiment("Ramsey experiment", wait_time=100, spin_echo=True, zb=zb)
result = ramsey_exp.scan_1D(
    num_exp=50,
    parameter=CarrierPulses.phase_729_sp,
    param_kwargs={
        "pulse_type": PulseTypes.PI_HALF,
        "transition": Transitions.SM12_TO_DM12,
        "zone": Zones.ZONE_3,
    },
    scan_range=scan_range,
)
```

Listing 6.2: We can use this code snippet to run the Ramsey experiment while scanning the phase of the 729 single-pass going to the working zone from 0 to 360 degrees with 100 data points of 50 shots each. The `zb` is a `Zedboard` instance again, connecting to the Zedboard. The parameter `CarrierPulses.phase_729_sp` is located in the `CarrierPulses` class storing the parameters we use for driving the 729 nm laser.

6.1 Single-zone Ramsey- and Spin Echo Sequences

We performed Ramsey- and spin echo sequences to test the phase coherence in the working zone which we wanted to use as a baseline for comparison with the transport sequences in the next chapters. The two different pulse sequences are explained in section 2.4.

6.1.1 Code

The Ramsey experiment sequence is shown in Lst. 6.3. Depending on the `spin_echo` keyword passed to the constructor of the experiment class we execute a spin echo or a pure Ramsey sequence on the qubit.

```
seq += crystal.pi_half(
    transition=Transitions.SM12_TO_DM12, endpoints=Zones.ZONE_3, phases=[0, 0]
)
seq.add_global_wait(wait_time=self.wait_time / 2)
if self.spin_echo:
    seq += crystal.pi(
        transition=Transitions.SM12_TO_DM12,
        endpoints=Zones.ZONE_3,
        phases=[0, 0],
    )
seq.add_global_wait(wait_time=self.wait_time / 2)
seq += crystal.pi_half(
    transition=Transitions.SM12_TO_DM12, endpoints=Zones.ZONE_3
)
```

Listing 6.3: The Ramsey sequence consists of two `pi_half` pulses interleaved by a wait time whose length is passed to the experiment constructor. If `self.spin_echo` is true (also set in the constructor) we perform another pi pulse after half the wait time.

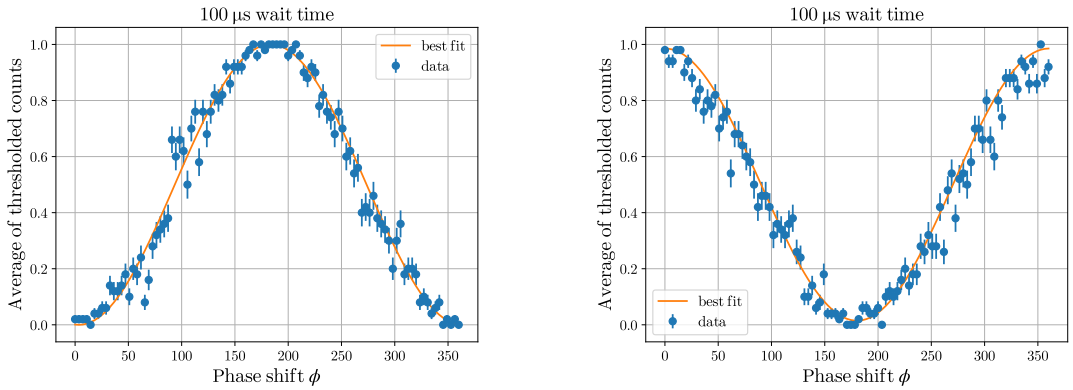
In the pure Ramsey scheme, we start with a $\pi/2$ pulse bringing the qubit into an equal superposition of $\left|S_{1/2}, m = -1/2\right\rangle$ and $\left|D_{5/2}, m = -1/2\right\rangle$. With the `phases=[0, 0]` argument, we keep the phases of both AOMs of the 729 nm laser at zero for the first two pulses, and only scan the phase of the final pulse. Then we add a wait time before another $\pi/2$ pulse on the same transition. The length of the wait time is determined by the `wait_time` argument passed to the constructor. The second $\pi/2$ pulse does not fix the phases to enable us to scan over it during the experiment.
The spin echo sequence adds the additional π pulse with fixed phases on the same transition after half the wait time.

6.1.2 Results

We performed both sequences for eight different wait times in the range of 0 µs to 10 000 µs measuring the probability of detecting the bright state as a function of the phase offset of the single-pass AOM. We measured the PMT counts for every shot and thresholded them at a value $n_{thr} = 22$.

The measurement results show the expected Ramsey oscillation (see eq. 2.23). Figs. 6.1a and 6.1b plot the average thresholded counts against the phase offset ϕ for a wait time of 100 µs for the Ramsey- and the spin echo sequence, respectively. The errors on the average were calculated as described in 2.2.3, taking the minimum of the quantum projection noise (eq. 2.17) and the error calculated using Laplace's rule of succession (eq. 2.18). The orange line corresponds to the fit of the Ramsey oscillation to the data.

We fitted the Ramsey- and spin echo oscillation to the data for each wait time to extract the amplitude (see Appendix A). The uncertainty is given by the standard error of the fit. The amplitude contrast is fitted to an exponential and Gaussian decay (eqs. 2.24, 2.25) as well as a combination of both in Figs. 6.2a and 6.2b. The Gaussian decay which is modelling the low bandwidth noise should be suppressed by the additional π pulse of the spin echo sequence. It

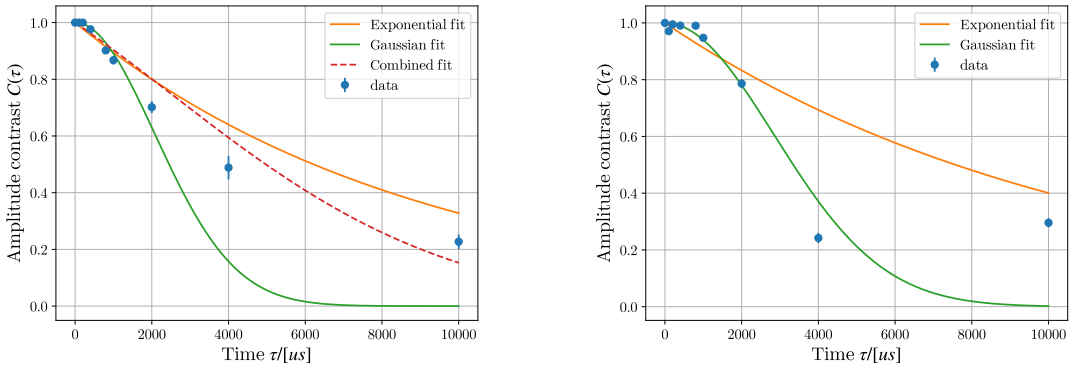


(a) Ramsey sequence oscillation.

(b) Spin echo sequence oscillation.

Figure 6.1: (a) Ramsey sequence and (b) spin echo oscillations for a wait time of 100 µs. The data points (blue) correspond to the average of the thresholded counts over 50 repetitions with an error given by the quantum projection noise. The data is fitted to a Ramsey oscillation (orange, see eq. 2.23) The best fits for amplitude contrast and phase shift are given by $A_r = 1.000 \pm 0.007, \phi_r = (-3.6 \pm 0.9)$ deg for the Ramsey sequence and $A_{sp} = 0.971 \pm 0.010, \phi_{sp} = (-180.3 \pm 1.1)$ deg for the spin echo sequence.

is anyway shown for comparison. We left out the combined fit for the spin echo sequence which coincides with the Gaussian fit.



(a) Ramsey amplitude contrast.

(b) Spin echo amplitude contrast.

Figure 6.2: Amplitude contrast of the (a) Ramsey- and the (b) spin echo sequence (blue) plotted against the interrogation time τ. The errors correspond to the standard error of the amplitude fit. The data of the Ramsey sequence is fitted to an exponential- (orange) and a Gaussian decay (green) and a combination of both (dotted, red) while the data of the spin echo sequence is fitted to an exponential- and a Gaussian decay.

Neither decay model fits the data well. This indicates that the noise sources in our laboratory cannot be fully described by high- and low-bandwidth noise. The low amplitude for $\tau = 4000$ µs is either a measurement error or could already hint at a noise source at a discrete frequency which was also observed in [11]. However, we do not have enough data points to make more accurate statements about this or the nature of the noise. We should probe especially the region from 2000 µs to 10 000 µs with more data points.

6.2 Multi-zone Ramsey Sequence

After successfully demonstrating the Ramsey- and spin echo sequences in the working zone, we wanted to perform a Ramsey experiment involving two zones to determine the viability of those operations. The first scheme we tried was applying a $\pi/2$ pulse on the optical qubit in zone 3, transporting the qubit to zone 2 and performing another $\pi/2$ pulse on the same transition before transporting it back for readout.

6.2.1 Code

Lst. 6.4 shows the code we used for the first multi-zone Ramsey experiment. We again start with a $\pi/2$ pulse with fixed phases to bring the qubit into an equal superposition of $\left|S_{1/2}, m = -1/2\right\rangle$ and $\left|D_{5/2}, m = -1/2\right\rangle$. Then we transport the ion into zone 2 with the `transport_event()` function which triggers the Fastino with a TTL signal. We add an additional wait time after that to make sure that the trap voltages settle and the ion does not move anymore. We then contine with another $\pi/2$ on the optical qubit transition whose phase we scan. After that, we transport the ion back to zone 3 for the readout.

```
seq += crystal.pi_half(
    transition=Transitions.SM12_TO_DM12, endpoints=Zones.ZONE_3, phases=[0, 0]
)
seq.sync()

# 3 -> 2
seq += transport_event()
seq.add_global_wait(wait_time=100)

seq += crystal.pi_half(
    transition=Transitions.SM12_TO_DM12, endpoints=Zones.ZONE_2
)
seq.sync()

# 2 -> 3
seq += transport_event()
seq.add_global_wait(wait_time=100)
```

Listing 6.4: The multi-zone Ramsey experiment. It starts with a `pi_half` pulse on the optical qubit transition. Then we transport the ion to zone 2 with `transport_event()` and apply another `pi_half` pulse where we scan over the phase. After that we transport the ion back.

The scans are performed with the code in Lst. 6.2 without the additional keyword arguments in the experiment constructor and scanning the phase in zone 2 rather than zone 3.

6.2.2 Results

We performed the multi-zone Ramsey sequence multiple times, measuring the probability of detecting the bright state as before.

Fig. 6.3 shows the average thresholded counts plotted against the scanned phase offset ϕ for two of those experiments, representing what we saw in all the other experiments. The two selected measurements were performed 1.5 min apart from each other, the errors are calculated as in the previous section.

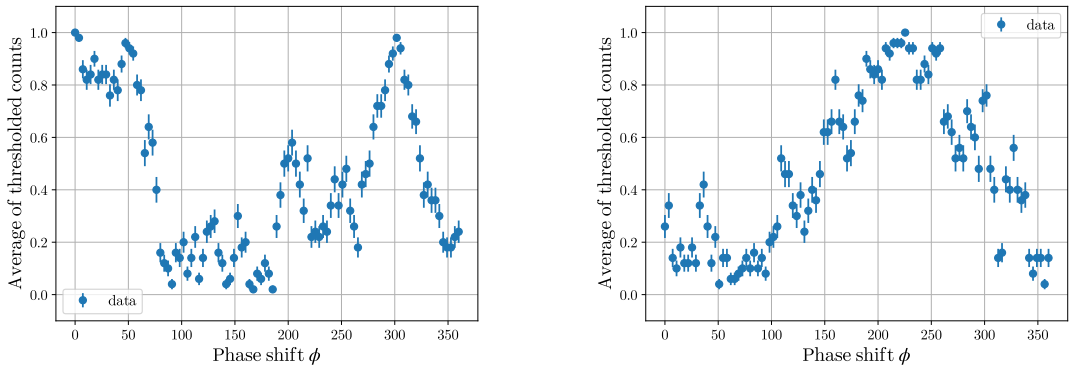


Figure 6.3: Two multi-zone Ramsey sequence measurements representing the data that was taken. Most measurements didn't show any clear sinusoidal pattern (left) with a few exceptions (right).

Both measurements show a high contrast close to one. However, with a few exceptions, most measurements did not show a clear sinusoidal pattern, which makes us think that there are phase fluctuations in the system faster than the timescale needed to complete a scan. One plausible reason could be the phase noise between the 729 nm lasers in the different zones. As the fibres going from the double-pass to the single-pass AOMs are about two metres long, even small thermal fluctuations, acoustic noise or stress on the fibres could change their length by a sufficiently large amount to cause phase noise.

6.3 Laser Phase-insensitive Multi-zone Ramsey Sequence

To circumvent the effect of relative phase fluctuations between zones, we decided to utilise a laser phase-insensitive scheme for the multi-zone Ramsey sequence (see section 2.4.2).

In the phase-insensitive scheme, we map the optical qubit to the Zeeman qubit before the transport and interrogation time τ while keeping the rest of the sequence the same. We do this by performing a π pulse on the $\left|D_{5/2}, m = -1/2\right\rangle \leftrightarrow \left|S_{1/2}, m = +1/2\right\rangle$ transition after the first $\pi/2$ and before the second $\pi/2$ pulse in the multi-zone Ramsey sequence (see section 6.2).

6.3.1 Code

Lst. 6.5 shows the code we used for the phase-insensitive multi-zone Ramsey experiment. It is similar to the one that we used in section 6.2.1 with an addition of π pulses on the $\left|D_{5/2}, m = -1/2\right\rangle \leftrightarrow$

$\left|S_{1/2}, m = +1/2\right\rangle$ transition after the $\pi/2$ pulse in zone 3 and before the $\pi/2$ pulse in zone 2. The scans are performed in the same way as in section 6.2.1.

```
seq += crystal.pi_half(
    transition=Transitions.SM12_TO_DM12, endpoints=Zones.ZONE_3, phases=[0, 0]
)
# Mapping onto Zeeman qubit
seq += crystal.pi(
    transition=Transitions.SP12_TO_DM12, endpoints=Zones.ZONE_3, phases=[0, 0]
)
seq.sync()

# 3 -> 2
seq += transport_event()
seq.add_global_wait(wait_time=100)

# Mapping back to optical qubit
seq += crystal.pi(
    transition=Transitions.SP12_TO_DM12, endpoints=Zones.ZONE_2, phases=[0, 0]
)
seq += crystal.pi_half(
    transition=Transitions.SM12_TO_DM12, endpoints=Zones.ZONE_2
)
seq.sync()

# 2 -> 3
seq += transport_event()
seq.add_global_wait(wait_time=100)
```

Listing 6.5: The phase-insensitive multi-zone Ramsey experiment. It modifies the multi-zone Ramsey experiment in Lst. 6.4 by adding `pi` pulses after the first and before the second `pi_half` pulse to map the optical qubit onto the Zeeman qubit.

6.3.2 Results

We performed the phase-insensitive multi-zone Ramsey sequence multiple times. Fig. 6.4 shows the average thresholded counts plotted against the scanned phase offset ϕ for two examples of the experiments, representing what we saw in all the other experiments.

The data shows a sinusoidal contrast pattern with best fits given by $A_{\text{left}} = 0.898 \pm 0.013, \phi_{\text{left}} = (109.3 \pm 1.2)\,\text{deg}$ and $A_{\text{right}} = 0.872 \pm 0.013, \phi_{\text{right}} = (97.7 \pm 1.2)\,\text{deg}$. We mainly attribute the amplitude contrast loss to the transport operation, which heats up the ion. There might also be a residual effect from the phase noise that is not cancelled by the hybrid qubit scheme as the interrogation time τ is non-zero due to the transports and their successive wait times. The

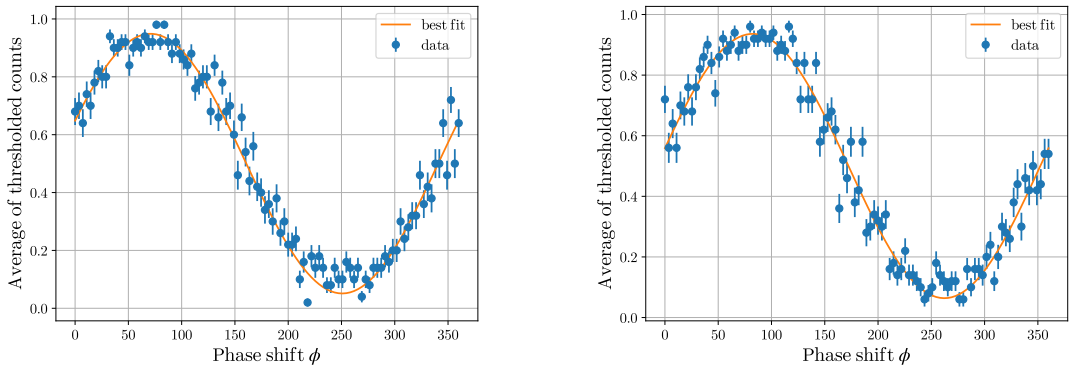


Figure 6.4: Two laser phase-insensitive multi-zone Ramsey sequence measurements taken 12 min apart. The plot shows the average of the thresholded counts plotted against the phase offset of the single-pass AOM in zone 2 (blue) fitted to a sinusoidal Ramsey oscillation (orange). The best fits for amplitude and phase shifts are given by $A_{\text{left}} = 0.898 \pm 0.013, \phi_{\text{left}} = (109.3 \pm 1.2)$ deg and $A_{\text{right}} = 0.872 \pm 0.013, \phi_{\text{right}} = (97.7 \pm 1.2)$ deg.

phase offset seems to drift only slightly over a period of 12 min and could be something that can be calibrated. It is due to the free evolution of the qubit in the Zeeman subspace and is thus influenced by the fluctuation and variation of the magnetic field along the transport path.

Chapter 7

Conclusion

There are many different approaches to creating quantum computers with trapped ions. They differ in the species of ions they use, the trap technology and design and their scaling approaches. This thesis tried to tackle the problem of creating a python framework that is able to interface with that flexibility. The library was designed with the goal to abstract various setups and creating a python interface that can be used to create pulse sequences for them in a way that the user has meaningful physics in mind.

The presented python package `pycrystal` follows a bottom-up approach to modelling the laboratory setup. To offer the desired flexibility, it remains very generic on its basis, leaving the user to implement most of the specific details. The framework builds on top of an already existing python API called `ionpulse sequence generator`. This library provides classes to create low-level instructions that can be sent to the M-ACTION control system. `pycrystal` abstracts the hardware used to control the laser pulses in an object-oriented way with its `beamline` module. Its main class `BeamLine` builds the highest-level abstraction of the lasers in the setups, offering support for different architectures like QCCD or long chains of ions with single-ion addressing. Furthermore, the `crystal` module offers the tools to create classes that model any ion crystal on a trap. Those crystal classes are intended to make use of the beamlines to implement all pulses that can be applied to the ions as methods of the class. This allows the end-user to directly apply the desired pulses to the ions while `pycrystal` takes care of triggering the hardware to deliver the requested pulses to the correct positions. Finally, `pycrystal` also offers the possibility of directly running pulse sequences on the control system and performing scans on the parameters used in the sequences. This was developed for debugging and testing purposes and will be moved into another project at a later point.

We demonstrated the new python framework on the Cryo setup of the TIQI group. This setup features a SET with a QCCD architecture and integrated photonics. We characterised the phase stability of the system by performing Ramsey experiments in a single trapping zone and multi-zone Ramsey schemes. The single-zone Ramsey experiments consisted of both Ramsey- and spin echo sequences. We used them to measure the amplitude contrast of the Ramsey oscillation as a function of the interrogation time. Our results did follow neither low- nor high-bandwidth noise models and thus indicated other noise sources in our laboratory.

The multi-zone Ramsey schemes were performed with a fixed interrogation time to determine the viability of operations on two zones. We noticed differential laser phase fluctuations between the zones and proposed and implemented a scheme to bypass that issue. To our knowledge, this is the first time multi-zone operations on a SET with integrated optics was demonstrated.

Future work will be devoted to benchmark the capability of such a trap for distributed quantum operations, which involves a full characterisation of both single-zone and multi-zone operations with Ramsey and spin echo sequences. We can also investigate the phase fluctuations we saw in our first multi-zone scheme by, for example, using shorter fibres coupling the single-pass AOMs to the double-pass AOM of the 729 nm light or protecting them from thermal fluctuations and acoustical noise.

On the software side, we can also add more features to `pycrystal`. This can be new beamline objects such as AODs, implementing recent features of the `ionpulse_sequence_generator` such as the phase tracking of the qubits, adding capabilities to the crystal classes such as keeping track of the position of the crystal, which is especially useful for sequences involving transport events, or adding features to the `ionpulse_sequence_generator` itself. This could be, for example, the implementation of a way to directly specify whether to add a sequence sequentially or in parallel. Moving or migrating the experiment class from `pycrystal` into another package is also something to be done. A scheduler functionality has to be implemented and integrated within pycrystal, but it doesn't have to be necessarily part of it, as the package scope is on the generation of experimental sequences. A future task will then be to develop an OS-independent multi-threaded scheduler, where generation and execution of the sequences can be kept as separate tasks.

Furthermore, for the experiments described in this thesis we used the Ionizer GUI as an interface between pycrystal and the control system. We utilised it both as a parameter database and as a means to trigger the experiments on the Zedboard. In the future, those functionalities should be implemented separately, defining a standard for the parameter database and switching to a more convenient communication with the hardware which can be integrated in the experiment scheduler.

Appendix A

Ramsey- and Spin-echo Data

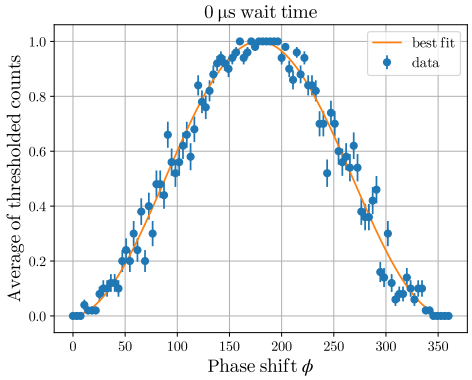


Figure A.1: Ramsey sequence with $\tau = 0\,\mu s$. Best fit parameters are given by $A = 1.000 \pm 0.007, \phi = (1.6 \pm 0.9)$ deg.

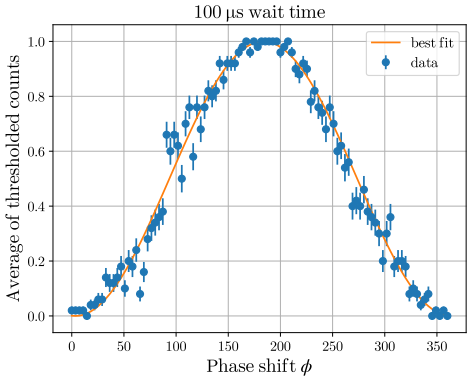


Figure A.2: Ramsey sequence with $\tau = 100\,\mu s$. Best fit parameters are given by $A = 1.000 \pm 0.007, \phi = (-3.6 \pm 0.9)$ deg.

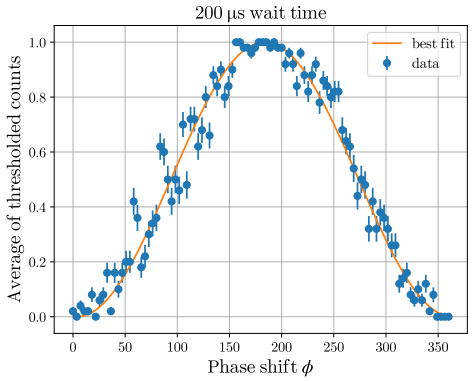


Figure A.3: Ramsey sequence with $\tau = 200\,\mu s$. Best fit parameters are given by $A = 1.000 \pm 0.008, \phi = (-3.8 \pm 1.1)$ deg.

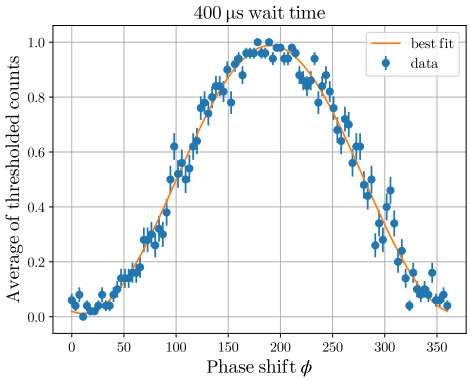


Figure A.4: Ramsey sequence with $\tau = 400\,\mu s$. Best fit parameters are given by $A = 0.977 \pm 0.009, \phi = (-10.3 \pm 1.0)$ deg.

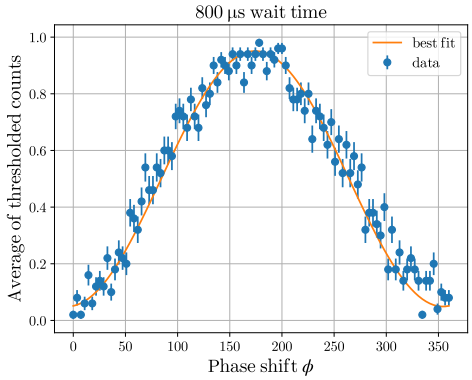


Figure A.5: Ramsey sequence with $\tau = 800\,\mu s$. Best fit parameters are given by $A = 0.902 \pm 0.012, \phi = (5.7 \pm 1.2)$ deg.

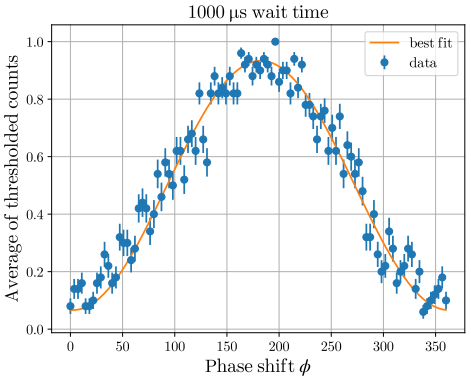


Figure A.6: Ramsey sequence with $\tau = 1000\,\mu s$. Best fit parameters are given by $A = 0.867 \pm 0.015, \phi = (-2.5 \pm 1.4)$ deg.

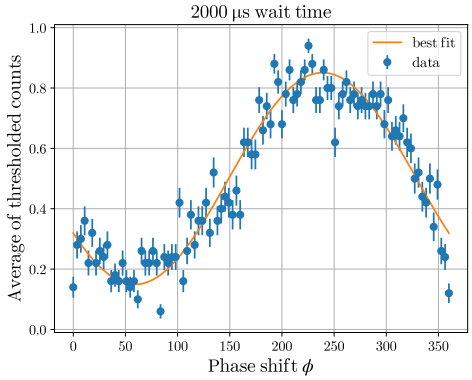


Figure A.7: Ramsey sequence with $\tau = 2000\,\mu s$. Best fit parameters are given by $A = 0.70 \pm 0.02, \phi = (-59.0 \pm 1.9)$ deg.

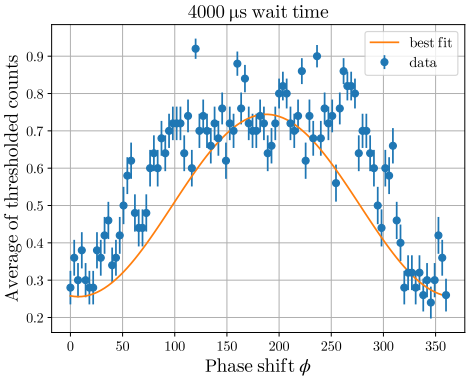


Figure A.8: Ramsey sequence with $\tau = 4000\,\mu s$. Best fit parameters are given by $A = 0.49 \pm 0.04, \phi = (-8 \pm 5)$ deg.

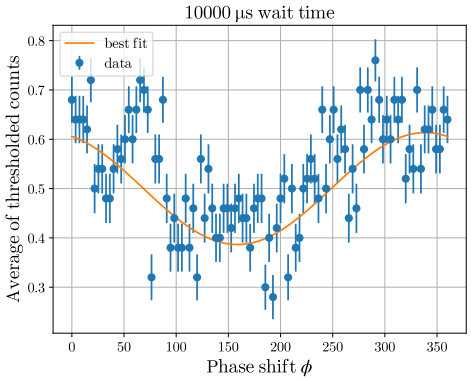


Figure A.9: Ramsey sequence with $\tau = 10\,000\,\mu s$. Best fit parameters are given by $A = 0.23 \pm 0.03, \phi = (-159 \pm 6)$ deg.

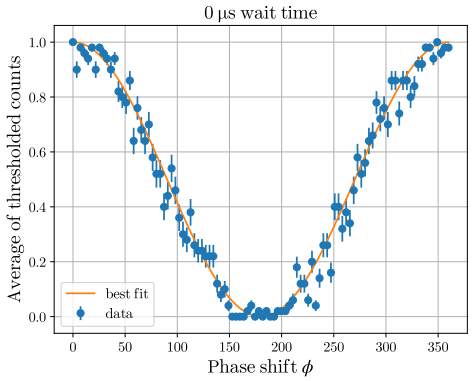


Figure A.10: Spin-echo sequence with $\tau = 0\,\mu s$. Best fit parameters are given by $A = 1.000 \pm 0.010, \phi = (-180.2 \pm 1.0)$ deg.

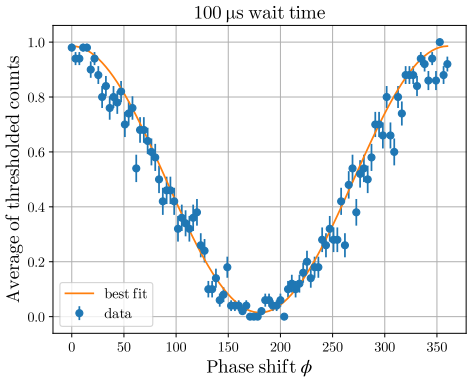


Figure A.11: Spin-echo sequence with $\tau = 100\,\mu s$. Best fit parameters are given by $A = 0.971 \pm 0.010, \phi = (-180.3 \pm 1.1)$ deg.

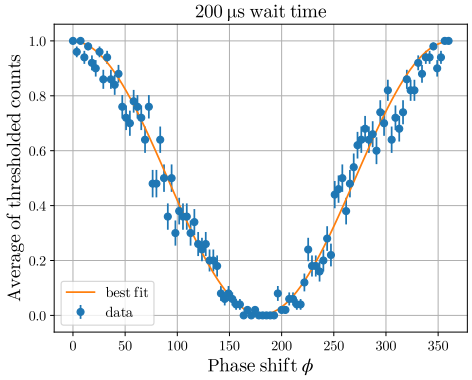


Figure A.12: Spin-echo sequence with $\tau = 200\,\mu s$. Best fit parameters are given by $A = 0.995 \pm 0.007, \phi = (-180.7 \pm 1.0)$ deg.

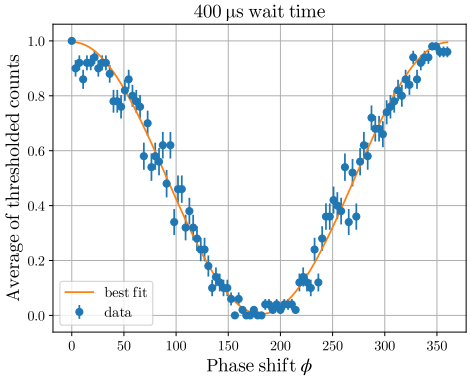


Figure A.13: Spin-echo sequence with $\tau = 400\,\mu s$. Best fit parameters are given by $A = 0.991 \pm 0.008, \phi = (-181.2 \pm 1.0)$ deg.

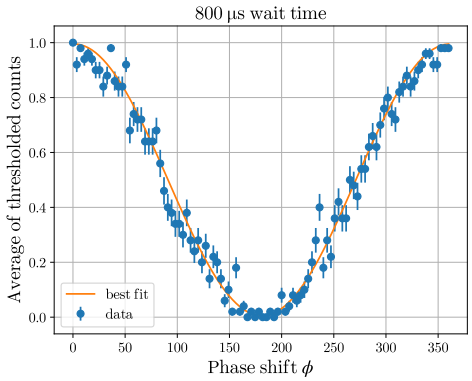


Figure A.14: Spin-echo sequence with $\tau = 800\,\mu s$. Best fit parameters are given by $A = 0.990 \pm 0.008, \phi = (-181 \pm 9)$ deg.

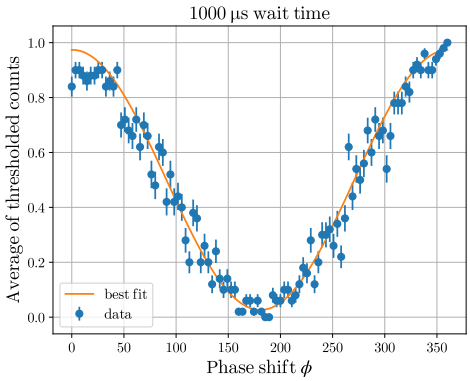


Figure A.15: Spin-echo sequence with $\tau = 1000\,\mu s$. Best fit parameters are given by $A = 0.948 \pm 0.010, \phi = (-180.8 \pm 1.2)$ deg.

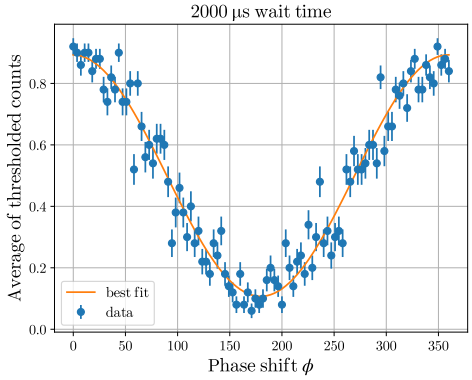


Figure A.16: Spin-echo sequence with $\tau = 2000\,\mu s$. Best fit parameters are given by $A = 0.786 \pm 0.016, \phi = (-179.6 \pm 1.5)\,\text{deg}$.

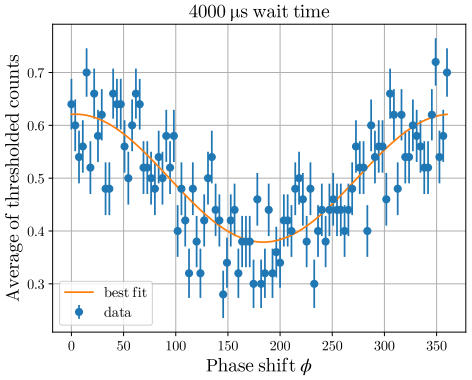


Figure A.17: Spin-echo sequence with $\tau = 4000\,\mu s$. Best fit parameters are given by $A = 0.242 \pm 0.018, \phi = (-184 \pm 4)\,\text{deg}$.

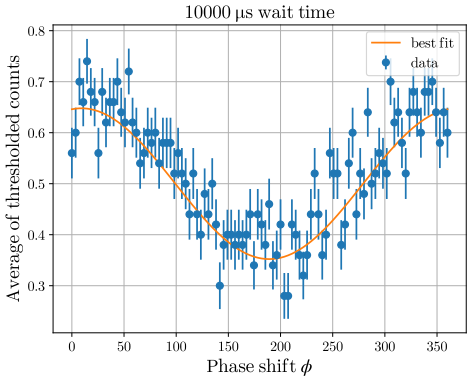


Figure A.18: Spin-echo sequence with $\tau = 10\,000\,\mu s$. Best fit parameters are given by $A = 0.296 \pm 0.016, \phi = (-189 \pm 3)\,\text{deg}$.

Bibliography

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

[2] C. Monroe et al. "Resolved-Sideband Raman Cooling of a Bound Atom to the 3D Zero-Point Energy". In: *Phys. Rev. Lett.* 75 (22 Nov. 1995), pp. 4011–4014.

[3] Ch. Roos et al. "Quantum State Engineering on an Optical Transition and Decoherence in a Paul Trap". In: *Phys. Rev. Lett.* 83 (23 Dec. 1999), pp. 4713–4716.

[4] C. Monroe et al. "Demonstration of a Fundamental Quantum Logic Gate". In: *Phys. Rev. Lett.* 75 (25 Dec. 1995), pp. 4714–4717.

[5] 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.

[6] J. M. Pino et al. "Demonstration of the trapped-ion quantum CCD computer architecture". In: *Nature* 592.7853 (Apr. 2021), pp. 209–213.

[7] Jungsang Kim and Changsoon Kim. "Integrated Optical Approach to Trapped Ion Quantum Computation". In: *Quantum Information and Computation* 9 (Dec. 2007).

[8] M. Ivory et al. "Integrated Optical Addressing of a Trapped Ytterbium Ion". In: *Physical Review X* 11.4 (Nov. 2021).

[9] R. J. Niffenegger et al. "Integrated multi-wavelength control of an ion qubit". In: *Nature* 586.7830 (Oct. 2020), pp. 538–542.

[10] Karan K. Mehta et al. "Integrated optical addressing of an ion qubit". In: *Nature Nanotechnology* 11.12 (Dec. 2016), pp. 1066–1070.

[11] Maciej Malinowski. *Unitary and Dissipative Trapped-Ion Entanglement Using Integrated Optics*. Doctoral Thesis. Eidgenössische Technische Hochschule (ETH) Zürich, Nov. 2021.

[12] J. P. Home. *Quantum Information Processing [Course notes], ETH Zürich*. Mar. 2021.

[13] Daniel A. Steck. *Quantum and Atom Optics*. available online at `http://steck.us/teaching` (2006).

[14] Mark Fox. "Quantum Optics: An Introduction". In: Oxford master series in atomic, optical, and laser physics. Oxford: Oxford Univ. Press, 2006. Chap. 4.

[15] John David Jackson. *Classical electrodynamics*. 3rd ed. Wiley, Aug. 1998.

[16] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2011.

[17] Christopher J. Foot. "Atomic Physics". In: Oxford master series in atomic, optical, and laser physics. Oxford: Oxford Univ. Press, 2005. Chap. 2.

[18] D. Leibfried et al. "Quantum dynamics of single trapped ions". In: *Rev. Mod. Phys.* 75 (1 Mar. 2003), pp. 281–324.

[19] Regina Lechner et al. "Electromagnetically-induced-transparency ground-state cooling of long ion strings". In: *Phys. Rev. A* 93 (5 May 2016), p. 053401.

[20] Dong Hu et al. "Ramsey interferometry with trapped motional quantum states". In: *Communications Physics* 1.1 (June 2018), p. 29.

[21] Chi Zhang. *Scalable technologies for surface-electrode ion traps*. Doctoral Thesis. Eidgenössische Technische Hochschule (ETH) Zürich, 2021.

[22] *Zynq-7000 SoC Data Sheet: Overview*. `https://docs.xilinx.com/v/u/en-US/ds190-Zynq-7000-Overview`. Accessed: 2022-10-24.

[23] Vlad Negnevitsky. *Feedback-stabilised quantum states in a mixed-species ion system*. Doctoral Thesis. Eidgenössische Technische Hochschule (ETH) Zürich, Sept. 2018.

[24] Martin Stadler. "Integrated laser amplitude stabilization for mixed species trapped-ion experiments". MA thesis. Eidgenössische Technische Hochschule (ETH) Zürich, Sept. 2018.

[25] Marco E. Stucki. "Enabling a quantum-gate-level interface with a trapped ion control system". MA thesis. Eidgenössische Technische Hochschule (ETH) Zürich, Dec. 2021.

[26] Alexander Ferk. "Integrated Real-Time Phase Modelling for Trapped Ion Quantum Information Processing". MA thesis. Eidgenössische Technische Hochschule (ETH) Zürich, Oct. 2022.

[27] *Qiskit*. `https://qiskit.org/`. Accessed: 2022-10-29.

[28] *pyGSTi*. `https://www.pygsti.info/`. Accessed: 2022-10-29.

[29] Liberto Beltrán. "A Backend for scalable Ion-Trap Quantum Computing". MA thesis. Eidgenössische Technische Hochschule (ETH) Zürich, 2022.

[30] H. Häffner et al. "Precision Measurement and Compensation of Optical Stark Shifts for an Ion-Trap Quantum Processor". In: *Phys. Rev. Lett.* 90 (14 Apr. 2003), p. 143602.

[31] V. Negnevitsky et al. "Repeated multi-qubit readout and feedback with a mixed-species trapped-ion register". In: *Nature* 563.7732 (Nov. 2018), pp. 527–531.

[32] Alfredo Ricci Vasquez et al. *Control of an atomic quadrupole transition in a phase-stable standing wave*. 2022.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

A flexible python framework for generating pulse sequences for multi-zone operations in ion traps

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Müller	Mose Marius

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 31.10.2022	Mose Müller

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.