Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Semester Thesis

Single-Qubit Gates Calibration in PycQED using Superconducting Qubits

Stefania Balasiu

Supervisor
Dr. Christian Kraglund Andersen

Prof. Dr. Andreas Wallraff

August 17th, 2017

Contents

Abstract **1**

1 Introduction and Motivation **1**

2 Theoretical Background **3**

2.1 Short Introduction to Circuit Quantum Electrodynamics . . . 3

2.1.1 System Design . . . 3

2.1.2 Qubit-Resonator Interaction . . . 6

2.2 Single-Qubit Gates Calibration . . . 7

2.2.1 Resonator Spectroscopy . . . 8

2.2.2 Qubit Spectroscopy . . . 8

2.2.3 Rabi Measurement . . . 10

2.2.4 Ramsey Measurement . . . 11

2.2.5 T_1 Measurement . . . 13

2.2.6 DRAG Pulse Calibration . . . 13

3 The Python for Circuit Quantum Electrodynamics (PycQED) Framework **15**

3.1 Introduction . . . 15

3.2 Framework Structure . . . 16

3.2.1 The Qubit Object . . . 17

3.2.2 Measurement Control Flow . . . 21

3.2.3 The Measurement Analysis Structure . . . 24

3.3 Using PycQED in an Experiment . . . 30

4 Single-Qubit Calibration using PycQED **35**

4.1 Resonator Spectroscopy . . . 35

4.2 Qubit Spectroscopy . . . 36

4.3 Rabi Measurements . . . 37

4.4 Ramsey Measurements . . . 41

4.5 T1 Measurements . . . 44

4.6 DRAG Pulse Calibration . . . 45

5 Conclusions and Outlook **47**

Appendix A Summary of Contributions to the PycQED Framework **48**

Appendix B Qubit Designs Used for the Measurements in this Work **51**

Acknowledgements **53**

References **54**

Abstract

The prospect of building the first commercial universal quantum computer is closer to becoming a reality than ever before. In particular, circuit quantum electrodynamics architectures based on superconducting qubits have produced highly promising results in the field of quantum information and computation. In the Quantum Device Lab (QuDev) at ETH Zürich, traditional work on superconducting qubits employs several software packages such as LabVIEW for qubit control and data acquisition, and Wolfram Mathematica for data processing and analysis. This thesis introduces PycQED, a new software platform for circuit QED, that unifies the entire experiment under a single Python-based environment. The framework was developed by Professor Leonardo DiCarlo's group at the Delft University of Technology, and is currently being extended in collaboration with the Quantum Computing team at QuDev. The complete set of single-qubit operations has been implemented in PycQED, such that the framework can now be used to calibrate these gates for both the first and second qubit energy transitions. This thesis covers the first quantum computing experiments at QuDev that were performed using PycQED, and it is the first work in the group that presents the PycQED framework.

1. Introduction and Motivation

Interest in building a universal quantum computer has risen considerably in the last few decades as the necessary technology has become available. The increasing level of enthusiasm and anticipation for new quantum information processing technologies around the globe, from corporate giants such as Microsoft, Google, and IBM, to research universities, and even including the mainstream media [1], has given the quantum computing science a real boost forward in terms of scalability. In May 2016, IBM opened to the public their "Quantum Experience" 5 qubit quantum computer [2]. Easily accessible to anyone over the IBM cloud, this innovation has enabled a great deal of research in quantum computation and information processing with 5 qubits [3], [4]. Microsoft's Station Q project [5] brings together several prestigious research teams, including groups at ETH Zürich and Delft University of Technology, in a global effort to build a quantum computer. Microsoft has also started to extend its research into Europe, with the first Microsoft research laboratory expected to open at Delft and Copenhagen in the near future [6].

A significant advancement in the field of quantum computing has come in the form of software frameworks specifically aimed at controlling quantum computers. Until recently, quantum information processing laboratories would rely on various different software packages for qubit control and measurement analysis, such as custom-designed routines in LabVIEW and C++ for control, and Wolfram Mathematica, Python, or MATLAB for data processing (see [7] and [8] for an example of what was used in our group at QuDev). Recent efforts have been made to move to more universal software frameworks that are specifically designed to automate a quantum computing experiment. In 2016, Steiger et al. launched ProjectQ [9], [10], a python-based framework for writing and implementing quantum computing algorithms on any physical back-end. Another example is the recently-developed quantum instruction language (Quil) released by Rigetti Computing [11]. Quil allows to implement and execute algorithms on hybrid classical/quantum architectures that consist of feedback loops between classical and quantum computers.

The Python for circuit Quantum ElectroDynamics (PycQED) measurement acquisition framework presented in this thesis is another software initiative that promises to become a universal package for quantum computing experiments with superconducting qubits. The platform was developed by Professor Leonardo DiCarlo's group at QuTech, and its further development is now done in collaboration with the quantum computing team at QuDev. Switching to the PycQED platform has several benefits. PycQED brings the entire experimental control together under one single software that has the drivers for all the equipment, stores all the qubit parameters, and performs the data analysis. The advantage here is that PycQED has access to all the experimental parameters, and hence all the necessary modifications and updates to those parameters can be done behind the scenes, without the need for the user to manually change them (this was sometimes the case in our previous setup).

The PycQED platform also makes extensive use of object-oriented features [12], and this aspect may truly show its power as the experiment scales up and there will be a need to manipulate larger systems with many more qubits. The object-oriented features of a programming language are needed for any large-scale software framework, be it classical or quantum. The structure of object-oriented programs is built from grouping together code that creates logical entities called objects. In PycQED, each new qubit or instrument is an object that stores all the relevant physical parameters, such as transition frequencies, drive pulse parameters, and readout parameters for the qubit, and addresses, port numbers, and channel numbers for the physical instruments. These objects would also contain functions that allow the user to manipulate them and to access their parameters. This logical structure of object-oriented programming facilitates both the maintenance process and the further development of the framework. Furthermore, future quantum computer designs with many qubits would simply need to create as many instances of the qubit object as necessary, without the need to alter the structure of the code.

PycQED has great potential for circuit QED experiments, but at the start of our collaboration with the DiCarlo group, its structure was missing several features of interest for our team, such as further routines for single-qubit gates. These gates represent logical operations with one qubit and are the basic units of quantum computation. Thus, gaining full and proper control over them is essential to the implementation of higher level quantum algorithms. The goal of the project presented in this thesis is to implement all the necessary calibration routines for single-qubit gates on both the first and the second qubit energy transitions. This thesis will present the PycQED framework with a focus on its functionality for calibrating single-qubit operations. Section 2 will give a short overview of circuit QED, and single-qubit manipulations will be presented. The first part of Section 3 will explain the structure of PycQED, and the overall flow between its layers during a measurement. Then, Section 3.2.3 will present the PycQED measurement analysis module, and will highlight the structural changes that have been made during this project. The section will end with an example showing how we currently use PycQED in our lab to calibrate single-qubit gates. Finally, Section 4 will present in detail the main focus of this project: improving the data analysis routines for extracting the relevant parameters from single-qubit calibration measurements. Each subsection will provide details about the implementation of a specific calibration measurement and will show experimental results, thus illustrating the current stage in the development of the PycQED framework at QuDev.

2. Theoretical Background

2.1 Short Introduction to Circuit Quantum Electrodynamics

2.1.1 System Design

The most basic architecture of circuit QED consists of a superconducting transmon qubit capacitively coupled to a readout coplanar waveguide (CW) resonator. The design used for most of the measurements presented in this thesis is shown in Figure 2.1, and contains eight qubits (yellow), each coupled to its own readout resonator (red). The entire architecture is micro-fabricated on a chip, making it essentially two-dimensional. Typical materials used for fabrication are silicon or sapphire for the insulator substrate, and aluminum or niobium for the superconducting structures [13]. The eight qubit chip has a sapphire substrate (black), the qubits are made out of aluminum using electron beam processing, and the resonator structures were made by etching a thin film of niobium previously deposited onto the substrate using photolithography.

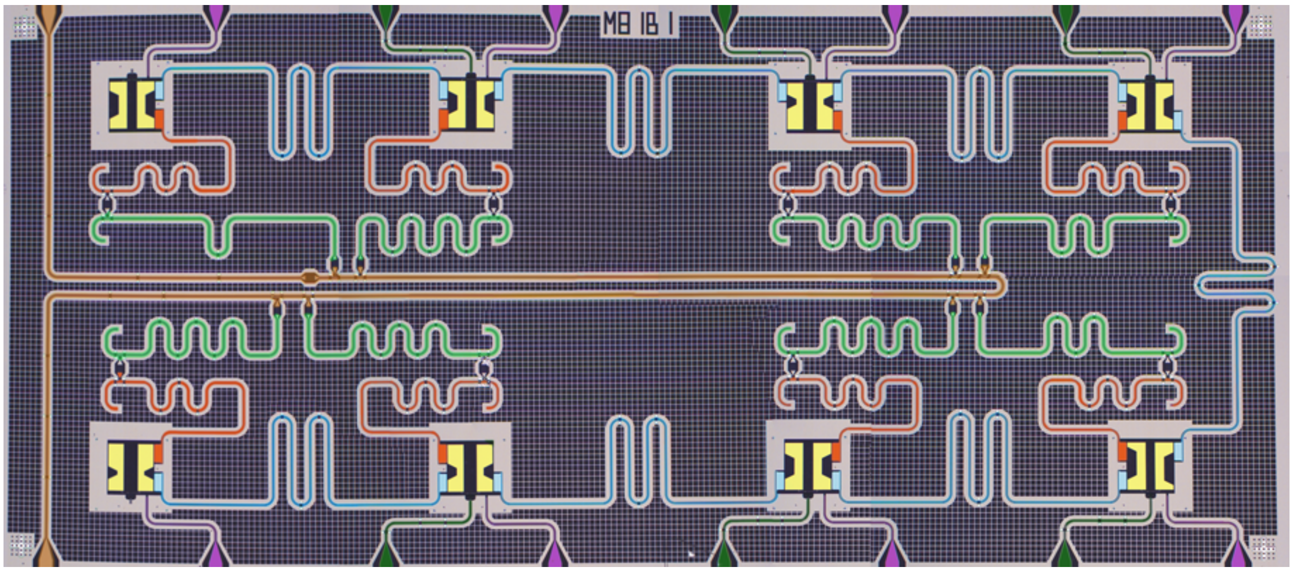


Figure 2.1: False-colored drawing of the 8-qubit chip used for most of the experiments presented in this thesis. See main text for details and Appendix B for a summary of the different samples used throughout this thesis.

The CW resonator is a two dimensional resonator which allows to access the information stored in the qubit in a non-destructive way through a quantum non-demolition (QND) measurement [14],[15]. The principles behind a QND measurement are explained in the next section. The resonator is also used to transmit quantum information between qubits on the chip. Thus, each of the eight qubits in Figure 2.1 is also capacitively coupled to its nearest neighbor(s) via the resonators in blue. This qubit-qubit coupling allows to perform two-qubit gates between nearest neighbors [16]. The reduced dimensionality of circuit QED gives this design a major advantage over three-dimensional architectures, such as cavity quantum electrodynamics(cQED) or Rydberg atoms, because the strength of the vacuum field inside the resonator increases with smaller volumes. This feature ensures a stronger coupling between the superconducting qubit and the resonator [14].

The purple lines in Figure 2.1 are the qubit *charge lines* which transmit the microwave pulses that drive each qubit. The *flux lines* shown in dark green are used to tune the qubit transition frequencies by applying a magnetic flux through the superconducting quantum interference device (SQUID) loop of each qubit (described below). *Purcell filters* (light green) sit between the readout resonators and the readout transmission line (brown), and protect the qubits from the Purcell decay ([17] and [18] present this effect in more detail).

The measurements presented in this thesis were performed on a *transmon* qubit, which is the improved version of the Cooper pair box (CPB), or charge qubit (details below). The transmon design for the eight qubits in Figure 2.1 is shown in Figure 2.2a, and the corresponding circuit diagram is illustrated in Figure 2.2b.

Both the transmon and the charge qubits exploit the superconducting properties of cooled electrons, which form Cooper pairs. The transmon has two superconducting islands (yellow rectangles in Figure 2.2a). The left island (Island1) capacitively couples to the readout resonator, and each island is connected to a qubit-qubit coupling resonator. Cooper pairs can tunnel between the islands via two Josephson junctions [20], which are non-linear inductors consisting of two superconducting areas separated by an insulating layer that permits Cooper pairs to tunnel between these areas [13]. A Josephson junction is described by the Josephson equations [13]:

$$I = I_c \sin\delta \tag{2.1}$$

$$V = \frac{\Phi_0}{2\pi}\dot{\delta} = \frac{\Phi_0}{2\pi I_0}\frac{1}{\cos\delta}\dot{I}, \tag{2.2}$$

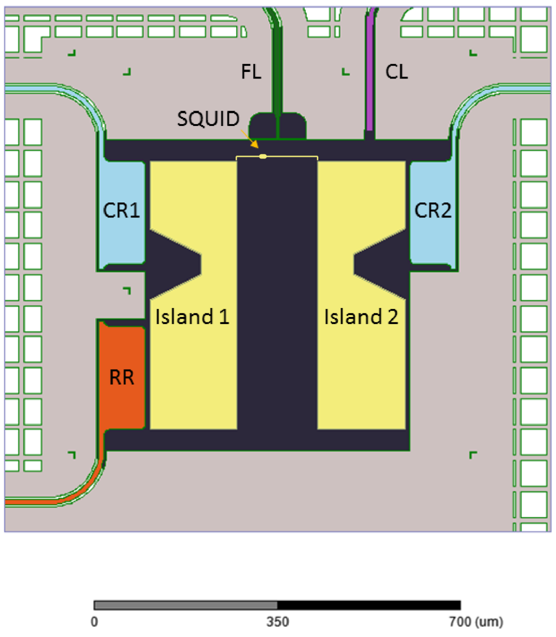
where I and V are the current and voltage in the junction, I_c is the junction critical current, $\Phi_0 = h/2e$ is the flux quantum, and $\delta = \delta_{Island1} - \delta_{Island2}$ is the difference between the phases of the Cooper pair wavefunctions in the two superconducting islands. The equations show that the Josephson junction is a non-linear element; it forms an anharmonic oscillator and thus gives the design its quantum properties.

For many quantum computing architectures it is important to be able to vary the critical current through the Josephson junction in order to tune the qubit transition frequencies. This feature is achieved by forming a SQUID loop out of two such junctions connected in parallel, as shown in Figure 2.2b. A magnetic flux applied through this SQUID loop effectively allows to change the qubit level separation by up to a few GHz. This degree of freedom becomes very useful for two-qubit gates implementations [21].

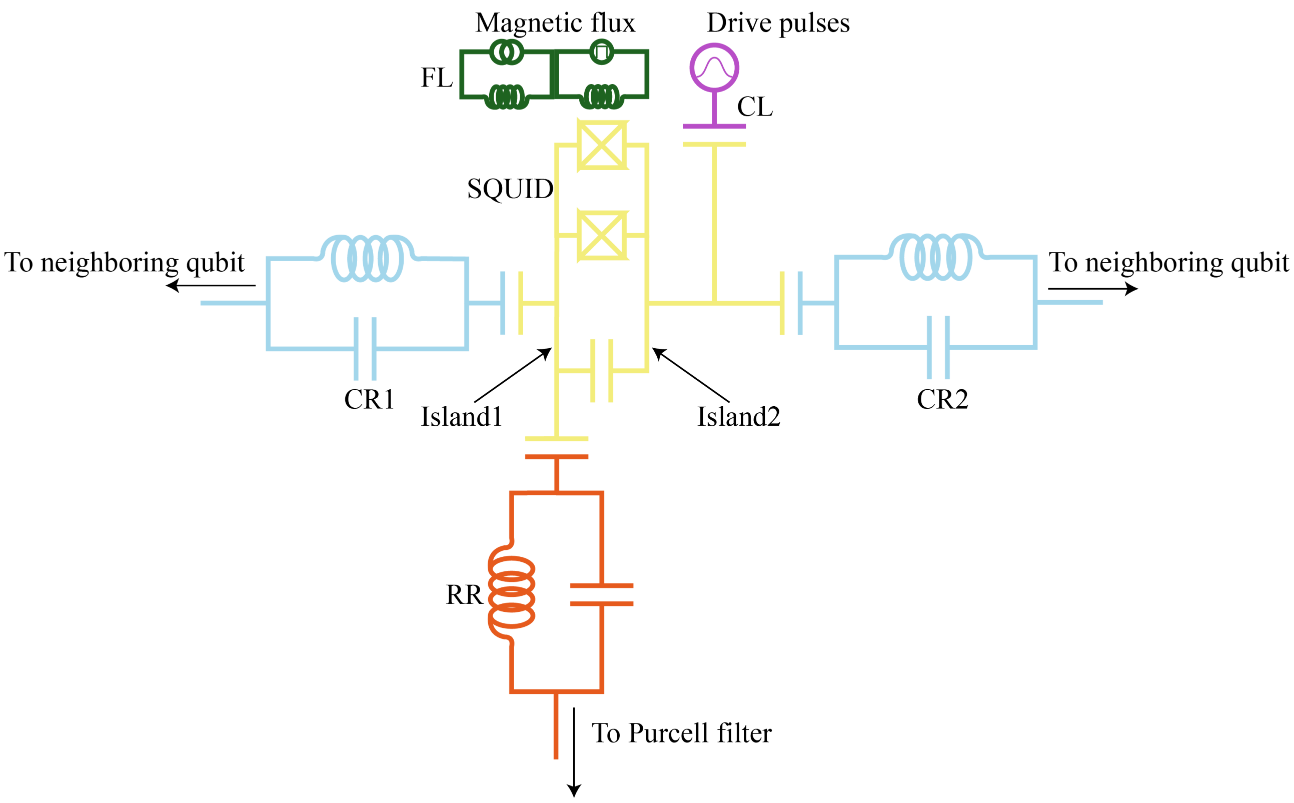
The charge qubit is completely characterized by the canonical variables δ and n, where the latter is the difference in the number of Cooper pairs on each island. The Hamiltonian of the Cooper pair box has an electrostatic contribution (from the capacitances in the box and external electric fields) and a magnetic contribution (from the potential energy of the Josephson junction) [13]:

$$H = H_{el} + H_{mag} = E_C(\hat{n} - n_g)^2 - E_{J_0}\cos\hat{\delta}. \tag{2.3}$$

In the equation above, $E_C = 4e^2/2C$ is the qubit *charging energy*, where C is the sum of all the capacitances in the box, $E_{J_0} = \hbar I_c/2e$ is the *specific Josephson energy* of the junction,



(a) Transmon



(b) Circuit diagram.

Figure 2.2: False-colored schematic (a) and circuit diagram (b) of the transmon qubit design used for the experiments presented in this work. The yellow SQUID line connecting the two islands was drawn in software. RR is the readout resonator, CR1 and CR2 are the coupling resonators between qubits, FL is the flux line that tunes the qubit frequencies, CL is the charge line used to drive the qubits; SQUID, Island1, and Island2 are explained in the text. Circuit diagram adapted from [19]

and n_g is the gate charge, which appears due to the voltage applied to the qubit control line. The magnetic term H_{mag} can be tuned by applying an external magnetic flux through the SQUID loop because $\delta \sim \Phi_{ext}$ [13]. The main difference between the Cooper pair box and the transmon is that the latter operates at a ratio $E_J/E_C > 50$, which is much higher than the value $E_J/E_C \sim 2$ used for the charge qubit [18]. This new regime is an enhancement in qubit design because it removes the dependence of the qubit energy levels on the gate charge, and it improves coherence by reducing the *charge dispersion* of the qubit energy levels[1] [18]. Increasing E_J/E_C also reduces the qubit *anharmonicity*, which is the difference in frequency between the second and first qubit transition frequencies resulting from the non-linear nature of the system. This reduction is undesirable because it makes it difficult to have good control over which of the qubit levels is being addressed. However, since the anharmonicity decreases algebraically with E_J/E_C, while the charge dispersion is reduced exponentially, the transmon offers a great advantage over the charge qubit at a reduced cost [18]. Furthermore, even this cost can be corrected for by designing qubit drive pulses that are optimized to couple to only one energy level (see Section 2.2.6).

2.1.2 Qubit-Resonator Interaction

For an N-level qubit, the coherent interaction between the CW resonator and the transmon qubit is described by the generalized Jaynes-Cummings Hamiltonian [18]:

$$\hat{H} = \sum_{i=0}^{N-1} \hbar\omega_i |i\rangle\langle i| + \hbar\omega_r \hat{a}^\dagger \hat{a} + \left[\sum_{i=0}^{N-2} \hbar g_{i,i+1} |i\rangle\langle i+1| \hat{a}^\dagger + \text{h.c.} \right], \tag{2.4}$$

where $\hat{a}^\dagger, \hat{a}$, are the creation and annihilation operators, and h.c. means Hermitian conjugate. For the above equation it was assumed that the rotating wave approximation (RWA) is valid[2], and that the ground state energy of the qubit is $E_0 = \hbar\omega_0 \neq 0$. The first term in Equation 2.4 represents the N energy levels of the transmon with frequencies ω_i, and the second term describes the resonator as a harmonic oscillator with frequency ω_r. The third, interaction term describes the energy exchange between the coupled qubit-resonator system: an excitation annihilated in the resonator is created in the qubit and vice versa. This interaction is characterized by the coupling strengths $g_{i,i+1}$ between the resonator and the i^{th} excitation of the qubit. g sets the angular frequency with which excitations are exchanged between the two systems. When g is larger than both the qubit decoherence rate (γ) and the rate of photon loss from the resonator (κ), the system is said to be in the *strong coupling regime*. If this regime cannot be achieved, then the system is not suitable for quantum information processing because the qubit and the resonator cannot coherently exchange an excitation before the system decoheres [20].

In quantum mechanics, measurement is intrusive and changes the state of the system being observed by projecting it into an eigenstate of the measurement operator. However, in QND measurements, the measurement operator commutes with the Hamiltonian of the system of interest, which in this case is the qubit. Thus, a QND measurement prevents mixing of the measured observable $\hat{\sigma}_z$ such that repeated measurements of the qubit will give the same results. For more details about QND measurements see [14] or [15]. In order to perform QND measurements of the qubit state, the system is placed in the *dispersive limit*, where the qubit and the resonator are far detuned from one another and $g_{i,i+1} \ll \Delta_i$ [15], [18]. In this limit, the qubit and the resonator do not exchange energy but they are still dispersively coupled to each other, such that the state of one has an influence on the state of the other, and vice versa.

[1]The charge dispersion refers to the sensitivity of the qubit energy level separations to fluctuations in the gate charge. These fluctuations are also referred to as shot noise.

[2]The RWA applies if $|\Delta_i| = |\omega_{i,i+1} - \omega_r| \ll \omega_{i,i+1} + \omega_r$ and $g_{i,i+1} \ll \omega_i, \omega_r$ [15].

This can be seen from the dispersive Hamiltonian, obtained by treating the interaction term in Equation 2.4 as a perturbation to the uncoupled Hamiltonian [13], and approximating the qubit by a two-level system with one excitation (but also including the effects of coupling to the second energy level):

$$\hat{H}_{\text{dispersive}} = \frac{\hbar\omega'_{01}}{2}\hat{\sigma}_z + \hbar(\omega'_r + \chi\hat{\sigma}_z)\hat{a}^\dagger\hat{a}. \tag{2.5}$$

The first term describes a qubit with renormalized transition frequencies $\omega'_{01} = \omega_{01} + \chi_{01}$. The qubit frequencies are shifted by the *Lamb shift* $\chi_{01} = g_{01}^2/\Delta_0$ produced by vacuum fluctuations inside the resonator. The second term is a dispersively shifted resonator with renormalized frequencies $\omega'_r = \omega_r - \chi_{12}/2$. The shift $\chi_{12}/2$ is caused by the interaction of the resonator with the second excited state of the qubit. $\chi = \chi_{01} - g_{12}^2/2\Delta_1 = \chi_{01} - \chi_{12}/2$ is the *dispersive shift* of the resonator, and is caused by coupling of the resonator to the qubit states. $\Delta_0 = \omega_{01} - \omega_r$ and $\Delta_1 = \omega_{12} - \omega_r$ are the detunings of the first and second qubit state frequencies from the resonator frequency, and $\hat{\sigma}_z$ is the Pauli Z operator. The second term in Equation 2.5 illustrates the principle of QND dispersive measurements: $\tilde{\omega}_r = \omega'_r + \chi\hat{\sigma}_z$, which can be thought of as the new resonator frequency, is pulled by the qubit, such that if the qubit is in the ground state $|g\rangle$ (first excited state $|e\rangle$), the resonator frequency is $\tilde{\omega}_r = \omega'_r + \chi$ ($\tilde{\omega}_r = \omega'_r - \chi$). This effect is illustrated in Section 2.2.2, Figure 2.4. Denoting $\hat{a}^\dagger\hat{a} = \hat{n}$, and rearranging this Hamiltonian in the following way

$$\hat{H}_{\text{dispersive}} = \frac{\hbar}{2}(\omega'_{01} + 2\chi\hat{n})\hat{\sigma}_z + \hbar\omega'_r\hat{a}^\dagger\hat{a}, \tag{2.6}$$

immediately shows the mirror effect of the dispersive shift, called the *AC-Stark shift*, whereby the effective qubit transition frequency $\tilde{\omega}'_{01} = \omega'_{01} + 2\chi\hat{n}$ changes depending on the number of photons n inside the resonator. Consequently, the dependence on n means that fluctuations in the photon number (*shot noise*) induce fluctuations of the qubit levels. This is a highly undesired effect as it induces dephasing in the qubit and hence loss of quantum information. However, this disadvantage can be overcome by placing the entire system in a vacuum and cooling it down to near zero kelvin. For more details about these phenomena, see [22]. All the effects described above can be observed during calibration measurements for single-qubit operations, which is the topic of the next section.

2.2 Single-Qubit Gates Calibration

In order for any quantum algorithm to be successful one must ensure that the quantum gates used in computation are well calibrated. Both single- and two-qubit gates are important in quantum computation, yet the current work will only focus on the former. Robust single-qubit gates are a prerequisite not only to implementing two-qubit gates, but also to characterizing general qubit performance; thus, developing software routines to calibrate and control single-qubit operations is a top priority.

A complete single-qubit calibration routine on the $|g\rangle \leftrightarrow |e\rangle$ transition follows these steps (adapted from [15]):

1. Find the readout resonator frequency ω_r by performing spectroscopy on the readout resonator.

2. Get an estimation of the qubit transition frequency ω_{ge} by performing spectroscopy on the qubit at the readout frequency determined in step 1.

3. Drive the qubit at the frequency found in step 2 in order to observe Rabi oscillations, and extract the first estimations of the π and $\pi/2$ drive pulse amplitudes by fitting the data to a cosine.

4. Perform a Ramsey measurement at a drive frequency that is intentionally detuned from the qubit frequency estimated in step 2 in order to find the true value of ω_{ge} from a fit to an exponentially decaying cosine.

5. Perform another Rabi measurement at the true qubit frequency found in step 4 and extract better values for the π and $\pi/2$ pulse amplitudes.

6. Perform a DRAG (derivative removal by adiabatic gate) pulse calibration measurement to find the optimal q_{scale} scaling factor such that the quadrature component of the drive field is $\epsilon_y(t) \propto q_{scale}\dot{\epsilon_x}(t)$.

7. Perform the Rabi measurement one last time to obtain the best values for the π and $\pi/2$ pulse amplitudes.

8. Measure the qubit energy relaxation time T_1.

9. Measure the averaged dephasing time $\mathrm{T}_2{}^\star$ for the qubit using a Ramsey measurement, and the qubit's real dephasing time T_2 using an Echo-Ramsey measurement. For details about the latter technique, see [23].

10. Validate the calibration routine by performing randomized benchmarking on the qubit. For details about this technique, see Section 2.5 in [13].

If one is also interested in the second excited state (qutrit), the calibration steps above must be repeated for the $|e\rangle \leftrightarrow |f\rangle$ transition, with a few minor differences, as explained in the remainder of this section. Calibration steps 1 through 8 and the measurement for the averaged dephasing time will be described in the remainder of this section, and the experimental results for these measurements are shown in Section 4.

2.2.1 Resonator Spectroscopy

In order to read out the qubit state, the best readout frequency of its corresponding readout resonator must be identified. Finding this frequency is not always easy, especially for a complex design like the chip in Figure 2.1 with eight resonators coupled to eight Purcell filters. The full transmission spectrum of this chip is shown in Figure 2.3. The transmission spectrum was obtained by driving all readout resonators with the same input signal applied at one end of the brown transmission line in Figure 2.1 and measuring the I and Q signal quadratures at the other end (see section 4.3 of [15] for details about the data acquisition process). Since any emitter driven on resonance will reflect most of the signal, a dip in the spectrum is expected whenever one of the resonances is reached. The best readout frequency for each qubit corresponds to the frequency point on the spectroscopic line of its readout resonator that is most sensitive to the dispersive shift discussed in Section 2.1.2. As seen in Figure 2.4 in the next subsection, a very good choice is typically the lowest point in the transmission spectrum since the effect of the dispersive shift subsides far away from the lowest point. For more details about resonator spectroscopy, see [24] and [13].

2.2.2 Qubit Spectroscopy

Qubit spectroscopy is performed in order to determine the approximate qubit frequencies, ω_{ge} and ω_{ef}. The frequency of the qubit drive signal is varied over a range of frequencies in order to measure the readout signal as a function of qubit drive frequency. The resulting graph will show a peak or dip at the qubit frequency depending on the chosen readout frequency. This is illustrated in Figure 2.4, which shows a measurement of the dispersive shift for readout

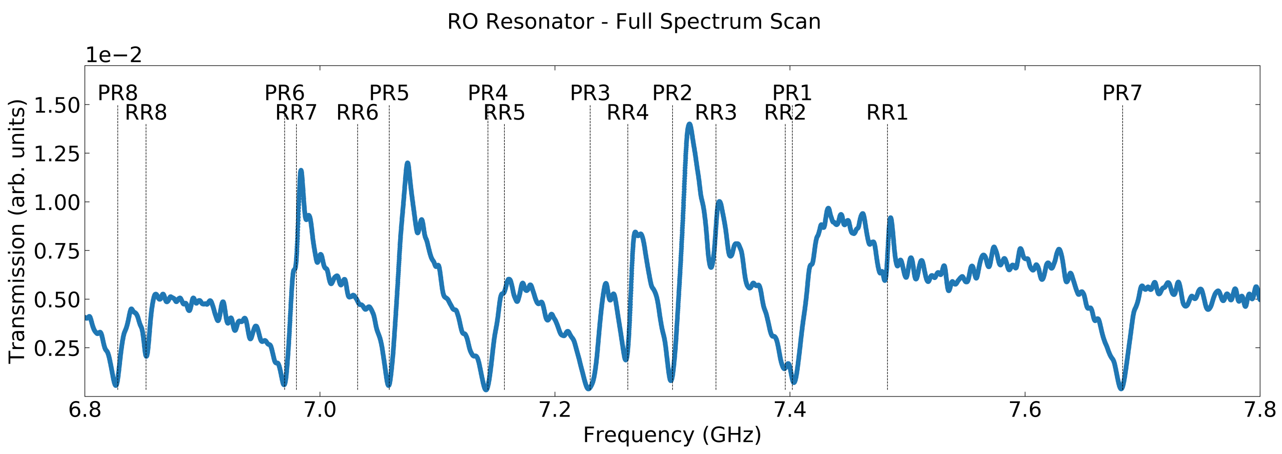


Figure 2.3: Transmission spectrum of all eight readout resonators (RR) and Purcell resonators (PR) on the chip shown in Figure 2.1. The graph was generated in Python, but the measurement was performed using the old LabVIEW setup. Plot is adapted from data measured by Johannes Heinsoo (QuDev) and Ants Remm (QuDev).

resonator 7 (see full readout spectrum in Figure 2.3), and two potential choices for the qubit readout frequency. For f_RO2, qubit spectroscopy will show a peak because the transmission power of the readout resonator is higher when the qubit is in the excited state (green) compared to when it is in the ground state (blue). A dip will appear for f_RO1, where the resonator shows a weaker response when the qubit is in the excited state.

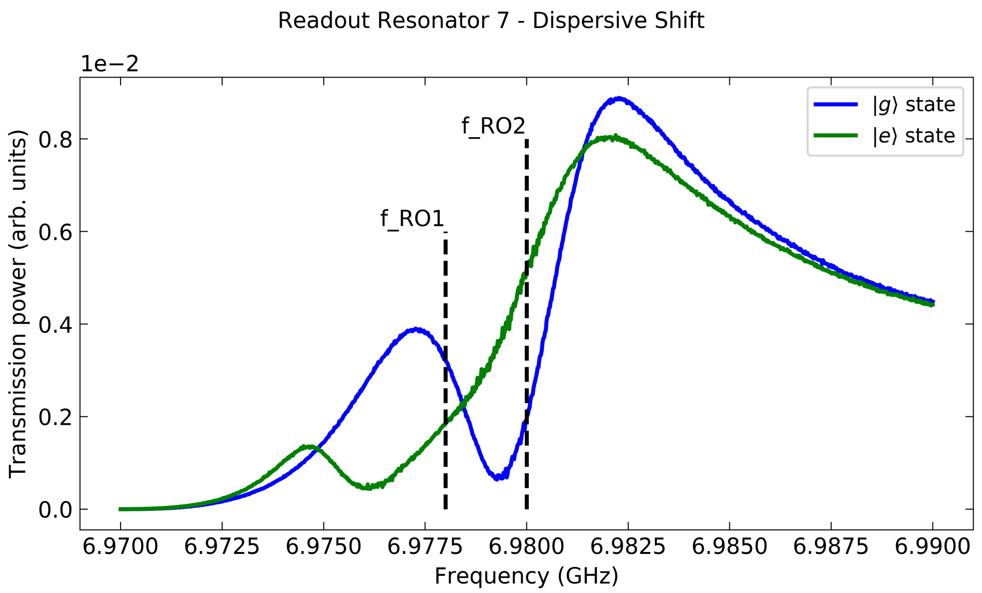
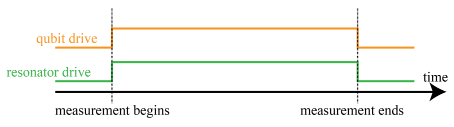
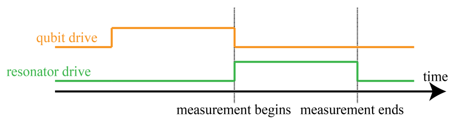


Figure 2.4: Measurement of the dispersive shift of RR7 in Figure 2.3. The transmission power spectrum is shown as a function of readout frequency when the qubit is in the ground state (blue) and in the excited state (green). The dashed lines indicate two potential choices for the readout frequency. For f_RO2, the qubit spectroscopy will show a peak, while for f_RO1 it will show a dip. See main text for details. The measurement was preformed by Ants Remm (QuDev).

Two types of qubit spectroscopy are commonly used to determine the qubit frequencies. In *continuous wave (cw)* spectroscopy (Figure 2.5a) both the qubit and the readout resonator are driven at the same time, while in *pulsed* spectroscopy (Figure 2.5b) the resonator drive is only turned on after the qubit drive has been turned off.



(a) Continuous wave



(b) Pulsed

Figure 2.5: Pulse sequences used for continuous wave (a) and pulsed (b) qubit spectroscopy. In the former, the qubit and the readout resonator are driven continuously and at the same time, while in the latter the readout signal is turned on only after the qubit drive has been switched off. See main text for details.

The downside of cw spectroscopy is that the number of photons inside the readout resonator will AC-Stark-shift the qubit frequencies during the qubit drive (see Section 2.1.2). Hence, this measurement must be done at low resonator drive power. This problem is avoided in pulsed spectroscopy because there are no photons in the resonator while the qubit is being driven (see [24] for details). In both types of spectroscopy, the qubit should be driven at low power to avoid power broadening the linewidth, which will lead to a decreased resolution of the qubit frequency [25]. However, to obtain values for both the $|g\rangle \leftrightarrow |e\rangle$ and $|e\rangle \leftrightarrow |f\rangle$ transition frequencies, the spectroscopic measurement must be performed at a higher qubit drive power than for regular spectroscopy. This higher power will stimulate a two-photon transition at $\omega_{gf/2}$, which will appear as a thinner spectroscopic line indicating the "$|gf/2\rangle$ transition." Then $\omega_{ef} = 2\omega_{gf/2} - \omega_{ge}$[3]. For more details about qubit spectroscopy, see [24] and [13].

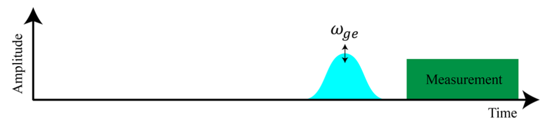
2.2.3 Rabi Measurement

By varying the amplitude of the qubit drive pulse, one can observe Rabi oscillations between two qubit energy levels. Since harmonic oscillators do not exhibit this behavior [25], observation of Rabi oscillations is a clear indication that the system being studied is indeed a qubit, i.e. an anharmonic system (see Section 2.1.1).

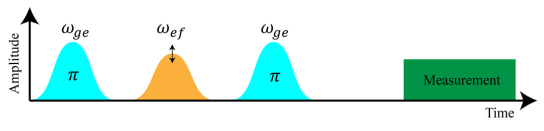
The Rabi experiment is also used to determine the π-pulse (the drive pulse amplitude that produces the first qubit population inversion; for example, taking the qubit from its $|g\rangle$ state to its $|e\rangle$ state), and the $\pi/2$-pulse (the drive pulse amplitude that brings the qubit in a superposition of two adjacent states). To do this, several pulses of varying amplitudes are applied to the qubit drive line, each followed by a measurement. The pulse sequences for a Rabi measurement on the $|g\rangle \leftrightarrow |e\rangle$ and the $|e\rangle \leftrightarrow |f\rangle$ transitions are illustrated in Figure 2.6a and Figure 2.6b, respectively. These procedures rotate the qubit around the x-axis (for a zero phase pulse) or the y-axis (90 degrees phase pulse) on the Bloch sphere, and map its final position onto

[3]The $|gf/2\rangle$ is really a virtual state in the two-photon process from $|g\rangle$ to $|f\rangle$. It is often referred to as the "$|gf/2\rangle$ state" because its frequency is somewhere around the value $\omega_{gf}/2$, and it is used as a reference for approximating the value of ω_{ef}. This two-photon process via a virtual state is different from two single photon transitions from $|g\rangle \leftrightarrow |e\rangle$ and $|e\rangle \leftrightarrow |f\rangle$ in that the virtual state is never populated. For more details about this type of process see section 8.4, and appendix E of [26].

the z-axis. At the end of the experiment, the probability for the qubit to be in the excited state as a function of the applied pulse amplitude will trace out a cosine. The π-pulse and the $\pi/2$-pulse amplitudes are then extracted from a fit to the data (see Section 4.3).



(a) Qubit



(b) Qutrit

Figure 2.6: Pulse sequences used for a Rabi measurement on the $|g\rangle \leftrightarrow |e\rangle$ (a) and $|e\rangle \leftrightarrow |f\rangle$ (b) qubit transitions. ω_{ge} and ω_{ef} are the $|g\rangle \leftrightarrow |e\rangle$ and $|e\rangle \leftrightarrow |f\rangle$ transition frequencies, respectively, π denotes a π-pulse at the respective frequency, and the vertical double arrow denotes a varying pulse amplitude. The time separation between the pulses is arbitrary, but must be shorter than the qubit energy relaxation time discussed in Section 2.2.5. Figures adapted from [8] and [13].

A Rabi measurement on the $|e\rangle \leftrightarrow |f\rangle$ transition requires an additional π-pulse at ω_{ge} at the beginning of the measurement in order to place the qubit population in the excited state. The additional $|g\rangle \leftrightarrow |e\rangle$ π-pulse shown in Figure 2.6b before each measurement is optional. This pulse maps the population back to the ground state, such that one effectively observes Rabi oscillations between $|g\rangle \leftrightarrow |f\rangle$. The disadvantage of using this procedure is that it relies on the assumption that the population in the $|e\rangle$ state is completely mapped to the $|g\rangle$ state by the $|g\rangle \leftrightarrow |e\rangle$ π-pulse, which is not a guarantee. The Rabi measurement on qubits and qutrits is explained in greater detail in [13].

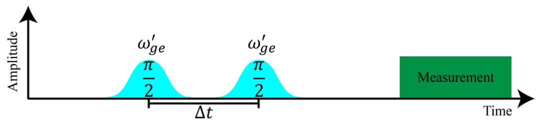
2.2.4 Ramsey Measurement

The Ramsey measurement identifies the true qubit transition frequency ω_{ge} by using the qubit frequency estimated with spectroscopy ω_{spec} (Section 2.2.2), which is typically slightly detuned from the real ω_{ge}. Figure 2.7a and Figure 2.7b illustrate the pulse sequences for a Ramsey measurement on a qubit and qutrit, respectively.

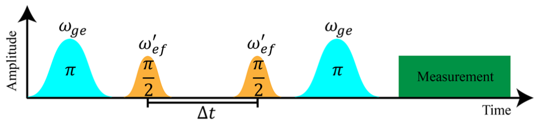
To measure the $|g\rangle \leftrightarrow |e\rangle$ transition frequency, two $\pi/2$-pulses are applied to the qubit at a frequency that is intentionally chosen to be detuned by δ (typically a few MHz) from ω_{spec}. Our lab usually uses $\delta = 4$ MHz. In PycQED, δ is called "artificial_detuning" because it is "artificially" introduced by the researcher during the measurement, and thus it is not an effect of the interactions inside the system. If we assume for the moment the ideal case where $\omega_{spec} = \omega_{ge}$, a calculation readily shows that the probability for the qubit to be in the excited state at the end of the measurement oscillates as a function of this artificial detuning δ and the time delay Δt:

$$P(|e\rangle) = \frac{1}{2}(1 + \cos(\Delta t \delta)). \tag{2.7}$$

In reality, $\omega_{spec} = \omega_{ge} - \delta'$, where δ' is the detuning from the real qubit frequency that we want to find. The result in this case will not oscillate at a frequency δ, but at $\omega_{Ramsey} = \delta - (-\delta')$.



(a) Qubit



(b) Qutrit

Figure 2.7: Pulse sequences used for a Ramsey measurement on the $|g\rangle \leftrightarrow |e\rangle$ (a) and $|e\rangle \leftrightarrow |f\rangle$ (b) qubit transitions. ω_{ge} is the $|g\rangle \leftrightarrow |e\rangle$ transition frequency, ω'_{ge} and ω'_{ef} are the applied qubit drive frequencies that are detuned from the $|g\rangle \leftrightarrow |e\rangle$ and $|e\rangle \leftrightarrow |f\rangle$ transition frequencies estimated with spectroscopy by a few MHz, π and $\pi/2$ denote a π-pulse and $\pi/2$-pulse at the respective frequencies, and Δt represents a variable time delay. The time separation between the pulses is arbitrary, but must be shorter than the qubit energy relaxation time discussed in Section 2.2.5. Figures adapted from [8] and [13].

Then, using that $\delta = |\omega_{pulse} - \omega_{spec}|$, the real qubit transition frequency can be recovered from:

$$\omega_{ge} = \omega_{spec} - \omega_{Ramsey} + \delta = \omega_{spec} - \delta'. \tag{2.8}$$

If the Ramsey measurement is now repeated, we are back to the case $\omega_{spec} = \omega_{ge}$, and a trace oscillating at δ will be observed.

The above procedure shows that performing a Ramsey measurement at one single artificial detuning cannot distinguish between $\omega_{spec} = \omega_{ge} \pm \delta'$ if $\delta' >> \delta$. The solution is to do two Ramsey measurements at two different artificial detunings, for example $\delta_1 = 4$MHz and $\delta_2 = -\ 4$MHz. Assuming that $\omega_{spec} = \omega_{ge} - \delta'$, we test the four cases

$$\omega_{ge} = \omega_{spec} + \omega_{Ramsey1} + \delta_1 = \omega_{spec} + 2\delta_1 + \delta' \tag{2.9}$$
$$\omega_{ge} = \omega_{spec} - \omega_{Ramsey1} + \delta_1 = \omega_{spec} - \delta' \tag{2.10}$$
$$\omega_{ge} = \omega_{spec} + \omega_{Ramsey2} + \delta_2 = \omega_{spec} + 2\delta_2 + \delta' \tag{2.11}$$
$$\omega_{ge} = \omega_{spec} - \omega_{Ramsey2} + \delta_2 = \omega_{spec} - \delta'. \tag{2.12}$$

The results of these four cases show that the real qubit frequency will be either $\omega_{spec} + 2\delta_{1,2} + \delta'$ or $\omega_{spec} - \delta'$. The correct qubit frequency will be given by the two cases above that produce the same result. In this theoretical example, it is easy to see that we indeed recover the correct answer, $\omega_{spec} - \delta'$, from 2.10 and 2.12. In practice, two of these four cases will always give the same result, which will be the correct qubit frequency (see Section 4.4).

As explained for the Rabi case in Section 2.2.3 and shown in Figure 2.7b, a Ramsey measurement on a qutrit requires an additional $|g\rangle \leftrightarrow |e\rangle$ π-pulse at the start and optionally one right before the measurement. Then the procedure described above is identical, with ω_{ge} replaced by ω_{ef}.

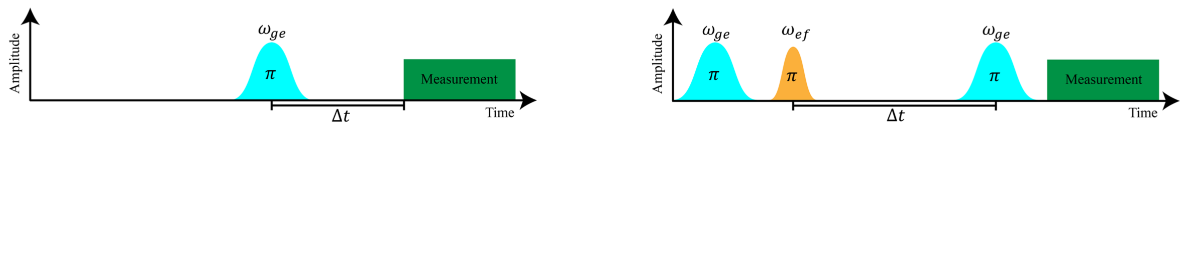
The Ramsey measurement is also used to find the averaged dephasing time $T_2{}^{\star}$[4]. The Ramsey oscillations will have an exponentially decaying envelope proportional to $e^{-\Delta t/T_2^*}$ due to the loss of phase information (dephasing) from the qubit during the measurement. The

[4]The distinction between the averaged ($T_2{}^{\star}$) and pure (T_2) dephasing times is explained in [8].

dephasing time can be easily extracted from a simple exponential envelope to the cosine fit function described above (see Section 4.4). More information about the Ramsey measurement can be found in [13].

2.2.5 T$_1$ Measurement

A measurement of the energy relaxation time is done using the pulse scheme shown in Figure 2.8a for a qubit, and in Figure 2.8b for a qutrit.



(a) Qubit (b) Qutrit

Figure 2.8: Pulse sequences used for a T$_1$ measurement on the $|g\rangle \leftrightarrow |e\rangle$ (a) and $|e\rangle \leftrightarrow |f\rangle$ (b) qubit transitions. ω_{ge} and ω_{ef} are the $|g\rangle \leftrightarrow |e\rangle$ and $|e\rangle \leftrightarrow |f\rangle$ transition frequencies, respectively, π denotes a π-pulse at the respective frequencies, and Δt represents a variable time delay. The time separation between the pulses is arbitrary, but must be shorter than the qubit energy relaxation time. Figures adapted from [8] and [13].

An excited system will decay to its thermal equilibrium state when left to relax in time. Thus, in order to measure this energy decay time (T$_1$) for a qubit, one applies a $|g\rangle \leftrightarrow |e\rangle$ π-pulse, and waits for a variable time delay Δt before measurement. The probability for the qubit to be in the excited state will show an exponential decay proportional to $e^{-\Delta t/T_1}$, and T$_1$ can be easily extracted from a fit to the data (see Section 4.5).

As explained in the previous two sections and shown in Figure 2.8b, in a T$_1$ measurement on the second excited state a $|g\rangle \leftrightarrow |e\rangle$ π-pulse must be applied first in order to place the population in the excited state, and an optional π-pulse can be applied before the measurement to map the population back to the ground state. Reference [13] offers more details about the T$_1$ measurement on qubits and qutrits.

2.2.6 DRAG Pulse Calibration

The DRAG pulse is an optimized qubit drive pulse that prevents leakage outside of the computational subspace, and phase errors due to coupling to neighboring energy levels.

A Gaussian pulse is generally preferred over a square-shaped pulse because it has a narrower frequency spectrum, and thus allows for better qubit control by reducing leakage into the $|f\rangle$ state [15]. A higher standard deviation for the Gaussian pulse in time domain makes it easier to target the qubit $|g\rangle \leftrightarrow |e\rangle$ transition. However, this width is limited by the qubit energy relaxation time since the qubit will decay during the operation if the pulse is longer than T$_1$. Moreover, even after controlling for these limitations, the Gaussian pulse's width in frequency domain may include the qubit $|e\rangle \leftrightarrow |f\rangle$ transition (this is especially likely for designs with relatively low anharmonicity). This scenario is very disadvantageous because it limits the ability to accurately manipulate the qubit. Luckily, the DRAG pulse can be calibrated to have a vanishing frequency component at the $|e\rangle \leftrightarrow |f\rangle$ transition, as illustrated in Figure 2.9.

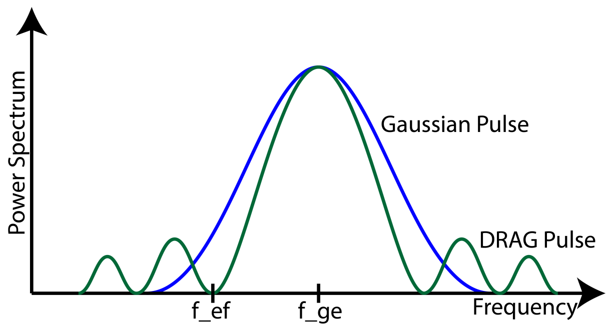


Figure 2.9: Schematic comparison between a regular Gaussian qubit drive pulse (blue) and a DRAG qubit drive pulse (green). The latter suppresses the unintended excitation of the qubit $|f\rangle$ state.

For a drive field expressed in quadrature form [15]:

$$\epsilon(t) = \epsilon_I(t)\cos(\omega_d t) + \epsilon_Q(t)\sin(\omega_d t), \tag{2.13}$$

with ϵ_I and ϵ_Q the in-phase and quadrature field components, and ω_d the drive frequency, the first order DRAG pulse is defined as [27]

$$\epsilon_I(t) = \epsilon_{Gauss}(t) \tag{2.14}$$

$$\epsilon_Q(t) = -\frac{\lambda^2}{4\Delta_{12}}\dot{\epsilon}_{Gauss}(t) = q_{scale}\dot{\epsilon}_{Gauss}(t). \tag{2.15}$$

In the two equations above, ϵ_{Gauss} is a Gaussian-shaped pulse, λ is the scaling factor from reference [27], $\Delta_{12} = \omega_{ef} - \omega_{ge}$ is the anharmonicity between the first two qubit levels, and $q_{scale} = -\lambda^2/4\Delta_{12}$ is the quadrature scaling factor that must be calibrated.

To find the value of q_{scale} that achieves the effect in Figure 2.9, the parameter is swept between ∼ -1.5 and ∼ 1.5, and the following three sets of pulses are applied for each q_{scale} value: $X_{\frac{\pi}{2}}X_\pi$, $X_{\frac{\pi}{2}}Y_\pi$, and $X_{\frac{\pi}{2}}Y_{-\pi}$, where X and Y denote the rotation axes about which the π- and $\pi/2$-pulses are applied. Since the π-pulse amplitude was previously calibrated with a Rabi measurement, the first set of pulses leave the qubit in an equal superposition of $|g\rangle$ and $|e\rangle$ states. Ideally, the other two sets of pulses should do the same, but because of phase errors introduced as a result of leakage outside of the computational subspace, the last two sets cause the qubit to move away from the equatorial plane on the surface of the Bloch sphere to a mixed state inside the sphere. Hence, as shown in the plots in Section 4.6, the measurement data of the $X_{\frac{\pi}{2}}X_\pi$ set traces out a constant line at 0.5 qubit population, and those of the other two sets trace out an increasing and a decreasing line, respectively. Since, in theory all three lines should intersect at an equal superposition of the qubit states, the q_{scale} value of this intersection point is the optimal q_{scale} parameter.

For a qutrit, the DRAG calibration requires an extra $|g\rangle \leftrightarrow |e\rangle$ π-pulse at the beginning and an optional one before each measurement. For more details about the DRAG pulse and the q_{scale} calibration see [15],[13],[7], and [27].

3. The Python for Circuit Quantum Electrodynamics (PycQED) Framework

3.1 Introduction

Integrated measurement and analysis software frameworks become more desirable as the experimental setups for developing quantum computers scale up and become more complex. Such cohesive frameworks offer several benefits over setups that employ different software packages for measurement control and data analysis. In particular, the communication between different stages of a measurement is improved because it is easier and less error-prone to develop and optimize a control software that must only access parameters and operations implemented within a single software environment.

Another feature that becomes desirable, especially as the number of qubits in quantum computer designs scales up, is a framework based on object-oriented languages. This section will show that working with objects has the great advantage that each qubit or instrument within the software framework is a logical entity (called an object), and thus it is much easier to manipulate and control. Since this encapsulation into objects creates a logically organized code structure, it greatly facilitates both the development of new features in the framework and the debugging process. Furthermore, object-oriented programming also reduces the risk of errors by using special functions called constructors and destructors [12], [28]. The former ensure that a new object instance is initialized correctly, while the latter ensure that all communications with instruments are properly closed and that the relevant information in memory is released upon the instance's destruction.

PycQED offers such an object-oriented integrated measurement environment designed in Python 3 and implemented specifically for circuit QED experiments. The PycQED project is a recent development by Leonardo DiCarlo's group at Delft University of Technology, Netherlands, and is licensed under the MIT License. PycQED is an extension of the QuCoDeS data-acquisition framework created by the Copenhagen-Delft-Sydney-Microsoft quantum computing consortium. QuCoDeS has most drivers and necessary support for communicating with the physical instruments, and contains the algorithms for defining the parameters in the PycQED meta-instrument classes, which will be described in this section.

In 2016, our group at QuDev have started a joint collaboration effort with the DiCarlo group to expand and improve the PycQED framework. At the start of our collaboration, the main PycQED skeleton was already functional and in a high stage of development. The main focus of the contribution presented in this work was on adding and improving single-qubit measurements and analysis routines, with the goal to make available the possibility to run all the single-qubit gates calibration steps enumerated in Section 2.2 for both the $|g\rangle \leftrightarrow |e\rangle$ and the $|e\rangle \leftrightarrow |f\rangle$ transitions. Many of these measurement and analysis algorithms were already partially or completely implemented at the start of this project. Yet, some of these routines, albeit functional, were either not producing the results that our team was interested in, or were not fully integrated into the framework flow. For example, the fitting routines would sometimes fail in some of the analysis classes for single-qubit operations, or a few parameters of interest to us, such as the π-pulse and $\pi/2$-pulse amplitudes, were not being calculated. Another example is that even though some of the arbitrary waveform generator (AWG) sequences for operations on the $|e\rangle \leftrightarrow |f\rangle$ transition were functional, they were not completely integrated into the higher levels of the code structure, and hence could not be called during an experiment.

In the remainder of this section, the PycQED structure will be presented in Section 3.2, and then Section 3.3 will give a more practical description of how to use PycQED to run circuit QED measurements. In Section 3.2.2 it is explained in more detail how PycQED controls a measurement, from the pulse designs and instrument configurations, to data acquisition using the Zurich Instruments Ultra High Frequency Lock-In (UHFLI) amplifier, and Section 3.2.3 presents the available data analysis routines. Overall, the main focus will be on the improvements made to the measurement and analysis routines for single-qubit operations, which were the main contributions of this work to the PycQED collaboration.

3.2 Framework Structure

PycQED makes heavy use of object-oriented programming features. This section will give technical details about how the objects in PycQED are interconnected and how objects inherit from one another. An object-oriented program works with logical entities called objects, which are defined by their classes. Hence, an object/class is comprised of parameters, functions (called its methods), and even other objects that describe the operation of a specific concept. For example, the most generic qubit object would contain qubit parameters such as its transition frequencies or its energy relaxation and dephasing times, but it would also contain methods that allow the user to access the qubit parameters, or to manipulate the qubit by driving it with microwave pulses. The qubit objects in PycQED follow this template and will be discussed in Section 3.2.1. PycQED also makes extensive use of the concept of inheritance. When an object inherits from another object (called the base object), the former can access and use all[1] the parameters and functions of the latter. Further details about these features can be found in [12] and [28].

The three main objects in PycQED that control the qubit characterization measurements are the Qubit object, the MeasurementControl object, and the MeasurementAnalysis object. The Qubit object is called directly by the user at the start of each measurement. All the relevant measurement parameters, like the array of pulse amplitudes for a Rabi experiment or the number of calibration points for the measurement (discussed in Section 3.2.3), are passed to this object, which then proceeds to pass them to the MeasurementControl, to the sweep functions, and to the measurement analysis classes.

In the following sections we will look more closely at the Qubit object in Section 3.2.1, then at the role of the MeasurementControl and its dependencies in Section 3.2.2, and finally at the structure of the MeasurementAnalysis class and its dependencies in Section 3.2.3. Throughout the text, the following convention is used in order to reduce confusion: the sans serif font family is used for classes, methods, and functions, *italics* is used for naming conventions of various parameters inside PycQED, and all file names are followed by the ".py" extension. For simplicity and for a clearer understanding of the highly layered PycQED structure, important parts of the source code will be exemplified, wherever appropriate, on a typical Rabi measurement.

[1]This is not true in general because inheritance cannot access the private variables of a class (see [28] for details). This thesis will discuss in detail only the inheritance structure of the measurement analysis classes for which the statement in the main text is correct.

3.2.1 The Qubit Object

The Qubit object used in our lab is the QuDev_transmon (QuDev_transmon.py), and it inherits from the Qubit object base class developed by the DiCarlo group (qubit_object.py). All qubit objects are meta-instruments. This means that they control the lower-level QuCoDeS instrument classes, which are the ones that communicate directly with the physical instruments in the lab.

As mentioned above, QuDev_transmon stores all the parameters that characterize one qubit, but it also stores the parameters that describe qubit operations. Typical operations that all qubits must have are listed below:

Readout (*RO*) The readout operation with the readout signal parameters, such as pulse type, pulse amplitude, pulse length, modulation frequency, and other parameters of the microwave pulses used for reading out the qubit state.

Spectroscopy (*Spec*) The qubit spectroscopy operation with the spectroscopy signal parameters, such as pulse type, pulse amplitude, pulse length, pulse delay, and other parameters of the microwave pulses used for qubit spectroscopy.

X180* and *X180_ef The qubit and qutrit drive pulse parameters (*pulse_pars*), such as the π-pulse and $\pi/2$-pulse amplitudes, q_{scale} factors, pulse modulation frequencies, standard deviation of the Gaussian envelope of the pulses, and other parameters of the microwave pulses used for driving qubits.

All operations and operation parameters pertaining to a qubit can be accessed by calling the get_operations_dict method. A possible output of calling this method is shown in Listing 3.1. All the qubit operation parameters discussed above and illustrated in Listing 3.1 are objects of the QuCoDeS Parameter class. They are created during the qubit initialization routine with the add_parameter method of the QuCoDeS Instrument class.

```
'RO qb': {'I_channel': '0',
          'Q_channel': '1',
          'RO_pulse_marker_channel': 'AWG1_ch3_marker2',
          'acq_marker_channel': 'AWG1_ch3_marker2',
          'acq_marker_delay': 1e-07,
          'amplitude': 0.1,
          'length': 2.2e-06,
          'mod_frequency': 25000000.0,
          'operation_type': 'RO',
          'phase': 0,
          'pulse_delay': 0,
          'pulse_type': 'MW_IQmod_pulse_UHFQC',
          'target_qubit': 'qb'},
'Spec qb': {'amplitude': 1,
            'channel': None,
            'length': 1e-05,
            'operation_type': 'MW',
            'pulse_delay': 1e-05,
            'pulse_type': 'SquarePulse',
            'target_qubit': 'qb'},
'X180 qb': {'I_channel': None,
            'Q_channel': None,
            'alpha': 1,
            'amp90_scale': 0.5,
            'amplitude': 1,
            'amplitude_90': 0.5,
            'mod_frequency': 100000000.0,
            'motzoi': 0,
            'nr_sigma': 6,
            'operation_type': 'MW',
            'phase': 0,
            'phi_skew': 0,
            'pulse_delay': 0,
            'pulse_type': 'SSB_DRAG_pulse',
            'sigma': 2e-08,
            'target_qubit': 'qb'}
```

Listing 3.1: Output of calling the get_operations_dict method of the QuDev_transmon class. See main text for details.

The methods in QuDev_transmon are the functions that the user calls to perform a measurement. The methods in this class use the following prefix convention:

measure_ These methods only perform the measurement, without altering any of the qubit parameters, or doing any data analysis specific for the measurement. Some examples of these routines are measure_ramsey, measure_T1, measure_rabi (the last function is described below).

find_ These methods perform the measurement by calling the *measure_* methods, they perform the relevant data analysis, and they can update the relevant qubit parameters based on the analysis results. Some examples include find_resonator_frequency, find_T1, find_amplitudes (the last function is described below).

calibrate_ Currently, these methods have the same functionality as the *find_* methods, but use a different name for clarity. Examples are calibrate_ramsey, which is described in Section 4.4, and calibrate_readout_weights.

calculate_ These methods do not perform a measurement. They use already-existing qubit parameters to calculate new ones. In the current PycQED structure, the only examples are calculate_anharmonicity, which is described below, and calculate_EC_EJ.

All the *measure_* methods have the same general structure illustrated in Listing 3.2 for measure_rabi. These methods prepare the instruments for the type of measurement that is about to be performed (line 33), then they send the information about what is being measured to the MeasurementControl object (lines 41-47), which runs the experiment (line 48). Finally, at the end, they instantiate a MeasurementAnalysis object with the relevant parameters from the user (line 52).

The method that uses measure_rabi to extract the π-pulse and $\pi/2$-pulse amplitudes from a Rabi measurement was developed during this project and is called find_amplitudes. Its source code is shown in Listing 3.3. This function is a *find_* method, and hence it calls the correct *measure_* method to perform a measurement (line 49 for a qubit, and line 56 for a qutrit), analyzes the data using the correct class from measurement_analysis.py (line 65), and at the end updates the relevant qubit parameters (lines 74-79) if the user wants to do so ($update$ = true).

Finally, an example *calculate_* method is calculate_anharmonicity, shown in Listing 3.4. This function simply subtracts the frequency of the first transition from the frequency of the second transition.

```
1  def measure_rabi(self, amps=None, MC=None, analyze=True,
2                   close_fig=True, cal_points=True, no_cal_points=2,
3                   upload=True, label=None,  n=1):
4
5      """
6      Varies the amplitude of the qubit drive pulse and measures the readout
7      resonator transmission.
8
9      Args:
10         amps            the array of drive pulse amplitudes
11         MC              the MeasurementControl object
12         analyse         whether to create a (base) MeasurementAnalysis
13                         object for this measurement; offers possibility to
```

```
                        manually analyse data using the classes in
                        measurement_analysis.py
        close_fig       whether or not to close the default analysis figure
        cal_points      whether or not to use calibration points
        no_cal_points   how many calibration points to use
        upload          whether or not to upload the sequence to the AWG
        label           the measurement label
        n               the number of times the drive pulses with the same
                        amplitude should be repeated in each measurement
    """

    if amps is None:
        raise ValueError("Unspecified amplitudes for measure_rabi")

    # Define the measurement label
    if label is None:
        label = 'Rabi-n{}'.format(n) + self.msmt_suffix

    # Prepare the physical instruments for a time domain measurement
    self.prepare_for_timedomain()

    # Define the MeasurementControl object for this measurement
    if MC is None:
        MC = self.MC

    # Specify the sweep function, the sweep points,
    # and the detector function, and run the measurement
    MC.set_sweep_function(awg_swf.Rabi(pulse_pars=self.get_drive_pars(),
                                       RO_pars=self.get_RO_pars(), n=n,
                                       cal_points=cal_points,
                                       no_cal_points=no_cal_points,
                                       upload=upload))
    MC.set_sweep_points(amps)
    MC.set_detector_function(self.int_avg_det)
    MC.run(label)

    # Create a MeasurementAnalysis object for this measurement
    if analyze:
        ma.MeasurementAnalysis(auto=True, close_fig=close_fig)
```

Listing 3.2: Source code for the measure_rabi method of the QuDev_transmon class. awg_swf.Rabi is the class defining the Rabi sequence inside the awg_sweep_functions.py module, and *ma* is the measurement_analysis.py module. The input parameters are explained in the grey text.

```
def find_amplitudes(self, rabi_amps=None, label=None, for_ef=False,
                    update=False, MC=None, close_fig=True, cal_points=True,
                    no_cal_points=None, upload=True, last_ge_pulse=True,
                    analyze=True, **kw):

    if not update:
        logging.warning("Does not automatically update the qubit pi and "
                        "pi/2 amplitudes. "
                        "Set update=True if you want this!")

    if MC is None:
        MC = self.MC

    if (cal_points) and (no_cal_points is None):
        logging.warning('no_cal_points is None. Defaults to 4 is for_ef==False,'
                        'or to 6 is for_ef==True.')
        if for_ef:
            no_cal_points = 6
        else:
            no_cal_points = 4

    if not cal_points:
        no_cal_points = 0

    # How many times to apply the Rabi pulse
    n = kw.get('n',1)

    if rabi_amps is None:
        amps_span = kw.get('amps_span', 1.)
        amps_mean = kw.get('amps_mean', self.amp180())
        nr_points = kw.get('nr_points', 30)
        if amps_mean == 0:
            logging.warning("find_amplitudes does not know over which "
                            "amplitudes to do Rabi. Please specify the "
                            "amps_mean or the amps function parameter.")
            return 0
        else:
            rabi_amps = np.linspace(amps_mean - amps_span/2, amps_mean +
                                    amps_span/2, nr_points)

    if label is None:
        if for_ef:
            label = 'Rabi_2nd' + self.msmt_suffix
        else:
            label = 'Rabi' + self.msmt_suffix

    # Perform Rabi
    if for_ef is False:
        self.measure_rabi(amps=rabi_amps, n=n, MC=MC,
                          close_fig=close_fig,
                          label=label,
                          cal_points=cal_points,
                          no_cal_points=no_cal_points,
                          upload=upload)
    else:
        self.measure_rabi_2nd_exc(amps=rabi_amps, n=n, MC=MC,
                                  close_fig=close_fig, label=label,
                                  cal_points=cal_points,
                                  last_ge_pulse=last_ge_pulse,
                                  no_cal_points=no_cal_points,
                                  upload=upload)

    # Get pi and pi/2 amplitudes from the analysis results
    if analyze:
        RabiA = ma.Rabi_Analysis(label=label, NoCalPoints=no_cal_points,
                                 close_fig=close_fig, for_ef=for_ef,
                                 last_ge_pulse=last_ge_pulse, **kw)

        rabi_amps = RabiA.rabi_amplitudes
        amp180 = rabi_amps['piPulse']
        amp90 = rabi_amps['piHalfPulse']

        if update:
            if for_ef is False:
                self.amp180(amp180)
                self.amp90_scale(amp90/amp180)
            else:
                self.amp180_ef(amp180)
                self.amp90_scale_ef(amp90/amp180)
    else:
        return
```

Listing 3.3: Source code for the find_amplitudes method of the QuDev_transmon class. In the input parameters, *for_ef* specifies whether to perform the measurement and analysis for the second qubit excitation, *update* specifies whether to update the relevant qubit parameters, *cal_points* specifies whether to use calibration points or not, *no_cal_points* specifies the number of calibration points to use, *upload* specifies whether to upload the created sequence to the AWG, *last_ge_pulse* specifies whether to apply a $|g\rangle \leftrightarrow |e\rangle$ π-pulse at the end of each sequence for qutrit measurements (Section 2.2), and *analyze* specifies whether to perform the data analysis. The role of calibration points will be described in Section 3.2.3. measure_rabi_2nd_exc performs the Rabi measurement on the $|e\rangle \leftrightarrow |f\rangle$ transition, and *ma* is the measurement_analysis.py module. "kw" means "keyword arguments," and it allows functions to receive an arbitrary number of input keyword arguments. For example, all the input parameters to measure_rabi are passed in as keyword arguments.

```python
def calculate_anharmonicity(self, update=False):

    if not update:
        logging.warning("Does not automatically update the qubit anharmonicity "
                        "parameter. Set update=True if you want this!")

    if self.f_qubit() == 0:
        logging.warning('f_ge = 0. Run qubit spectroscopy or Ramsey.')
    if self.f_ef_qubit() == 0:
        logging.warning('f_ef = 0. Run qubit spectroscopy or Ramsey.')

    anharmonicity = self.f_ef_qubit() - self.f_qubit()

    if update:
        self.anharmonicity(anharmonicity)

    return  anharmonicity
```

Listing 3.4: Source code for the calculate_anharmonicity method of the QuDev_transmon class. *f_qubit* and *f_ef_qubit* are the $|g\rangle \leftrightarrow |e\rangle$ and $|e\rangle \leftrightarrow |f\rangle$ transition frequencies.

3.2.2 Measurement Control Flow

This section presents the interdependence between the most important classes necessary to run a measurement: MeasurementControl, detector functions, and sweep functions. The first part describes the communication between these objects during a measurement, and then the rest of the section looks more closely at a particular category of sweep functions, the AWG sequences, and explains how they are created in PycQED.

The MeasurementControl Object and its Dependencies

The MeasurementControl (MC) is the meta-instrument class that is in charge of running an experiment. It passes the correct instructions to the physical instruments before and during the measurement, and saves the acquired data in Hierarchical Data Format (HDF) (described at the end of this section). For each measurement, the MC needs to know from the qubit object the following three items:

Sweep Function An object that sets the parameters which will be swept over during the measurement. These sweep parameters are set in a format that is expected by the MC. Examples: all the AWG sweep functions classes defined in the module awg_sweep_functions.py, and discussed later in this section.

Sweep Points An array of values for the parameter that will be swept over in the measurement. Examples: the drive pulse amplitudes for Rabi oscillations, or the delay times for a Ramsey experiment.

Detector Function A meta-instrument that is in charge of the data acquisition process by communicating directly with the acquisition device. Examples: all the classes in the module detector_functions.py

From Listing 3.2 we see that for a Rabi measurement the sweep function (line 41) was the Rabi class from awg_sweep_functions.py (this function is described in more detail below), and the detector function (line 47) was int_avg_det. There are two types of sweep and detector functions: *hard*(ware-controlled) and *soft*(ware-controlled). A measurement must contain only one of these types for both sweep and detector functions, i.e. one cannot use, for example, a *soft* sweep and a *hard* detector.

The *hard* functions prepare the measurement which is then entirely controlled by the physical hardware in the lab. At the end of each measurement, the acquired data is returned to PycQED. For example, the QuDev_transmon function int_avg_det used by the measure_rabi method is an instance of the *hard* detector object UHFQC_integrated_average_detector defined in detector_functions.py. This object performs an integrated average of the readout data using the UHFLI.

The *soft* functions control the entire measurement themselves. They define a sweep point on an instrument and use another instrument to measure it. The measure_(qubit)spectroscopy method, for instance, uses the Heterodyne_probe *soft* detector, an object that in turn controls the HeterodyneInstrument. The latter is a meta-instrument that communicates with multiple physical instruments (microwave generators, AWGs etc.) defined by the user when an instance of the qubit object is created.

In the current PycQED version, the naming convention for *homodyne* and *heterodyne* does not necessarily refer to the experimental setups that these names conventionally describe. In a homodyne acquisition setup, the mixer that upconverts the signal for driving the readout resonator and the mixer that downconverts the output signal use the same local oscillator (LO) signal. In a heterodyne scheme, the LO signals for the upconversion (UC) and downconversion (DC) mixers are provided by different microwave generators (see section 4.3 in [15] for details about IQ modulation). In PycQED, for example both measure_spectroscopy and measure_resonator_spectroscopy use the Heterodyne_probe *soft* detector, to which they pass the *self.heterodyne* instrument object defined in QuDev_transmon. Thus it appears as if this measurement should only be used with the heterodyne setup. Yet this is not necessarily so; the two spectroscopy routines also work for a homodyne setup, because the *self.heterodyne* parameter can actually be any (meta)-detector that is passed in by the user when the qubit is initialized. For more details about homodyne and heterodyne detection in the context of circuit QED see section 4.3 in [15] and section 6.3 in [29].

The acquired data is returned to the MC where the method create_experimentaldata_dataset prepares it for the MeasurementAnalysis object and saves it using the h5py package. h5py is a Python-style interface to the HDF5 format [30]. The latter is a platform-independent package developed by the HDF Group to facilitate storage and handling of big data [31]. MeasurementControl saves the measurement data using the Data object defined in hdf5_data.py, which is a wrapper of an h5py data object.

In the remainder of this section, the *hard* sweep functions, and in particular the classes that create the AWG sequences, will be discussed in more detail.

AWG Sequences Classes as Sweep Functions

All time-domain measurements presented in this thesis (Rabi, Ramsey, T1, DRAG pulse calibration) use the AWG *hard* sweep functions classes, which are the ones that upload the pulse sequences presented in Section 2.2 to the AWG. These routines follow the template illustrated in Listing 3.5 for the Rabi class.

The prepare method in Listing 3.5 (line 17) passes all the relevant parameters to the Rabi_seq function (line 19) shown in Listing 3.6. The Rabi_seq function then proceeds to define (lines 23-28) and upload (line 36) the *pulse_list* to the AWG via the Pulsar object, a meta-instrument that controls the communication with the AWGs. Pulsar needs to know the Sequence object and the list of Elements to upload. An Element contains a list of Pulse objects, which create a waveform of time-amplitude pairs for each AWG channel. An example of a Pulse is the SSB_DRAG_pulse defined in pulse_library.py. As the name suggests, this object implements the DRAG pulse described in Section 2.2.6.

Once it has the list of pulses, Pulsar also needs to know the instructions for how to build an AWG sequence. This information is contained in the Sequence object, which is a list of instructions specifying the Elements that must be used, the order in which they must occur, and the number of times that each should be repeated. The Sequence object also contains information that instructs the AWG whether to expect a trigger and at what point it should expect it.

Finally, the QuCoDeS object Station contains all the information about an existing experimental setup, thus defining the current work station. Hence, line 36 in Listing 3.6 instructs PycQED to upload the newly-generated Rabi sequence to the AWGs in the current work station. All the instruments and meta-instruments (including Pulsar) must be added to the Station object during each initialization of PycQED.

```python
class Rabi(swf.Hard_Sweep):

    def __init__(self, pulse_pars, RO_pars, n=1, cal_points=True,
                 no_cal_points=2, upload=True, return_seq=False):
        super().__init__()
        self.pulse_pars = pulse_pars
        self.RO_pars = RO_pars
        self.n = n
        self.cal_points = cal_points
        self.no_cal_points=no_cal_points
        self.upload = upload
        self.name = 'Rabi'
        self.parameter_name = 'amplitude'
        self.unit = 'V'
        self.return_seq = return_seq

    def prepare(self, **kw):
        if self.upload:
            sqs.Rabi_seq(amps=self.sweep_points,
                         pulse_pars=self.pulse_pars,
                         RO_pars=self.RO_pars,
                         cal_points=self.cal_points,
                         no_cal_points=self.no_cal_points,
                         n=self.n, return_seq=self.return_seq)
```

Listing 3.5: Source code for the Rabi AWG sweep function class inside awg_sweep_functions.py. In the input parameters, *pulse_pars* and *RO_pars* are the qubit drive parameters and the readout parameters described in Section 3.2.1, *n* is a number specifying how many times to repeat the same sequence before measurement, *cal_points* are the measurement calibration points, *no_cal_points* specifies the number of calibration points to use, *upload* specifies whether or not to upload the sequence to the AWG, and *return_seq* instructs the Rabi_seq function to return the generated sequence object. The role of calibration points will be explained in Section 3.2.3. sqs refers to the single_qubit_tek_seq_elts.py module which contains all the objects that generate AWG sequences for single qubits. "kw" means "keyword arguments" and it allows functions to take an arbitrary number of input keyword arguments, such as all the input parameters to Rabi_seq.

```python
def Rabi_seq(amps, pulse_pars, RO_pars, n=1, post_msmt_delay=3e-6, no_cal_points=2,
             cal_points=True, verbose=False, upload=True, return_seq=False):

    seq_name = 'Rabi_sequence'
    seq = sequence.Sequence(seq_name)
    station.pulsar.update_channel_settings()
    el_list = []
    pulses = get_pulse_dict_from_pars(pulse_pars)

    for i, amp in enumerate(amps):  # seq has to have at least 2 elts
        # 4 calibration points
        if cal_points and no_cal_points==4 and \
                (i == (len(amps)-4) or i == (len(amps)-3)):
            el = multi_pulse_elt(i, station,[pulses['I'], RO_pars])
        elif cal_points and no_cal_points==4 and \
16                  (i == (len(amps)-2) or i == (len(amps)-1)):
17              el = multi_pulse_elt(i, station, [pulses['X180'], RO_pars])
18          # 2 calibration points
19          elif cal_points and no_cal_points==2 and \
20                  (i == (len(amps)-2) or i == (len(amps)-1)):
21              el = multi_pulse_elt(i, station,[pulses['I'], RO_pars])
22          else:
23              pulses['X180']['amplitude'] = amp
24              pulse_list = n*[pulses['X180']]+[RO_pars]
25
26              # copy first element and set extra wait
27              pulse_list[0] = deepcopy(pulse_list[0])
28              pulse_list[0]['pulse_delay'] += post_msmt_delay
29
30              el = multi_pulse_elt(i, station, pulse_list)
31
32          el_list.append(el)
33          seq.append_element(el, trigger_wait=True)
34
35      if upload:
36          station.pulsar.program_awgs(seq, *el_list, verbose=verbose)
37
38      if return_seq:
39          return seq, el_list
40      else:
41          return seq
```

Listing 3.6: Source code for the rabi_seq function inside single_qubit_tek_seq_elts.py. In the input parameters, *amps* is the array of Rabi amplitudes, *pulse_pars* and *RO_pars* are the qubit drive parameters and the readout parameters described in Section 3.2.1, *cal_points* are the measurement calibration points (see main text), *no_cal_points* specifies the number of calibration points to use, *upload* specifies whether or not to upload the sequence to the AWG, *return_seq* decides whether to return the generated sequence object, *n* is a number specifying how many times to repeat the same sequence before measurement, and *post_msmt_delay* is the time to wait between the measurement of one run and the start of the next run. *verbose* is a Python parameter that instructs a function to output detailed logging information during its operation. The role of calibration points in lines 12-21 will be explained in Section 3.2.3.

3.2.3 The Measurement Analysis Structure

This section will present the overall structure of the measurement_analysis.py module, with a focus on the analysis routines needed for single-qubit gates calibration. Most of this structure already existed at the start of our collaboration with the DiCarlo group, but several features have been improved throughout this project. This section will describe the structural changes that have been made to measurement_analysis.py, and Section 4 will present in detail the original and current states of the analysis classes for single-quit operations.

The MeasurementAnalysis object was designed to be the lowest base class for all the analysis routines discussed in this thesis. Thus, it implements functions that are common to all these classes. The get_naming_and_values(_2D) method[2] extracts all the measurement parameters from the data file. In particular, it extracts the array of sweep points (*sweep_points*) and the array of acquired data (*measured_values*), and it gets the names and units of the measured variables. This method was improved during this project to also produce the *scaled_sweep_points* array with the sweep points scaled to the appropriate units for each measurement.

The default_ax(is) and default_fig(ure) methods create plots with the standard sizes for one and two column articles defined in the Physical Review Letters (PRL) journal [32] (this feature was added during this project). The plot_results_vs_sweepparam method is the main plotting

[2]The "2D" refers to a two-dimensional measurement where two independent variables are swept and the response of the dependent variable is measured.

function used by all analysis routines to plot the final results versus the sweep parameters (for example the $|e\rangle$ population vs. time delay values for Ramsey). This method was already implemented before this project, and it was incorporated into some of the analysis routines for single-qubit calibration measurements. During this project, this plotting function was also included into the remaining routines and has been improved to produce well-proportioned graphs (all figures in Section 4 illustrate plots produced with this method).

The MeasurementAnalysis base class also save(s)_fitted_parameters that result from the fit, save(s)_computed_parameters that are computed from the fit results (π-pulse , $\pi/2$-pulse etc.), and save(s)_fig(ure).

In each analysis class, the run_default_analysis method is the main function that controls the entire flow of operations required in each routine. In MeasurementAnalysis this method simply produces two subplots of the *measured_values* vs. *sweep_points*. Depending on the measurement type, the *measured_values* array can store either the raw I and Q data acquired by the UHFLI (for example for all time-domain measurements in Sections 2.2.3-2.2.6), or the complex magnitude and angle of the I and Q data (for example for spectroscopy measurements). The plots produced by the run_default_analysis method of the base class for a resonator spectroscopy measurement is shown in Figure 3.1.

In order to generate plots of the unmodified data like the one shown in Figure 3.1, all the other analysis classes inherit this method either directly from the MeasurementAnalysis base class, or indirectly from another intermediate base class, such as TD_Analysis discussed below. Each analysis class then proceeds to add operations and plots that are specific to each routine (see Section 4).

The analysis classes for qubit spectroscopy and resonator spectroscopy inherit directly from MeasurementAnalysis. All the other single-qubit calibration analysis routines deal with time-domain experiments, and were designed to inherit first from TD_Analysis, which itself inherits from MeasurementAnalysis. The TD_Analysis class adds on top of the MeasurementAnalysis base class those functions that are common to all time-domain analysis routines. In particular, the normalize_data_to_calibration_points method in TD_Analysis uses calibration points (*cal_points*) to extract the most useful information from a data set. This calibration process is explained in detail later in this section. At the start of our collaboration, this method could only support 4 calibration points. As a result of this project, the algorithm is now more robust and allows the user to choose between using 2, 4, 6, or no calibration points for a measurement.

As mentioned above, the TD_Analysis class inherits the run_default_analysis method from MeasurementAnalysis. An important improvement added during this project is that now TD_Analysis also generates a plot object of the calibrated data versus the *sweep_points*, which is accessible to each analysis class for single-qubit time-domain operations. These specific classes fit the calibrated data to the relevant fitting model, extract the parameters of interest, and plot the fit results on the same figure that was produced in TD_Analysis. As a result, no code is duplicated, and the plots generated by all time-domain routines shown in Section 4 are consistent.

This layered inheritance structure was the original intent for measurement_analysis.py. Yet, it was not being properly used before the completion of this project. For example, the code for producing the plots shown in Figure 3.1 was duplicated several times throughout the entire inheritance tree instead of reusing the same algorithm from the run_default_analysis method of MeasurementAnalysis. This type of implementation not only introduces redundancy and decreases code readability, but it also produces a lack of consistency between plots produced with different routines. Consequently, an additional improvement as a result of this project is that all the figures produced by the analysis classes for single-qubit operations are consistent in that they use the same font sizes, marker sizes, and line widths.

The remainder of this section will describe how calibration points are used to extract the most information about the qubit state from the raw measurement data.

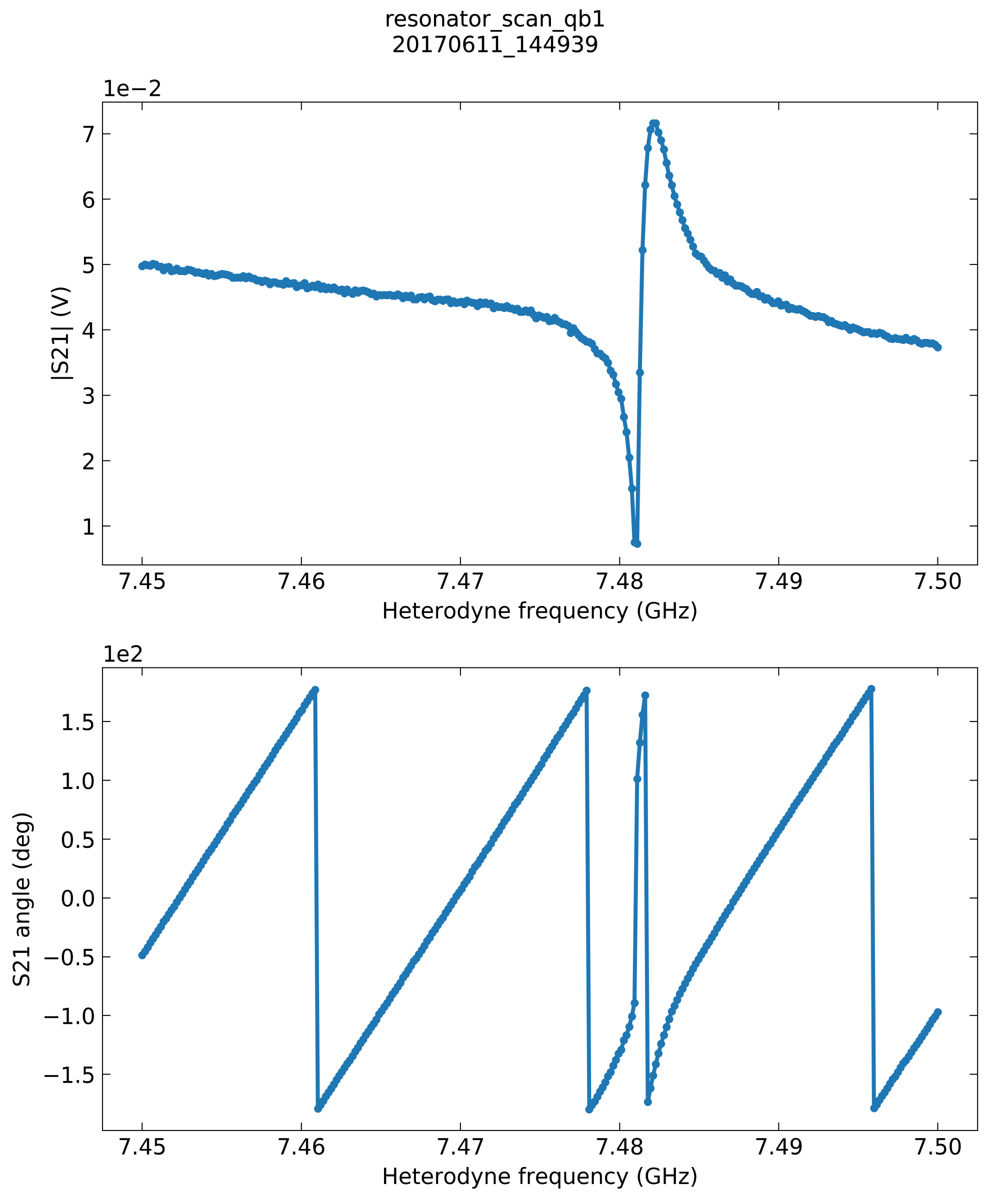


Figure 3.1: Complex amplitude ($|S21|$) and phase (*S21 angle*) of the I and Q raw data from a resonator spectroscopy measurement plotted as a function of the input frequency to the readout resonator (*heterodyne frequency*). In the current version of PycQED, the x-axis label will always be *Heterodyne frequency* irrespective of whether a heterodyne or homodyne acquisition setup was actually used for the measurement (see Section 3.2.2 for details).

Measurement Calibration for Time-Domain Analysis

Calibration points are used to find the best way to combine the I and Q data stored in *measured_values* into one single data set with as little loss of information about the qubit state as possible. For a Rabi measurement, for example, the information about the qubit population must be extracted from the raw I and Q data. If we look back at the rabi_seq in Listing 3.6, the *no_cal_points* parameter decides for how many values in the array of pulse amplitudes to use calibration sequences instead of the normal Rabi sequence described in Section 2.2.3. If there are 2 *cal_points*, then the last two sequences in the Rabi measurement consist of an '*I*' pulse each. '*I*' is the PycQED notation for an identity pulse, i.e. nothing is applied and the qubit remains in the $|g\rangle$ state. The reason why two identity pulses are used instead of one is explained below. If 4 *cal_points* are passed in, then the last four sequences are used as calibration sequences such that the first two of these last four sequences are I-pulses (qubit remains in $|g\rangle$), and the very last two sequences are *X180*-pulses. '*X180*' is the PycQED notation for a π-pulse on the $|g\rangle \leftrightarrow |e\rangle$ transition, which brings the qubit population into the $|e\rangle$ state. There is also the third option where the *cal_points* parameter is False. In this case a regular Rabi sequence is applied for all the sweep points, and no *cal_points* are used.

The reason for using two identical calibration sequences for each calibration state is to improve the accuracy of each calibration measurement. When the data is modified based on these points, the results from each pair of identical calibration measurements are averaged. Then, just as one would expect, there are indeed only *no_cal_points*/2 calibration states.

The matter becomes slightly more complicated for qutrit measurements, as here we have three transmon states and can have up to 6 *cal_points*. The source code for rabi_2nd_exc_seq is shown in Listing 3.7. As we have seen in the previous paragraph, the *cal_points* place the qubit population in one of the two qubit states for the duration of the measurement. For a qutrit then, we expect the *cal_points* to do the same. It is easy to see that for 6 *cal_points*, the last six sequences do precisely this: the first two of these last six leave the qubit in $|g\rangle$, the middle two leave it in $|e\rangle$, and the last two leave it in $|f\rangle$. For 4, 2, and no *cal_points*, the behavior is the same as described in the previous paragraph.

```python
def Rabi_2nd_exc_seq(amps, pulse_pars, pulse_pars_2nd, RO_pars, n=1,
                     cal_points=True, no_cal_points=4, upload=True, return_seq=False,
                     post_msmt_delay=3e-6, verbose=False, last_ge_pulse=True):

    seq_name = 'Rabi_2nd_exc_sequence'
    seq = sequence.Sequence(seq_name)
    station.pulsar.update_channel_settings()
    el_list = []
    pulses = get_pulse_dict_from_pars(pulse_pars)
    pulses_2nd = get_pulse_dict_from_pars(pulse_pars_2nd)

    for i, amp in enumerate(amps):  # seq has to have at least 2 elts
        # 6 calibration points
        if cal_points and no_cal_points == 6 and \
                (i == (len(amps)-6) or i == (len(amps)-5)):
            el = multi_pulse_elt(i, station, [pulses['I'], pulses_2nd['I'], RO_pars])
        elif cal_points and no_cal_points == 6 and \
                (i == (len(amps)-4) or i == (len(amps)-3)):
            el = multi_pulse_elt(i, station, [pulses['X180'], pulses_2nd['I'], RO_pars])
        elif cal_points and no_cal_points == 6 and \
                (i == (len(amps)-2) or i == (len(amps)-1)):
            el = multi_pulse_elt(i, station, [pulses['X180'], pulses_2nd['X180'], RO_pars])
        # 4 calibration points
        elif cal_points and no_cal_points == 4 and \
                (i == (len(amps)-4) or i == (len(amps)-3)):
            el = multi_pulse_elt(i, station, [pulses['I'], pulses_2nd['I'], RO_pars])
        elif cal_points and no_cal_points == 4 and \
                (i == (len(amps)-2) or i == (len(amps)-1)):
            el = multi_pulse_elt(i, station, [pulses['X180'], pulses_2nd['I'], RO_pars])
        # 2 calibration points
        elif cal_points and no_cal_points == 2 and \
                (i == (len(amps)-2) or i == (len(amps)-1)):
            el = multi_pulse_elt(i, station, [pulses['I'], pulses_2nd['I'], RO_pars])
        else:
            pulses_2nd['X180']['amplitude'] = amp

            pulse_list = [pulses['X180']]+n*[pulses_2nd['X180']]

            if last_ge_pulse:
                pulse_list += [pulses['X180']]

            pulse_list += [RO_pars]

            # copy first element and set extra wait
            pulse_list[0] = deepcopy(pulse_list[0])
            pulse_list[0]['pulse_delay'] += post_msmt_delay
            el = multi_pulse_elt(i, station, pulse_list)
        el_list.append(el)
        seq.append_element(el, trigger_wait=True)

    if upload:
        station.pulsar.program_awgs(seq, *el_list, verbose=verbose)

    if return_seq:
        return seq_name, el_list
    else:
        return seq
```

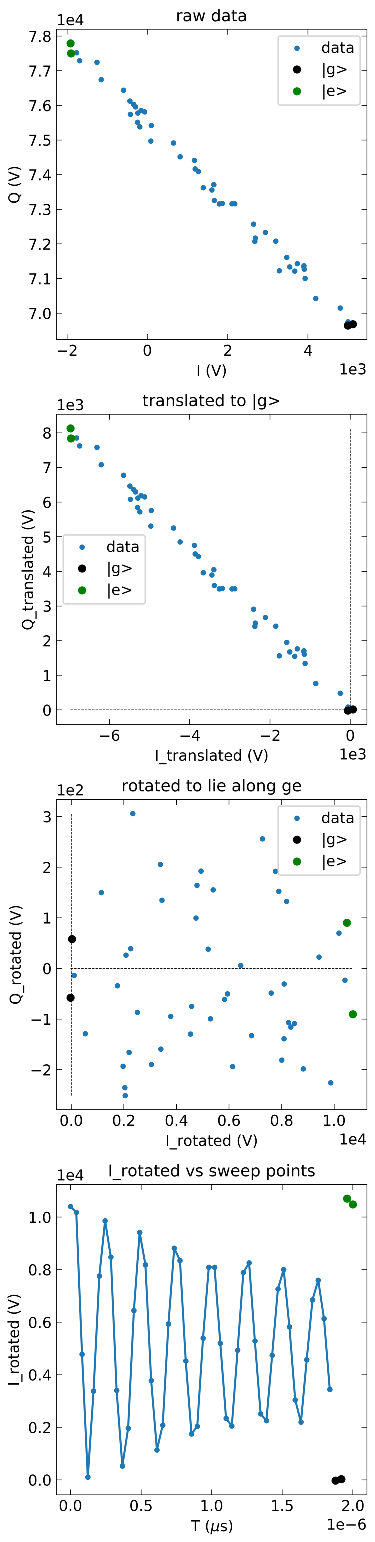
Listing 3.7: Source code for the rabi_2nd_exc_seq function inside single_qubit_2nd_exc_sqs.py. This routine implements the same functionality as the Rabi_seq function discussed in Section 3.2.2, but for a qutrit. In the input parameters, *amps* is the array of Rabi amplitudes, *pulse_pars* and *RO_pars* are the qubit drive parameters and the readout parameters described in Section 3.2.1, *pulse_pars_2nd* are the drive parameters for the $|e\rangle \leftrightarrow |f\rangle$ transition, n is a number specifying how many times to repeat the same sequence before measurement, *cal_points* are the measurement calibration points, *no_cal_points* specifies the number of calibration points to use, *upload* specifies whether or not to upload the sequence to the AWG, *return_seq* decides whether to return the generated sequence object, *post_msmt_delay* is the time to wait between the measurement of one run and the start of the next run, and *last_ge_pulse* decides whether there will be a $|g\rangle \leftrightarrow |e\rangle$ π-pulse at the end of each sequence (see Section 2.2.3). *verbose* is a Python parameter that instructs a function to output detailed logging information during its operation.

To help understand what the *cal_points* do, the raw data from two Ramsey experiments plotted in the complex plane are shown in the topmost graphs of Figure 3.2, where the *cal_points* are marked in black, green, and red, for the $|g\rangle$, $|e\rangle$, and $|f\rangle$ states, respectively. The question can now be reformulated to ask, on which axis in this plane should the data points be projected in order to obtain the most relevant information about the qubit state compared to just simply discarding the Q data points[3] (the y axis in the top row of Figure 3.2).

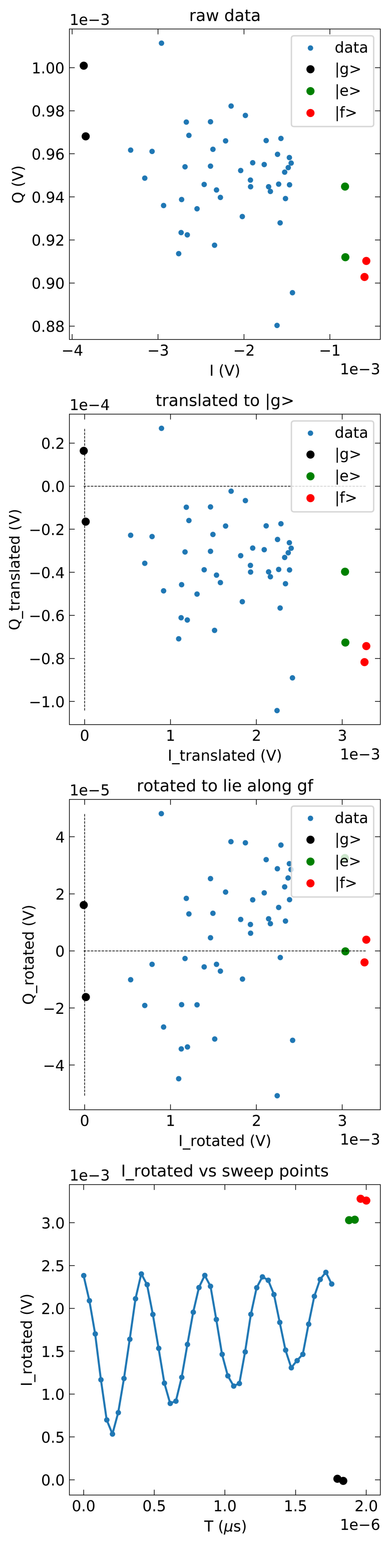
If no calibration points are used, then we have no certain information that any one of the data points truly represents a known qubit state. Thus, the best strategy is to use principal component analysis (PCA) in order to project the data onto the line that minimizes the normal distance from each data point to the line. This is different from least squares minimization where the distance along the x or y axis from each data point to the line is minimized. For details on PCA see [33].

For all other cases, the data is first translated such that the qubit $|g\rangle$ state is placed at the origin of the coordinate system (see second row of graphs in Figure 3.2). This effectively means that everything is done with respect to the $|g\rangle$ state. If only 2 *cal_points* exist, then we know

[3]This technique is sometimes used in practice because most of the information about the qubit is contained in the I data. Some of the analysis routines have support for cases where only one quadrature is measured, but not all. This issue was outside the scope of this project.



(a) Ramsey_ge
4 *cal_points*



(b) Ramsey_ef
6 *cal_points*
last_ge_pulse=True

Figure 3.2: The steps for measurement calibration based on calibration points are illustrated for a Ramsey measurement on a qubit (a) and qutrit (b). I and Q refer to the in-phase an quadrature data arrays stored in *measured_values*. See main text for details.

that they represent the $|g\rangle$ qubit state. A least squares optimization routine is employed to find the line of best fit through the data that also necessarily intercepts the origin ($|g\rangle$ state). The data is then projected onto it.

If there are 4 *cal_points*, the data points are then rotated such that they lie along the axis intersecting the two qubit states described by those calibration points (third row in Figure 3.2). For example, if the analysis is for a qubit, then the 4 *cal_points* are the $|g\rangle$ and $|e\rangle$ states (Figure 3.2a). If there are 6 *cal_points*, and a $|g\rangle \leftrightarrow |e\rangle$ π-pulse was applied before each measurement (*last_ge_pulse*=True in Listing 3.7), then only the *cal_points* that place the qutrit in the $|g\rangle$ and $|f\rangle$ states will be used to rotate the data, because the oscillations will be between those two states (Figure 3.2b). Similarly, for 6 *cal_points* and *last_ge_pulse*=False, only the $|e\rangle$ and $|f\rangle$ states calibration points are used. After the correct rotation is performed, the data is projected onto the respective axis by keeping only the first array (the x coordinates) in the rotated data and discarding the other. The last row in Figure 3.2 plots this array versus the *sweep_points*.

The function that performs the measurement calibration for all the cases discussed above is called rotate_and_normalize_data, and is defined in analysis_toolbox.py.

3.3 Using PycQED in an Experiment

This section will show how to use PycQED to perform the single-qubit calibration measurements discussed in Section 2.2.

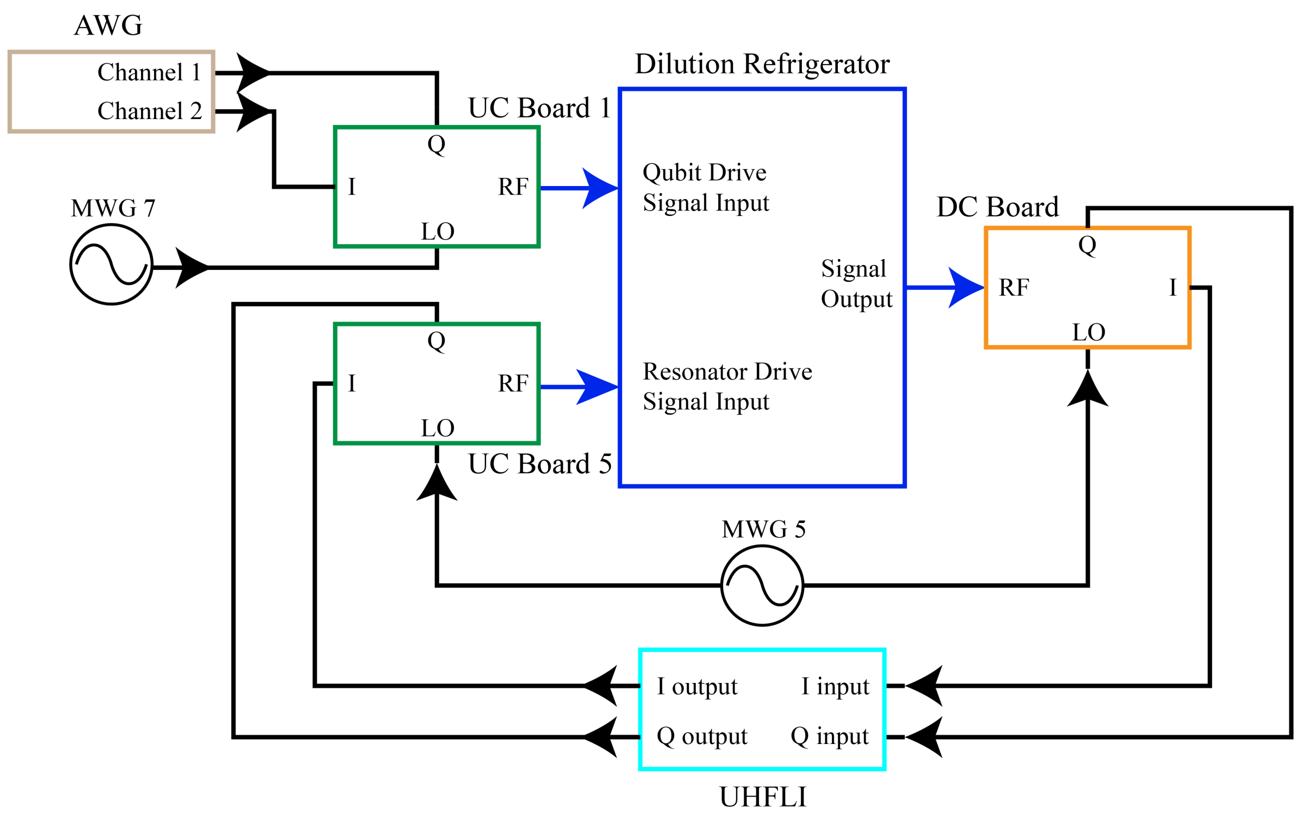
All the relevant folders on the PycQED PC are located on the Data drive. The data folder is where each measurement is saved, and the Control software folder contains the local copies of the PycQED and QuCoDeS repositories, as well as the Jupyter Notebooks used to actually perform the measurements (this interface is illustrated in the three figures shown below). For logging purposes, the current convention is that a different notebook should be used on each new day of measurements.

The first cell of each notebook must call the *init*(ialization) script for the particular chip and setup used. Most[4] of the work in this thesis was performed on the eight qubit chip presented in Section 2.1.1, which uses the initialization script bluefors1_M81B1.py, where M81B1 is the chip name. The role of this script is to import all the required files from the PycQED and QuCoDeS repositories, to open the connections to all the instruments and initialize all meta-instruments (including all the 8 qubits available on the chip), and to initialize a QuCoDeS station and add all the meta-instruments to it.

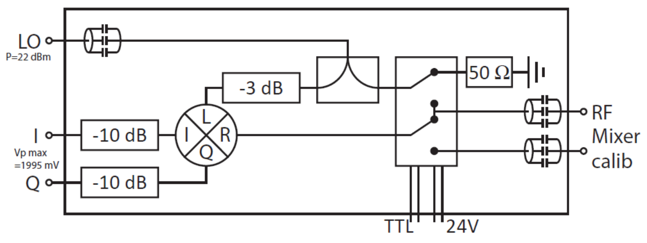
The room-temperature electronics setup used for this example is shown in Figure 3.3a. Upconversion board 5 (UC5) was used for readout, with the LO provided by microwave generator (MWG) 5, and UC2 was used for qubit drive, with the LO provided by MWG 7. The intermediate frequency (IF) for both UC5 and the downconversion board (DC) was generated by the UHFLI, and for UC2 it was generated by channels 1 and 2 of AWG1. Each UC board has a switch that decides whether the LO signal is routed directly to the RF output (*bypass* mode), or whether it first gets upconverted by the mixer (*modulated* mode). The circuit diagram of the upconversion boards used in our lab is illustrated in Figure 3.3b. For the measurements shown in the three figures below, the *bypass* mode is only used for the continuous wave qubit spectroscopy measurement.

Figure 3.4 shows how to import the *init* file, how to set the control switches inside the upconversion boards and the warm amplifier (*WA*), how to set the voltage to the coil that tunes the frequencies of the desired qubit, and how to perform resonator spectroscopy by calling the find_resonator_frequency method of QuDev_transmon. The *homodyne* object is the

[4]See Appendix B for a summary of the different samples used throughout this thesis.



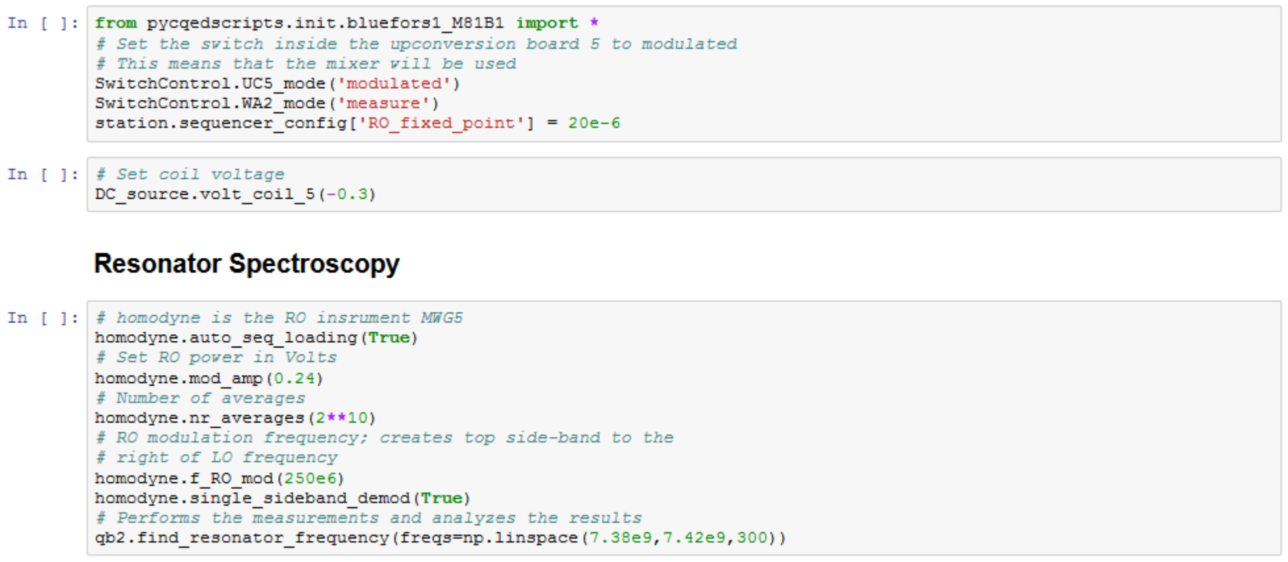
(a) Room temperature setup.



(b) Upconversion board circuit diagram.

Figure 3.3: (a) Room temperature setup for the microwave components and devices used in the example presented in this section for how to perform measurements in PycQED. See main text for details. (b) Circuit diagram for the upconversion (UC) boards used in our lab. LO is the local oscillator, I and Q are the in-phase and quadrature components of the intermediate frequency (IF) signal, and ”Mixer calib” is the output used for mixer calibration (more details about mixer calibrations can be found in [15]). The radio-frequency (RF) port of the UC board mixer is used as an output that is routed to the dilution refrigerator, while the RF port of the downconversion (DC) board is used as an input from the dilution refrigerator.

meta-instrument that controls all the readout equipment: the microwave generator providing the LO signal, the instrument providing the IF signal, the acquisition instrument (in this case the UHFLI), and the trigger instrument (typically the AWG). In this case, the class instance name homodyne does refer to a homodyne setup, where the mixers inside the readout upconversion and resonator signal downconversion boards use the same LO signal, which for this setup was MWG 5. The readout power (in Volts) and the IF frequency are set with the *mod_amp* and *f_RO_mod* parameters, respectively. Finally, the *RO_fixed_point* sets the time at which the AWG should generate the readout pulse trigger.



```
In [ ]: from pycqedscripts.init.bluefors1_M81B1 import *
        # Set the switch inside the upconversion board 5 to modulated
        # This means that the mixer will be used
        SwitchControl.UC5_mode('modulated')
        SwitchControl.WA2_mode('measure')
        station.sequencer_config['RO_fixed_point'] = 20e-6
```

```
In [ ]: # Set coil voltage
        DC_source.volt_coil_5(-0.3)
```

Resonator Spectroscopy

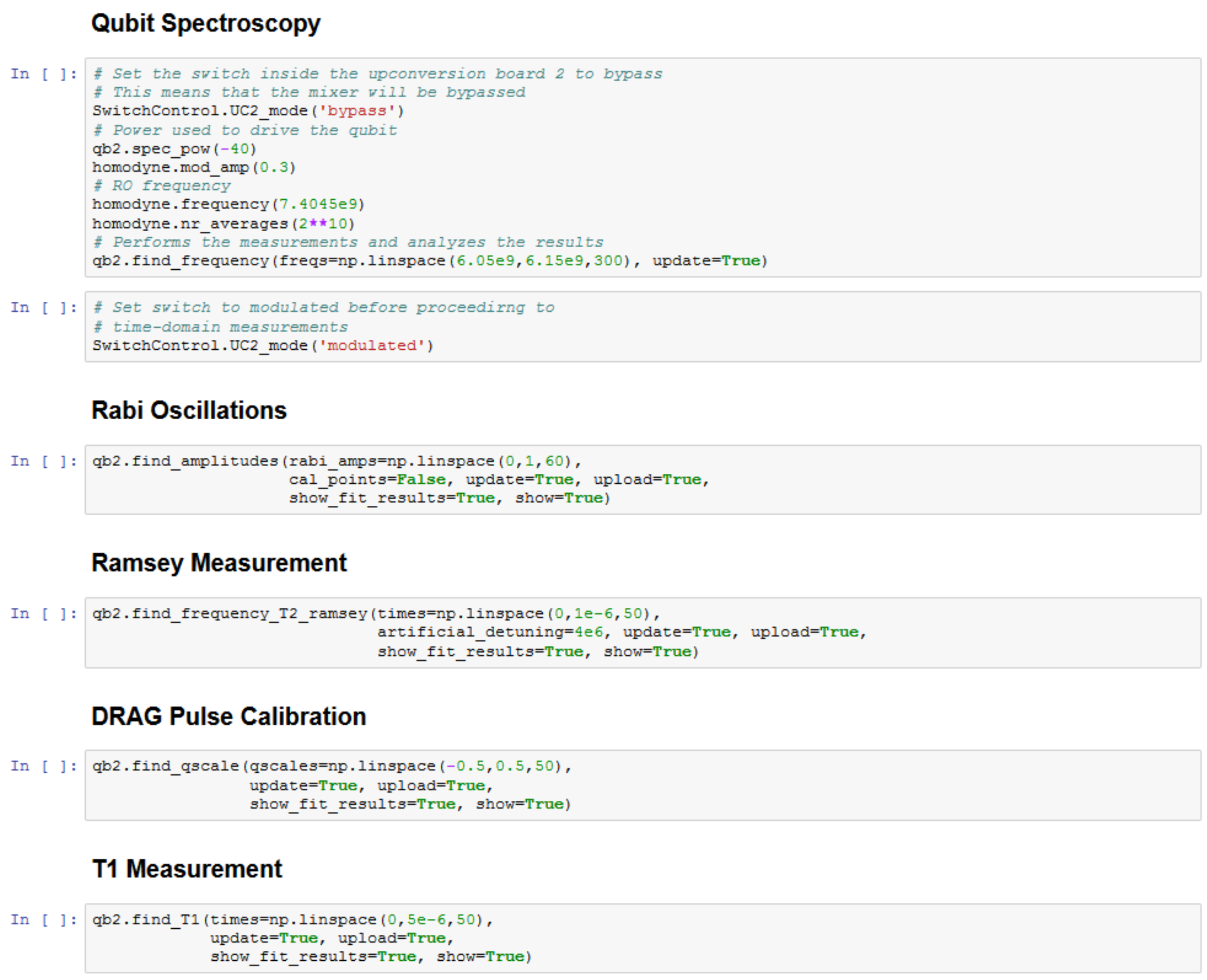
```
In [ ]: # homodyne is the RO insrument MWG5
        homodyne.auto_seq_loading(True)
        # Set RO power in Volts
        homodyne.mod_amp(0.24)
        # Number of averages
        homodyne.nr_averages(2**10)
        # RO modulation frequency; creates top side-band to the
        # right of LO frequency
        homodyne.f_RO_mod(250e6)
        homodyne.single_sideband_demod(True)
        # Performs the measurements and analyzes the results
        qb2.find_resonator_frequency(freqs=np.linspace(7.38e9,7.42e9,300))
```

Figure 3.4: General settings and resonator spectroscopy. All input values must be in SI units. *WA* and *UC* refer to the warm amplifier and the upconversion boards, respectively. The *RO_fixed_point* sets the time at which the readout pulse occurs, DC_source is the instrument that sets the DC coil voltages, and homodyne is the readout meta-instrument.

Figure 3.5 and Figure 3.6 show how to perform the single-qubit gates calibration measurements on a qubit and qutrit, respectively. Both spectroscopy measurements are continuous wave (see Section 2.2.2 for details). Switching between a qubit and a qutrit operation is easy in PycQED as it only requires setting the *for_ef* (or *analyze_ef* for qubit spectroscopy[5]) parameter to True. In these two figures, *update* decides whether to update the relevant qubit parameters, *upload* decides whether to upload the respective sequences to the AWG (since this takes some time, it is useful to be able to prevent it if the same measurement is performed multiple times in a row), *show_fit_results* displays the fit report, *show* displays all images produced by the analysis routines, and *last_ge_pulse* decides whether to finish each sequence in a qutrit measurement with a π-pulse on the $|g\rangle \leftrightarrow |e\rangle$ transition.

The readout frequency can be set either by configuring the frequency of the homodyne meta-instrument directly (as is done for both resonator and qubit spectroscopy), or by changing the *f_RO* qubit parameter. This option allows the user to both store a different value of this parameter for each qubit, and control it directly through the readout meta-instrument for measurements that might not involve a qubit. The same applies for all other readout parameters.

[5]This parameter has a different name for qubit spectroscopy because here the relevant changes in the measurement settings (higher qubit drive power and wider sweep range) are made by the user not by PycQED behind the scenes. The only information PycQED does need to know is whether to search for two peaks instead of one in the analysis routine.



Qubit Spectroscopy

```
In [ ]: # Set the switch inside the upconversion board 2 to bypass
        # This means that the mixer will be bypassed
        SwitchControl.UC2_mode('bypass')
        # Power used to drive the qubit
        qb2.spec_pow(-40)
        homodyne.mod_amp(0.3)
        # RO frequency
        homodyne.frequency(7.4045e9)
        homodyne.nr_averages(2**10)
        # Performs the measurements and analyzes the results
        qb2.find_frequency(freqs=np.linspace(6.05e9,6.15e9,300), update=True)
```

```
In [ ]: # Set switch to modulated before proceedirng to
        # time-domain measurements
        SwitchControl.UC2_mode('modulated')
```

Rabi Oscillations

```
In [ ]: qb2.find_amplitudes(rabi_amps=np.linspace(0,1,60),
                            cal_points=False, update=True, upload=True,
                            show_fit_results=True, show=True)
```

Ramsey Measurement

```
In [ ]: qb2.find_frequency_T2_ramsey(times=np.linspace(0,1e-6,50),
                                     artificial_detuning=4e6, update=True, upload=True,
                                     show_fit_results=True, show=True)
```

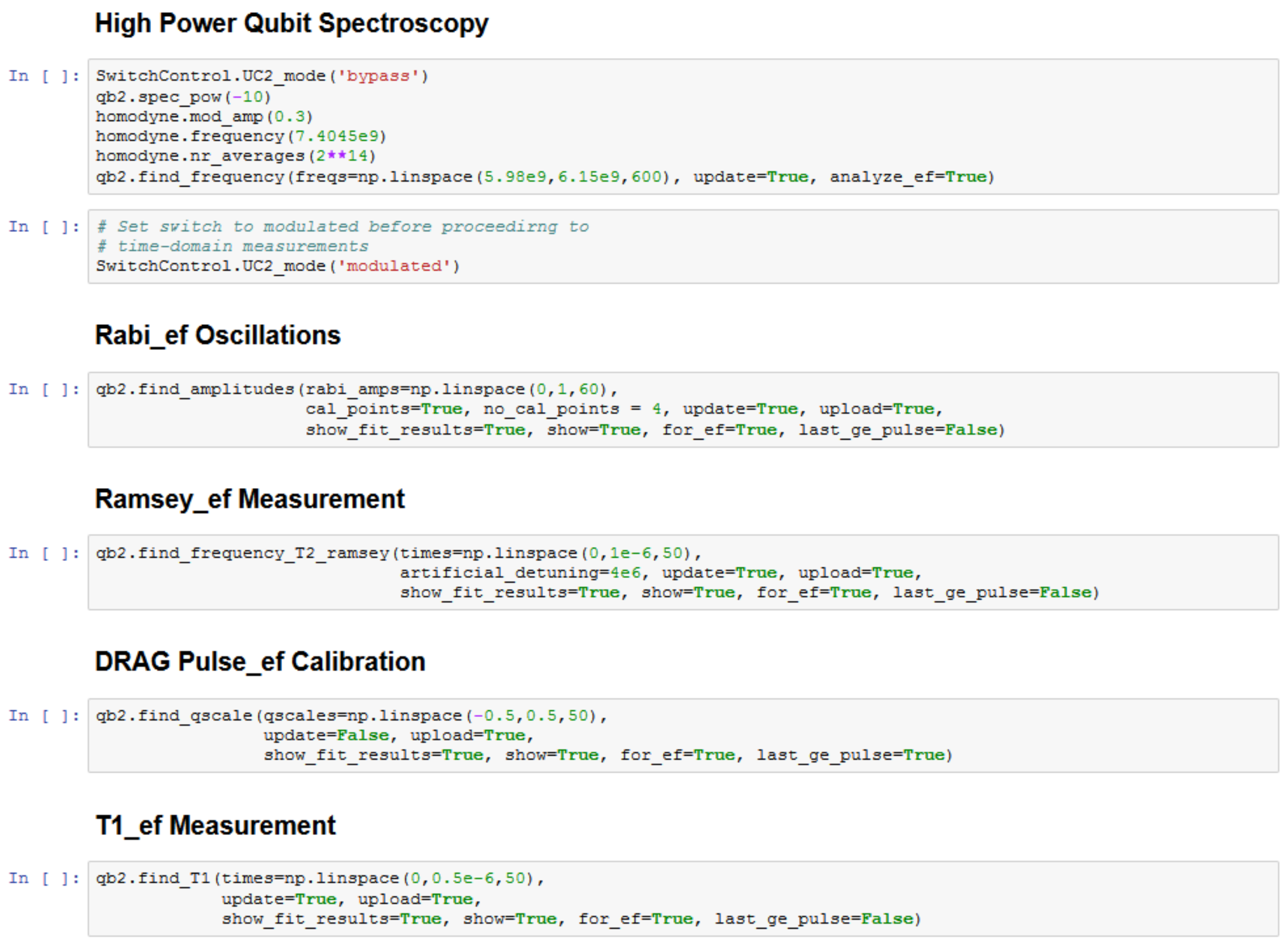
DRAG Pulse Calibration

```
In [ ]: qb2.find_qscale(qscales=np.linspace(-0.5,0.5,50),
                        update=True, upload=True,
                        show_fit_results=True, show=True)
```

T1 Measurement

```
In [ ]: qb2.find_T1(times=np.linspace(0,5e-6,50),
                    update=True, upload=True,
                    show_fit_results=True, show=True)
```

Figure 3.5: Qubit calibration measurements. All input values must be in SI units. *UC2* refers to the upconversion board used for qubit drive, and homodyne is the readout meta-instrument.



High Power Qubit Spectroscopy

```
In [ ]: SwitchControl.UC2_mode('bypass')
        qb2.spec_pow(-10)
        homodyne.mod_amp(0.3)
        homodyne.frequency(7.4045e9)
        homodyne.nr_averages(2**14)
        qb2.find_frequency(freqs=np.linspace(5.98e9,6.15e9,600), update=True, analyze_ef=True)
```

```
In [ ]: # Set switch to modulated before proceedirng to
        # time-domain measurements
        SwitchControl.UC2_mode('modulated')
```

Rabi_ef Oscillations

```
In [ ]: qb2.find_amplitudes(rabi_amps=np.linspace(0,1,60),
                            cal_points=True, no_cal_points = 4, update=True, upload=True,
                            show_fit_results=True, show=True, for_ef=True, last_ge_pulse=False)
```

Ramsey_ef Measurement

```
In [ ]: qb2.find_frequency_T2_ramsey(times=np.linspace(0,1e-6,50),
                                     artificial_detuning=4e6, update=True, upload=True,
                                     show_fit_results=True, show=True, for_ef=True, last_ge_pulse=False)
```

DRAG Pulse_ef Calibration

```
In [ ]: qb2.find_qscale(qscales=np.linspace(-0.5,0.5,50),
                        update=False, upload=True,
                        show_fit_results=True, show=True, for_ef=True, last_ge_pulse=True)
```

T1_ef Measurement

```
In [ ]: qb2.find_T1(times=np.linspace(0,0.5e-6,50),
                    update=True, upload=True,
                    show_fit_results=True, show=True, for_ef=True, last_ge_pulse=False)
```

Figure 3.6: Qutrit calibration measurements. All input values must be in SI units. *UC2* refers to the upconversion board used for qubit drive, and homodyne is the readout meta-instrument.

4. Single-Qubit Calibration using PycQED

In Section 3.2.3 the overall structure of measurement_analysis.py was presented and only MeasurementAnalysis and TD_Analysis, the base classes of all single-qubit calibration analysis routines, were described in detail. This section will present the data analysis classes for each of the calibration measurements described in Section 2.2. All these classes existed at the start of our collaboration with the DiCarlo group. Each subsection below will discuss the status of the respective class at the start of this project, whether improvements have been made and what they are, and will illustrate the current status of these routines with experimental results.

4.1 Resonator Spectroscopy

The resonator spectroscopy analysis class is called Homodyne_Analysis. Adding major improvements to this routine, such as more complex fitting models, was not the focus of this project. However, the generated plots have been improved as described in Section 3.2.3, and the existing models have been tested. This class can fit the complex amplitude of the signal to a "sloped Hanger" function (describing the structure shown in Figure 4.1a between 7.49 GHz and 7.5 GHz), or the square of that amplitude (the power) to a Lorentzian function (shown in Figure 4.1b):

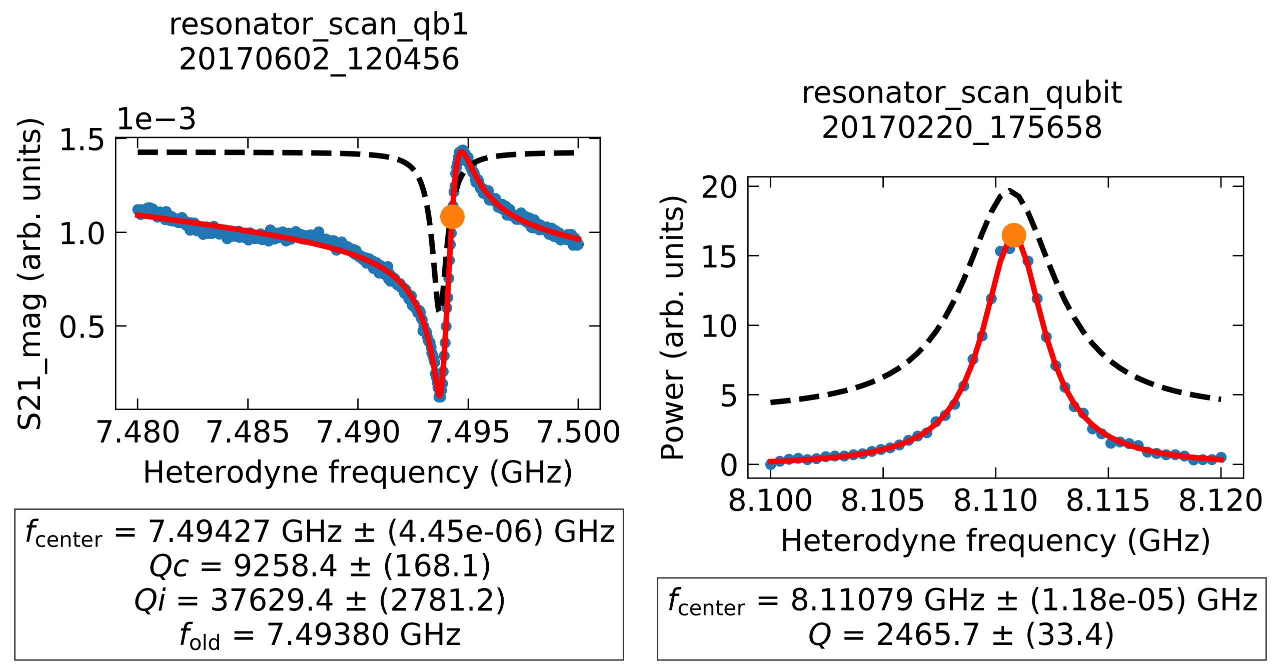
$$\left|\mathrm{A}\frac{1-\frac{Q}{Q_e}e^{i\theta}}{1+2i\frac{f-f_0}{f_0}}\left(1+df\frac{f-f_0}{f_0}\right)\right| \tag{sloped Hanger}$$

$$\frac{\mathrm{A}}{\pi}\frac{\kappa}{(f-f_0)^2+\kappa^2}+\text{offset}. \tag{Lorentzian}$$

In the equations above, df is the slope of the deviation of the hanger structure away from a potentially non-flat background, f_0 is the resonance frequency of the resonator, f is the independent variable which is swept in the measurement, A is the amplitude, Q and Q_e are the total and external quality factors of the resonator (see [15]), θ is a phase angle describing the capacitance couplings to the resonator, κ is the full width at half maximum (FWHM) linewidth, and offset is the offset from zero.

A few issues are already apparent in Figure 4.1. First, Figure 4.1a already indicates in the interval 7.48 GHz - 7.49 GHz that the fit will fail as soon as the data deviates from an ideal Hanger function shown in red. Hence, this model cannot be used for a spectrum as complicated as the one shown in Figure 2.3. Therefore, better models for describing resonators inside Purcell filters must be implemented.

Second, while the initial guess for the fit in Figure 4.1b is good, the one for the Hanger model is wrong. This is because they both use the peak_finder function from analysis_toolbox.py to find the tallest (smallest) peak (dip) in the data, which then becomes the initial guess for the f_0 parameters in the functions above. While this method works well for a Lorentzian fit (see Section 4.2), it very often fails for the Hanger model. This initial guess estimation must also be improved.



(a) Hanger model

(b) Lorentzian model

Figure 4.1: (a) Spectroscopy of readout resonator 1 from Figure 2.3. The amplitude of the complex signal is shown as a function of sweep frequency with a fit to the PycQED sloped Hanger function. (b) Spectroscopy of a readout resonator showing the signal power as a function of the sweep frequency, with a fit to a Lorentzian. The data in (b) was measured by Ants Remm (QuDev) on a different chip than the one presented in Section 2.1.1 (see Appendix B). The dashed traces in both images represent the initial guesses for the fits. These traces can be removed by setting the parameter *show_guess* to False.

The QuDev_transmon method that performs a complete resonator spectroscopy measurement and analysis is called find_resonator_frequency. Section 3.3 shows a typical call to this function. The *fitting_model* type that the analysis class should use is passed in to this function by the user (the default value is 'hanger'). Due to the issues mentioned above, currently the readout frequency is most often extracted by eye from interactive versions of plots like the one in Figure 3.1. The main reason for not optimizing this routine is that the issues mentioned above were only properly observed during testing in the lab, which occurred towards the end of the project. This occasion to test the developments added throughout this project was also used for the first proper testing in the lab of many new features in PycQED that were developed during the months of chip design.

4.2 Qubit Spectroscopy

The analysis routine for qubit spectroscopy is called Qubit_Spectroscopy_Analysis. This class was highly improved during the course of this project. Just like the Homodyne_Analysis discussed in the previous section, the original version of this routine also used peak_finder to find initial guesses for the fit function parameters, and then fitted the data to the same Lorentzian function as shown in the previous section. There was no support for finding and fitting to the second peak/dip at $f_{gf/2}$ described in Section 2.2.2, and even for just one Lorentzian, the fit would often fail because of inaccurate initial guesses.

The success of the fit depends very strongly on how good the initial guesses are. Therefore, peak_finder was optimized to find peaks or dips more reliably. However, after the first round of improvements, the fitting routine would still sometimes fail depending on the value of *num_sigma_threshold*. This parameter defines at how many standard deviations above the background noise the algorithm should start to search for peaks/dips. The background noise is calculated from the input percentage parameter *percentile* such that *percentile*/100 of the number of values stored in the data array are considered noise. The value of *num_sigma_threshold* varies greatly between data sets. Therefore, the current qubit spectroscopy fitting routine uses the maximum point in the data set for peaks, and the minimum point in the data set for dips. This has produced consistently good results like the ones in Figure 4.2.

The functionality of Qubit_Spectroscopy_Analysis was also extended to include a fit to a double Lorentzian function, thus adding the support for finding the $|g\rangle \leftrightarrow |f/2\rangle$ frequency. The algorithm first finds the tallest peak or the lowest dip. For simplicity let us assume it has found a peak. The algorithm then searches for the next two highest peaks, one to the left of the tallest at a frequency f_{left} and amplitude A_{left}, and one to the right of the tallest at a frequency f_{right} and amplitude A_{right}. The frequency of the tallest peak will be denoted by f_{main}. If $A_{left} > A_{right}$ then f_{left} is chosen to correspond to $f_{gf/2}$ and f_{main} to f_{ge}. If $A_{left} < A_{right}$, then f_{right} is chosen to represent f_{ge} and f_{main} is reassigned to $f_{gf/2}$[1].

Figure 4.2 shows the plots produced with this routine for a normal (qubit) spectroscopy (a) and a high power (qutrit) spectroscopy (b). The latter illustrates the power broadening effect discussed in Section 2.2.2, which widens the $|g\rangle \leftrightarrow |e\rangle$ transition peak compared to the $|g\rangle \leftrightarrow |f/2\rangle$ transition peak. The y-axis labels "S21 distance" refer to the distance away from the ground state. Qubit_Spectroscopy_Analysis combines the complex amplitude and phase of the acquired signal with the function calculate_distance_ground_state from analysis_toolbox.py. This function defines the ground state as the mean of the lowest 70 percent of the signal's real and imaginary parts. This makes sense, as the qubit is mostly in the ground state during qubit spectroscopy, and the ground state does not shift the readout resonator. Hence, it is indirectly represented by the noisy background signal in Figure 4.2.

The markers showing the peak frequencies and text boxes below each plot are additional minor improvements added during this project. The text boxes print the values and standard deviations of the peak frequencies and of the FWHM line widths from the fit results, as well as the old values for the respective transition frequencies stored in the qubit object.

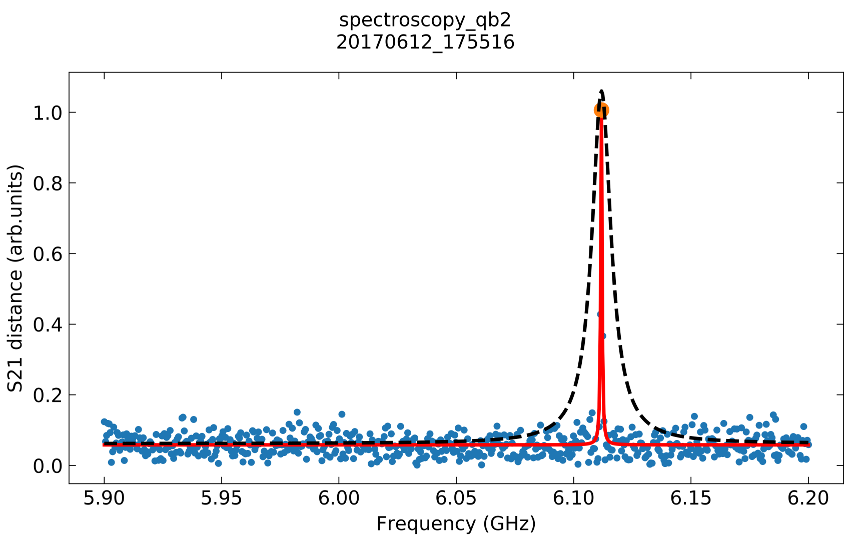
The QuDev_transmon method that performs a complete qubit or qutrit spectroscopy measurement and analysis is called find_frequency. This function needs to be told through the input parameter *method*, what type of spectroscopy measurement to perform (the default option is the continuous wave spectroscopy presented in Section 2.2.2), and whether to perform the analysis on a regular or a high power spectroscopy (*analyze_ef* False or True, respectively). A typical call to this function for a qubit and a qutrit is shown in Section 3.3.

4.3 Rabi Measurements

The Rabi measurement analysis class, Rabi_Analysis, fits the data that was rotated and projected based on calibration points as described in Section 3.2.3 to a cosine function of the form:

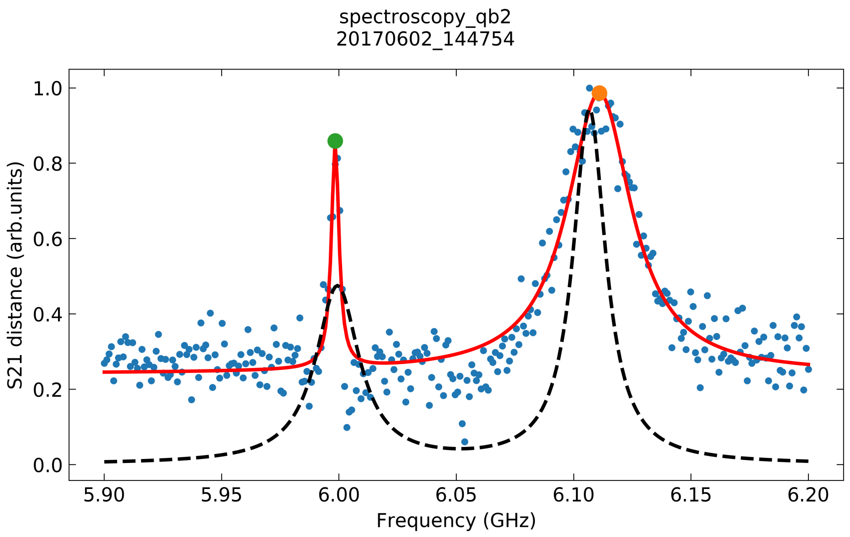
$$\mathrm{B}\cos(2\pi f A - \phi) + \mathrm{offset}, \tag{4.1}$$

[1]This is because $f_{gf/2}$ is always smaller than f_{ge}, and hence the peak corresponding to the former will be to the left of the one corresponding to the latter.



f0=6.11181 GHz ± (0.02) MHz
old f0=6.06785 GHz
kappa0=0.3330 MHz ± (0.02) MHz

(a) RO amplitude = 0.24V
Qubit drive power = -40 dBm
2^{14} averages



f0=6.11090 GHz ± (0.41) MHz
old f0=6.10900 GHz
kappa0=16.6474 MHz ± (0.69) MHz
f0_gf/2=5.99840 GHz ± (0.17) MHz
old f0_gf/2=0.00000 GH
kappa_gf=1.9275 MHz ± (0.25) MHz

(b) RO amplitude = 0.01V
Qubit drive power = 0 dBm
2^{14} averages

Figure 4.2: (a) Regular and (b) high power qubit spectroscopy using the settings shown below each image. The two readout amplitudes, which set the readout power, cannot be directly compared because the data in (a) was taken with an additional 23 dB of gain from an amplifier inserted before the readout input port of the dilution refrigerator. Thus, for (a) this voltage corresponds to around -36 dBm, and for (b) to around -42 dBm. The dashed traces in both images show the guesses for the fits. These traces can be removed by setting the parameter *show_guess* to False. The y-axis label refers to the distance of the measured qubit state from the known qubit ground state. See main text for details.

where B is the cosine amplitude, f is the frequency (in units of 1/Volt), A is the qubit drive amplitude which is swept during the measurement, ϕ is the phase, and offset is the offset of the cosine from zero.

The Rabi_Analysis class existed at the start of this project, but was incomplete. The routine would simply fit the magnitude of the signal ($\sqrt{\mathrm{I}^2 + \mathrm{Q}^2}$) to a cosine, calibration points were not being used, and the π- and $\pi/2$-pulses were not being calculated.

As a result of this project, PycQED can now be used to perform Rabi measurements and analyses on both qubits and qutrits. The QuDev_transmon method measure_rabi (which was already implemented before this project) drives Rabi oscillations between $|g\rangle \leftrightarrow |e\rangle$. Rabi oscillations on the second excitation are driven using measure_rabi_2nd_exc, which was implemented during this project. Even though the source code for Rabi_seq (Listing 3.6) and rabi_2nd_exc_seq (Listing 3.7) existed, they were improved to let the user choose between using 0, 2, or 4 calibration points for the qubit, and up to 6 *cal_points* for the qutrit. All this support for the Rabi measurement is brought together inside the QuDev_transmon method find_amplitudes (discussed in Section 3.2.1, and shown in Listing 3.3). This function conveniently lets the user choose between all these types of measurements by simply changing the input parameters *for_ef* (switch between qubit and qutrit), *last_ge_pulse* (bring qutrit to $|g\rangle$ before each measurement), *cal_points* (use *cal_points* or not), and *NoCalPoints* (how many *cal_points* to use). A typical call to the find_amplitudes method is shown in Section 3.3.

The results of Rabi measurements and analyses on a qubit and qutrit are shown in Figure 4.3. By passing to the Rabi_Analysis class the *for_ef* and *NoCalPoints* parameters, the routine uses the appropriate measurement calibration algorithm to process the raw data as described in Section 3.2.3. Thus, the same analysis class can be used to analyze the Rabi measurements for both the $|g\rangle \leftrightarrow |e\rangle$ and the $|e\rangle \leftrightarrow |f\rangle$ transitions, with only a minor change in the y-axis label as seen in Figure 4.3.

The red markers in Figure 4.3 show the π-pulse (top) and the $\pi/2$-pulse (middle), which are calculated from the fit results using:

$$\pi\text{-pulse} = \frac{1}{2 \cdot \text{frequency\_fit}} \tag{4.2}$$

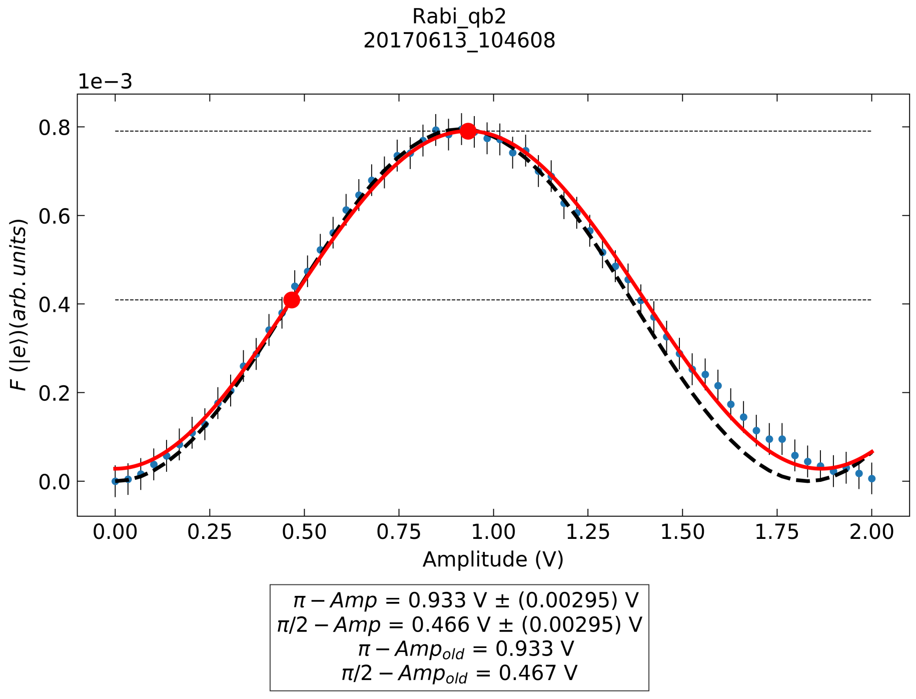
$$\pi/2\text{-pulse} = \frac{1}{4 \cdot \text{frequency\_fit}}. \tag{4.3}$$

Based on the definitions of the π-pulse and $\pi/2$-pulse given in Section 2.2.3, these expressions were obtained from Equation 4.1 for zero phase, by setting $2\pi f A = \pi$ and $2\pi f A = \pi/2$, respectively. In order to describe the physics of the qubit state, the projected data returned after calibration to *cal_points* is flipped as necessary to ensure it will always start at (closest to) zero (see Section 2.2.3 for why this must be so). This allows to fix the phase value at zero during the fit.

The dashed black line shows that the initial guess for the fit was very good. The frequency guess is obtained by computing the power spectrum of the data and finding the index of the largest response in the first half of the range[2]. Then the guess frequency is this index divided by the sweep range (in order to get the correct scaling, because the cosine frequency must have units 1/Volt). This method was adapted from [7].

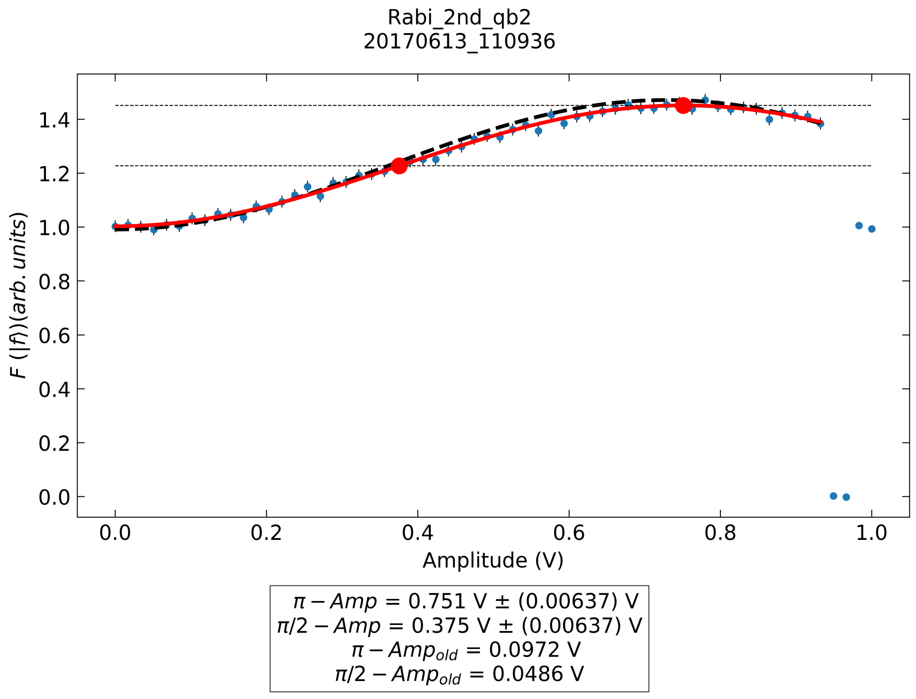
However, if the trace does not cover at least one full period of the cosine (as is the case for both plots in Figure 4.3), then this method for finding the guess frequency might fail. This is because the index will always be 1, and the inverse of the sweep range is not guaranteed to give a correct estimation of the oscillation frequency. In this case the routine assumes that the difference between the maximum and minimum values of the sweep points represents half

[2]Only half the spectrum is considered due to the symmetry of the Fourier transform; see Chapter 3 of [34].



$\pi - Amp$ = 0.933 V ± (0.00295) V
$\pi/2 - Amp$ = 0.466 V ± (0.00295) V
$\pi - Amp_{old}$ = 0.933 V
$\pi/2 - Amp_{old}$ = 0.467 V

(a) Rabi oscillations between $|g\rangle \leftrightarrow |e\rangle$
0 *cal_points*



$\pi - Amp$ = 0.751 V ± (0.00637) V
$\pi/2 - Amp$ = 0.375 V ± (0.00637) V
$\pi - Amp_{old}$ = 0.0972 V
$\pi/2 - Amp_{old}$ = 0.0486 V

(b) Rabi oscillations between $|e\rangle \leftrightarrow |f\rangle$
4 *cal_points*

Figure 4.3: Rabi measurement on a qubit (a) and qutrit (b) using the QuDev_transmon method find_amplitudes. The parameters with the "old" subscript show the previously measured values for the π-pulse and $\pi/2$-pulse amplitudes. The two red markers show the location of the π-pulse and $\pi/2$-pulse amplitudes, and the y-axis label denotes the population in the respective qubit state in arbitrary units. The last 4 data points in (b) are the $|g\rangle$ (lowest two) and $|e\rangle$ (higher two) calibration points, as explained in Section 3.2.3. The dashed black traces in both images show the initial guesses for the fits. These traces can be removed by setting the parameter *show_guess* to False.

the oscillation period T_{half}, and computes the frequency guess as f_guess $= 1/(2T_{half})$. This algorithm can be further improved in the future to deal with cases where the data trace is shorter than half a cosine period, which is useful in order to find very accurate values for the π- and $\pi/2$-pulse amplitudes by sweeping over a narrow amplitude range.

4.4 Ramsey Measurements

The Ramsey_Analysis class fits the calibrated data to the exponentially decaying cosine function

$$\mathrm{A}e^{-\Delta t/\mathrm{T}_2^\star}[\cos(2\pi f\Delta t + \phi) + \text{oscillation\_offset}] + \text{exponential\_offset}, \tag{4.4}$$

and calculates the new qubit frequency based on a single value of the *artificial_detuning* parameter, using Equation 2.8 introduced in Section 2.2.4. In Equation 4.4, A is the amplitude, f is the frequency of the Ramsey oscillations ($\omega_{Ramsey}/2\pi$ in Section 2.2.4), Δt is the time delay which is varied between the two $\pi/2$-pulses, $\mathrm{T_2}^\star$ is the averaged dephasing time, and ϕ is the phase. For more details about these parameters, see Section 2.2.4. The two offsets are the offsets from zero for the cosine and the exponential function, respectively.

The Ramsey_Analysis was already very well implemented in PycQED at the start of this project, and hence it was not greatly modified. The same sequences and QuDev_transmon *measure_* methods existed for the Ramsey measurement as for the Rabi measurement described in the previous section. Hence, the same additional functions and parameters as described in the previous section have also been added here, such that the user can now perform a Ramsey measurement and analysis for both qubits and qutrits by calling the find_frequency_T2_ramsey method. Figure 4.4 shows the output of this method for a Ramsey measurement on a qutrit with the option *last_ge_pulse*=True (see Section 3). This figure shows the same Ramsey data that was used to illustrate the calibration procedure in Figure 3.2b. A typical call to the find_frequency_T2_ramsey method is shown in Section 3.3.

Before this project, there was no support for performing the Ramsey calibration at two different artificial detuning values, which is accomplished as explained in Section 2.2.4. This measurement was implemented during this project in the new QuDev_transmon method, calibrate_ramsey. Additionally, the Ramsey_Analysis_multiple_detunings class, which performs the new analysis routine, was added to measurement_analysis.py. This class inherits from TD_Analysis, and uses the original Ramsey fitting routine to find the oscillation frequencies of both Ramsey measurements. These results are then used to find the correct new qubit frequency as explained in Section 2.2.4.

The input parameters to the calibrate_ramsey function are the same as the ones used with the find_frequency_T2_ramsey method. However, each sweep point passed in by the user will be duplicated, since the same measurement, but with a different value of the *artificial_detuning,* is performed for each sweep point. Hence, PycQED will process the Ramsey measurement with an input array for the time delay values that has twice as many sweep points.

The results produced by the calibrate_ramsey routine for a Ramsey measurement on a qubit is shown in Figure 4.5, for which the four cases described in Section 2.2.4 give:

$$\mathrm{f}_{ge} = \mathrm{f}_{spec} + \mathrm{f}_{Ramsey1} + \delta_1 = 5682.710590\text{MHz} \tag{4.5}$$
$$\mathrm{f}_{ge} = \mathrm{f}_{spec} - \mathrm{f}_{Ramsey1} + \delta_1 = 5680.708343\text{MHz} \tag{4.6}$$
$$\mathrm{f}_{ge} = \mathrm{f}_{spec} + \mathrm{f}_{Ramsey2} + \delta_2 = 5681.310502\text{MHz} \tag{4.7}$$
$$\mathrm{f}_{ge} = \mathrm{f}_{spec} - \mathrm{f}_{Ramsey2} + \delta_2 = 5680.708431\text{MHz}. \tag{4.8}$$

As expected, two of these four cases (4.6 and 4.8) produce very similar values, which are also closest to the estimated qubit frequency (f_{qubit_old} in Figure 4.5). Thus, the real qubit frequency is approximately 5.580708 GHz.

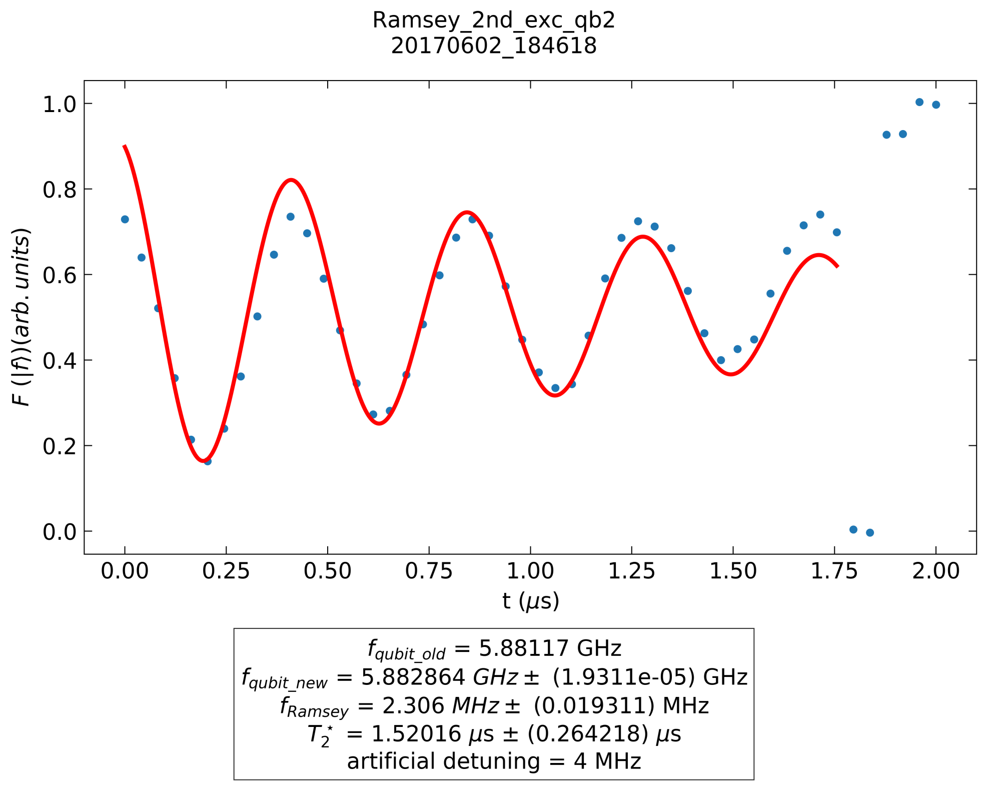


Figure 4.4: Ramsey measurement on the $|e\rangle \leftrightarrow |f\rangle$ transition for the artificial detuning value 4 MHz. f_{qubit_old} is the original qubit frequency that had to be calibrated, f_{qubit_new} is the correct qubit frequency identified as described in Section 2.2.4, f_{Ramsey} is the frequency of each oscillation obtained from the fit, and $T_2{}^\star$ is the averaged dephasing time. The last 6 data points are the $|g\rangle$ (lowest two), the $|e\rangle$ (next highest two), and the $|f\rangle$ (topmost two) calibration points, as explained in Section 3.2.3. The y-axis label denotes the population in the respective qubit state in arbitrary units.

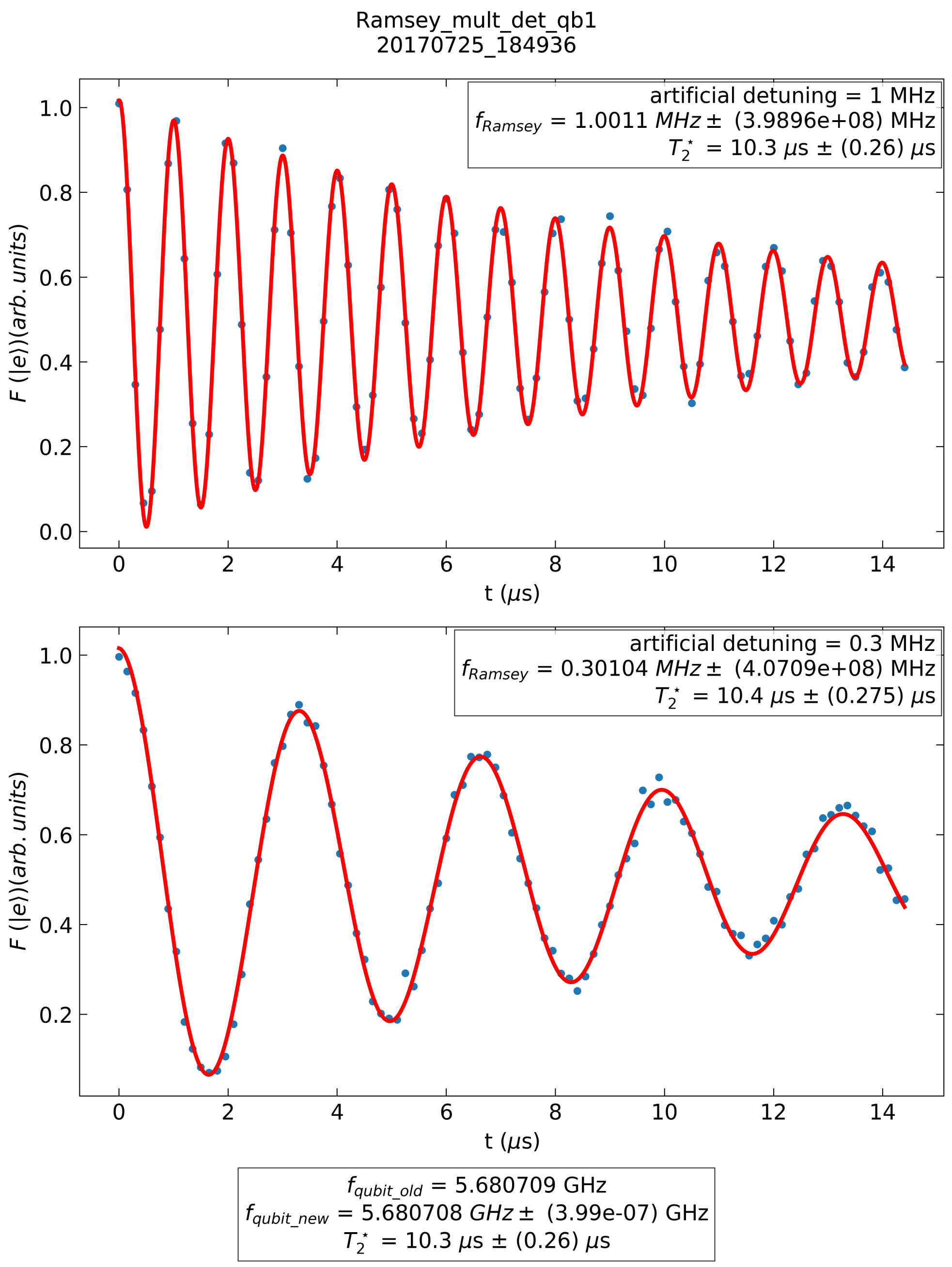


Figure 4.5: Ramsey measurement on the $|g\rangle \leftrightarrow |e\rangle$ transition for the artificial detuning values 1 MHz and 0.3 MHz. f_{Ramsey} is the frequency of each oscillation obtained from the fit, $T_2{}^\star$ is the averaged dephasing time, f_{qubit_old} is the original qubit frequency that had to be calibrated, and f_{qubit_new} is the correct qubit frequency identified as described in Section 2.2.4. The calibration points are not shown on these plots. The y-axis labels denote the population in the $|e\rangle$ state in arbitrary units.

4.5 T1 Measurements

The routine that analyzes the T1 data is the T1_Analysis class, which performs a fit to an exponentially decaying function of the form:

$$\mathrm{A}e^{-\Delta t/\mathrm{T}_1} + \text{offset}, \tag{4.9}$$

where A is the amplitude, Δt is the varying delay between the π-pulse and readout, T_1 is the desired energy relaxation time, and offset is the offset from zero. See Section 2.2.5 for details about the T1 calibration measurement.

The T1 measurement and analysis for the $|e\rangle$ state were already completely implemented and were working very well. During this project, support was added for measuring and analyzing the energy relaxation time for the $|f\rangle$ state. The QuDev_transmon method that is used to perform a complete T_1 measurement on a qubit or a qutrit is called find_T1. As described for the Rabi measurement in Section 4.3, this function also allows the user to easily choose how many calibration points to use (by setting *cal_points*, *no_cal_points*), and whether to perform the measurement on a qubit or qutrit (by setting *for_ef*, *last_ge_pulse*). As already mentioned in the previous two subsections, by passing in the *for_ef* and *NoCalPoints* parameters, T1_Analysis can be used for T_1 measurements on both qubits and qutrits. A typical call to find_T1 is shown in Section 3.3.

Figure 4.6 shows the result if using find_T1 to measure the energy relaxation time of the qubit $|e\rangle$ state.

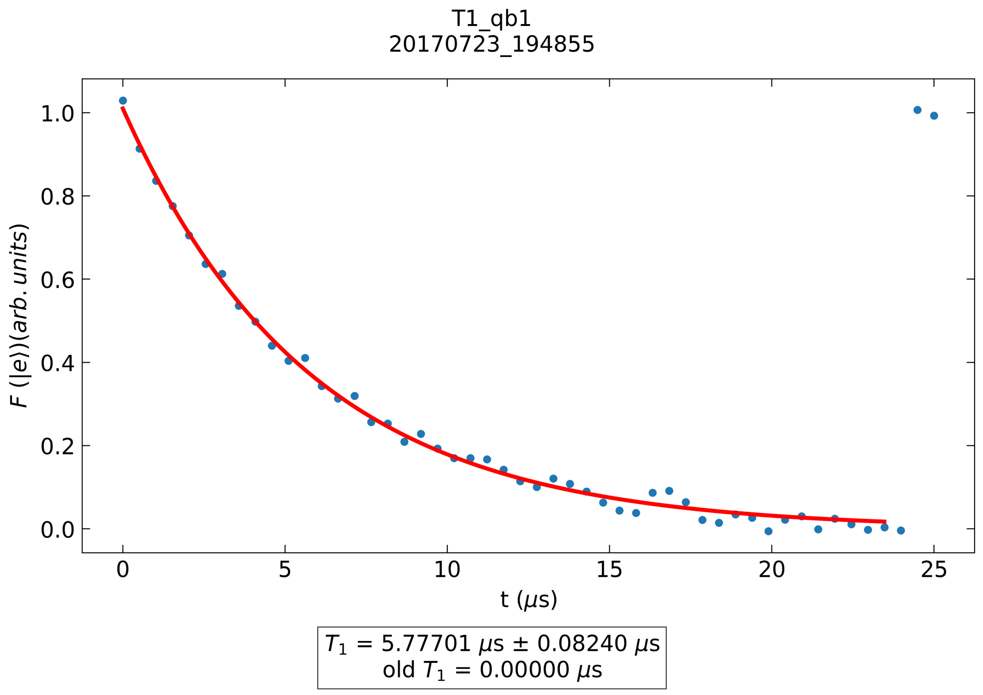


Figure 4.6: Energy relaxation time measurement for the qubit $|e\rangle$ state. *old T_1* shows the previously measured value of the energy relaxation time. Here it is zero because this was the first T_1 measurement performed on qubit 1. The last 4 data points are the $|g\rangle$ (lowest two) and the $|e\rangle$ (higher two) calibration points, as explained in Section 3.2.3. The fact that the $|g\rangle$ calibration points appear as a continuation of the measurement confirms that the qubit has indeed relaxed to the ground state. The y-axis label denotes the excited state population in arbitrary units.

4.6 DRAG Pulse Calibration

The QScale_Analysis class analyzes the DRAG pulse calibration data by fitting a constant line to the $X_{\frac{\pi}{2}}X_{\pi}$ data, and two general lines described by $y = mx + b$ to the $X_{\frac{\pi}{2}}Y_{\pi}$ and $X_{\frac{\pi}{2}}Y_{-\pi}$ data, as described in Section 2.2.6.

The quadrature scaling factor q_{scale} in PycQED is called *motzoi*[3], and has a slightly different definition compared to the one given in Section 2.2.6:

$$motzoi = \frac{q_{scale}}{\sigma_{Gauss}}, \tag{4.10}$$

where σ_{Gauss} is the standard deviation in seconds of the Gaussian pulse applied to the in-phase quadrature of the drive field. Hence, the *motzoi* parameter will require a smaller sweep range than the q_{scale}.

At the start of this project, the original DRAG pulse calibration measurement and analysis routines were using a slightly different technique to find the *motzoi* parameter. The routine would apply the two sets of pulses $X_{\pi}Y_{\frac{\pi}{2}}$ and $Y_{\pi}X_{\frac{\pi}{2}}$ for each *motzoi* value. In the ideal case, both these sets of pulses map the qubit population to an equal superposition of two adjacent qubit levels. This routine has not been tested. However, a calculation on the Bloch sphere of the probability for the qubit to be in the excited state after each set of pulses as a function of the drive frequency ω_d gives:

$$P(|e\rangle) = 0.5[1 - \cos(\omega_d \Delta t)\cos(\omega_d \Delta t)] \qquad (X_{\pi}Y_{\frac{\pi}{2}})$$
$$P(|e\rangle) = 0.5[1 - \cos(\omega_d \Delta t)\sin(\omega_d \Delta t)], \qquad (Y_{\pi}X_{\frac{\pi}{2}})$$

where Δt is a fixed time delay before measurement. A simulation of this result for some values of $\omega_d \Delta t$ is shown in Figure 4.7. As illustrated in Section 2.2.6, Figure 2.9, the q_{scale} (or *motzoi*) modifies the frequency spectrum of the drive field. Therefore, the information about the *motzoi* parameter is contained in ω_d. The original analysis routine, Motzoi_XY_Analysis, is expected to extract the ideal *motzoi* parameter from the drive frequency value at which the first intersection point of the two curves in Figure 4.7 occurs.

The traditional measurement and analysis routines for calibrating the DRAG pulse described in Section 2.2.6 have been implemented during this project for both a qubit and a qutrit. All the implemented routines follow the same structure as already described in the previous three sections. The QuDev_transmon method that performs a complete DRAG pulse calibration measurement and analysis for a qubit or a qutrit is called find_qscale, and it gives the user the same options as mentioned in the previous three sections. A typical call to the find_qscale method is shown in Section 3.3, and a DRAG pulse calibration measurement for the $|e\rangle$ state, obtained by using this method is shown in Figure 4.8.

[3]Named after the first author of reference [35], which was the first theoretical proposal of the DRAG pulse.

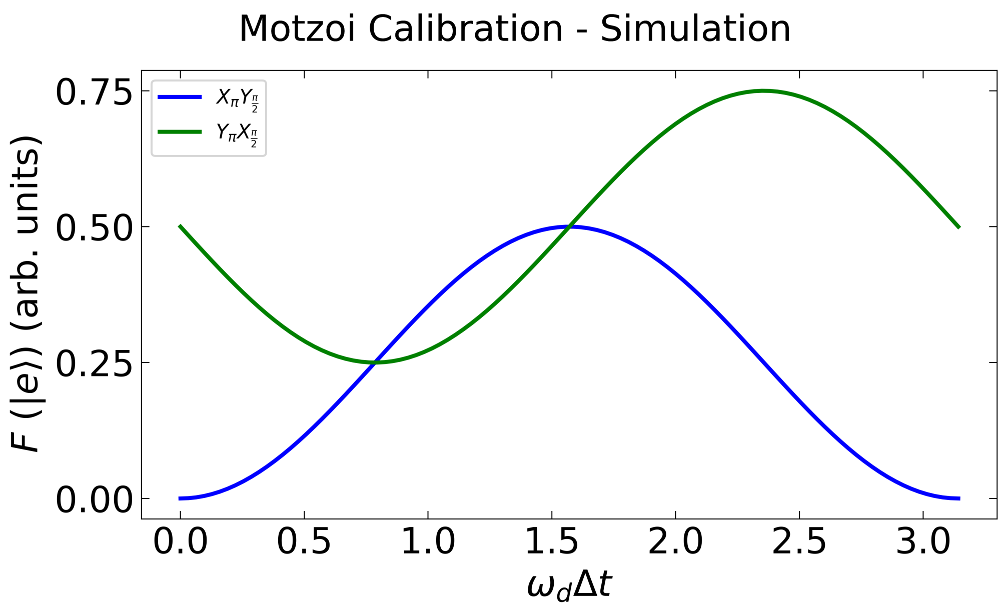


Figure 4.7: Simulation of the probability for the qubit to be in the excited state as a function of the qubit drive frequency ω_d. Δt is the fixed time delay before measurement, and the y-axis label denotes the population in the qubit $|e\rangle$ state in arbitrary units.

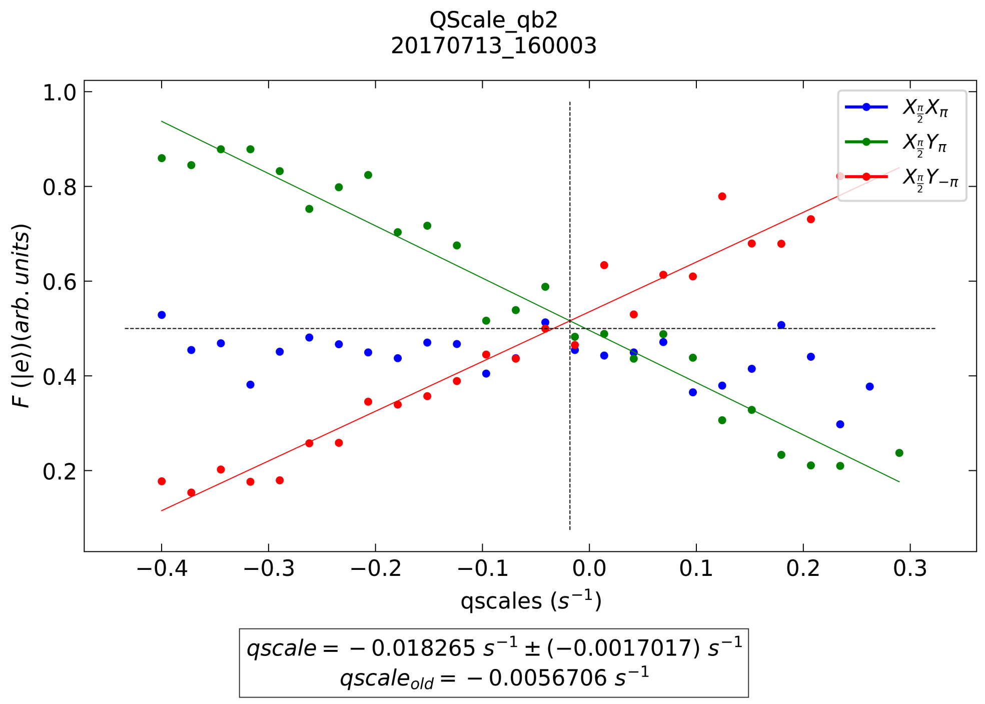


Figure 4.8: Calibration measurement for the q_{scale} factor of the DRAG pulse on the qubit $|e\rangle$ state. $qscale_{old}$ shows the previously measured value of q_{scale}. The x-axis label is the *motzoi* parameter but we denote it by q_{scale} to remove confusion. The y-axis label denotes the excited state population in arbitrary units

5. Conclusions and Outlook

As a result of this project, the PycQED software framework at QuDev now has full support for calibrating single qubit and qutrit operations. The features added throughout this project have already been used by other team members to find the readout and qubit transition frequencies using spectroscopy measurements, to measure the π-pulse and $\pi/2$-pulse amplitudes of the qubit drive signal with a Rabi measurement, to determine the accurate transition frequencies and the averaged dephasing time with a Ramsey measurement, to measure the energy relaxation time with a T1 measurement, and to calibrate the quadrature scaling factor for a DRAG pulse with a QScale measurement.

Currently, each of these measurements has its own *find_* method that must be called individually. The long-term goal will be to bring all these methods together under a single, highly optimized and very flexible calibrate_single_qubit_operations function, that will eventually replace the current LabVIEW-based QubitCalib routine (this framework is described in [7] and [8]).

Moreover, several of the analysis routines can be further improved in the future. In particular, more accurate fitting models must be found for the resonator spectroscopy analysis in order to correctly find resonance frequencies from more complex spectra that result from readout resonators coupled to Purcell filters. The Rabi analysis class can also be improved to fit to and extract the π-pulse and $\pi/2$-pulse amplitudes from traces that are shorter than half a cosine period, and that do not necessarily start at zero. This will be useful to extract the π-pulse and $\pi/2$-pulse amplitudes with more accuracy and precision.

The necessary next step in PycQED development, and a natural continuation of this project, is to build on the existing support for measuring, calibrating, and characterizing two-qubit gates. Several two-qubit gates together with single-qubit rotations form universal quantum gate sets (see chapter 4 of [36]), and thus PycQED will then be capable of implementing any quantum algorithm. This achievement, together with all the other benefits of PycQED highlighted in the introduction (Section 1), will equip our team with the best tools for scaling up towards the realization of a universal quantum computer.

A. Summary of Contributions to the PycQED Framework

QuDev_transmon

Added routines:

find_amplitudes Performs a complete Rabi calibration measurement on a qubit or a qutrit.
Updates the π-pulse amplitude parameter *amp180(_ef)*, and the *amp90_scale(_ef)* parameter[1] of the qubit object.
Performs calls to the measure_rabi and measure_rabi_2nd_exc methods of QuDev_transmon, which run the measurement for a qubit and qutrit, respectively, and to the Rabi_Analysis class inside measurement_analysis.py, which analyzes the data and computes the π- and $\pi/2$-pulse amplitudes.
Described in Section 3.2.1, and Section 4.3.

find_frequency_T2_ramsey Performs a complete Ramsey calibration measurement on a qubit or qutrit for one single value of the *artificial_detuning*.
Updates the qubit frequency parameter *f(_ef)_qubit*, and the $T_2^\star$ parameter *T2_star(_ef)* of the qubit object.
Calls the measure_ramsey and measure_ramsey_2nd_exc methods of QuDev_transmon, which run the measurement for a qubit and qutrit, respectively, and the Ramsey_Analysis class inside measurement_analysis.py, which analyzes the data and computes the new qubit frequency and the averaged dephasing time $T_2^\star$.
Described in Section 4.4.

calibrate_ramsey Same functionality as find_frequency_T2_ramsey above, but for two values of the *artificial_detuning*.
Performs calls to the QuDev_transmon methods measure_ramsey_multiple_detunings and measure_ramsey_2nd_exc_multiple_detunings, which run the measurement for a qubit and qutrit, respectively, and to the Ramsey_Analysis_multiple_detunings class inside the module measurement_analysis.py, which analyzes the data and computes the new qubit frequency and the averaged dephasing time $T_2^\star$.
Described in Section 4.4.

find_T1 Performs a complete measurement of the energy relaxation time T_1 on a qubit or qutrit.
Updates the T_1 parameter *T1(_ef)* of the qubit object.
Performs calls to the measure_T1 and measure_T1_2nd_exc methods of QuDev_transmon, which run the measurement for a qubit and qutrit, respectively, and to the T1_Analysis

[1] This parameter is defined as *amp90_scale*=($\pi/2$-pulse amplitude)/(π-pulse amplitude).

class inside measurement_analysis.py, which analyzes the data and finds the energy relaxation time.
Described in Section 4.5.

find_qscale Performs a complete calibration measurement for the q_{scale} factor of the DRAG pulse on a qubit or qutrit.
Updates the q_{scale} parameter *motzoi(_ef)* of the qubit object.
Calls the measure_qscale and measure_qscale_2nd_exc methods of QuDev_transmon, which run the measurement for a qubit and qutrit, respectively, and the QScale_Analysis class inside measurement_analysis.py, which analyzes the data and finds the best q_{scale} factor.
Described in Section 4.6.

AWG Sequences

Added routines:

Sequences for a Ramsey measurement with multiple detunings Functions that create and upload to the AWG the sequences for a Ramsey calibration measurement for two values of the *artificial_detuning*.
Ramsey_seq_multiple_detunings for a qubit; located in single_qubit_tek_seq_elts.py.
Ramsey_2nd_exc_seq_multiple_detunings for a qutrit; located in single_qubit_2nd_exc_seqs.py.

Sequences for a DRAG pulse calibration measurement Functions that create and upload to the AWG the sequences for calibrating the q_{scale} factor of a DRAG pulse.
QScale for a qubit; located in single_qubit_tek_seq_elts.py.
QScale_2nd_exc_seq for a qutrit; located in single_qubit_2nd_exc_seqs.py.

T1_2nd_exc_seq This function creates and uploads to the AWG the sequence for measuring the energy relaxation time of the qubit $|f\rangle$ state.

The functions that generate these sequences have not been specifically addressed in this thesis, but they follow the templates discussed in Section 3 for a Rabi measurement, specifically that shown in Listing 3.6 for a qubit, and that shown in Listing 3.7 for a qutrit.

Measurement Analysis

This work has only addressed the analysis routines needed for the single-qubit calibration measurements described in Section 2.2. The PycQED measurement analysis structure is discussed in Section 3.2.3, and specific classes are described in Section 4.

Improvements to existing code structure:

Inheritance tree MeasurementAnalysis is the lowest base class for all analysis classes. All time-domain analysis routines (Rabi, Ramsey, QScale, T1) use the intermediate base class TD_Analysis, which itself inherits from MeasurementAnalysis. Discussed in Section 3.2.3.

Option to use more calibration points PycQED can now calibrate a measurement based on 0, 2, 4, or 6 calibration points. Prior to this project, there was support for using only 4 calibration points. Discussed in Section 3.2.3.

Consistent plots All generated plots now use the same figure sizes, marker sizes, font sizes, and line widths. Saved figures have no overlapping features and fit nicely inside the figure window. Discussed in Section 3.2.3, and shown in Section 4.

Qubit_Spectroscopy_Analysis Additional support was implemented for finding the second peak/dip corresponding to the $|g\rangle \leftrightarrow |f/2\rangle$ transition frequency. The peak-/dip-finding routine used to establish initial guesses for a Lorentzian fitting function was improved to find the desired features more reliably (the routine would often fail before the start of this project). Discussed in Section 4.2.

Rabi_Analysis Minimal support existed for this routine at the start of this project. The work presented in this thesis has greatly improved the fitting and plotting routines, and has implemented code that computes the π-pulse and $\pi/2$-pulse amplitudes from the fit results. Discussed in Section 4.3.

Scaled sweep points The sweep points are scaled correctly for each measurement in order to have nicer plots (for example, the delay times in a Ramsey measurement are in μs, and the spectroscopy sweep frequencies are in GHz). This scaling was implemented in the MeasurementAnalysis base class, not for each particular routine. Hence, this feature can easily be used by all analysis classes. Discussed in Section 3.2.3.

Added routines:

Calibration routines based on 0 and 2 calibration points The calibration algorithm for 6 *cal_points* could reuse the existing routine for 4 *cal_points*, but the latter was not suited for 0 and 2 *cal_points*. The algorithm for 0 *cal_points* employs principal component analysis, while the one for 2 *cal_points* uses least squares minimization to project the two-dimensional data onto a single axis. Discussed in Section 3.2.3.

Ramsey_Analysis_multiple_detunings Analyzes data from the calibrate_ramsey measurement routine (see above). Discussed in Section 4.4.

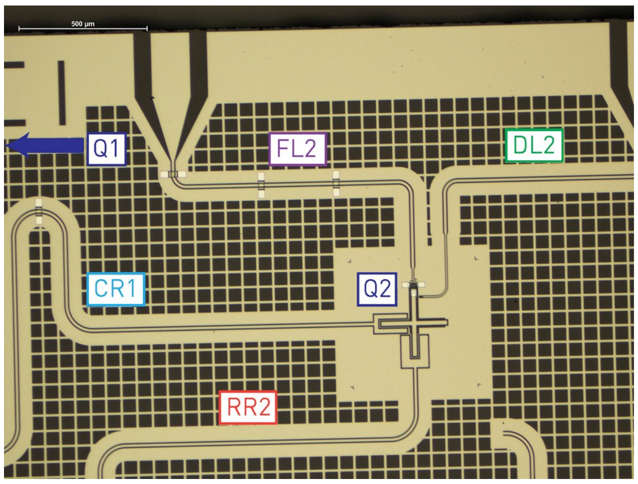
QScale_Analysis Analyzes data from the find_qscale measurement routine (see above). Discussed in Section 4.6.

B. Qubit Designs Used for the Measurements in this Work

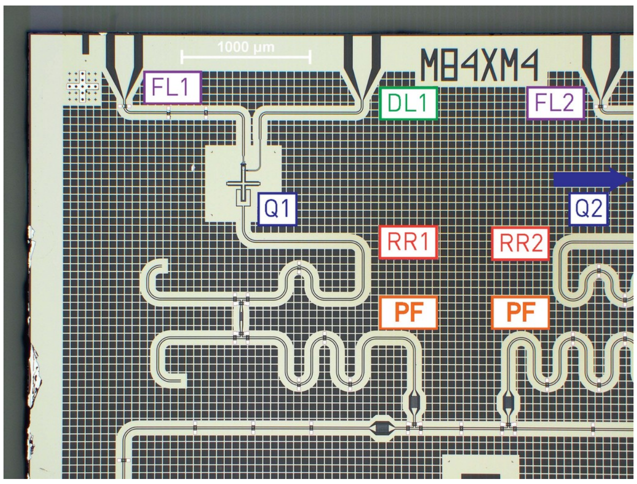
The work presented in this thesis was carried out in parallel with the team's efforts to test new qubit designs. Hence, the measurements shown throughout this thesis have been performed on several different samples. Table B.1 summarizes the measurement results shown in this thesis, and the chips on which they were performed. The latter are listed in the chronological order in which they were fabricated. The chip designs that were not discussed in Section 2.1.1 are shown in Figure B.1.

Sample name	**Short description**	**Measurements**	**Reference**
M65B1	4 qubit chip	Lorentzian resonator spectroscopy	Figure 4.1b
M81B1	See Section 2.1.1	Full resonator spectrum	Figure 2.3
		Resonator dispersive shift	Figure 2.4
		Raw data plots	Figure 3.1
		Measurement calibration plots	Figure 3.2
		PycQED usage example code	Figures 3.4-3.6
		Resonator spectroscopy	Figure 4.1
		Qubit spectroscopy	Figure 4.2
		Rabi measurements	Figure 4.3
M84XMC1	First two-qubit swissmon design	Ramsey for 1 detuning	Figure 4.4
		T_1 measurements	Figure 4.6
		DRAG pulse calibration measurement	Figure 4.8
M84XM4	Second two-qubit swissmon design	Ramsey for 2 detunings	Figure 4.5

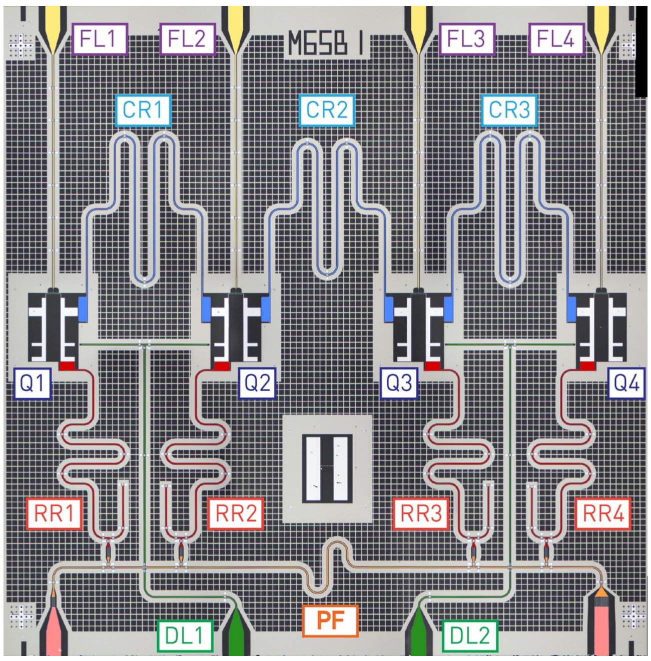
Table B.1: Summary of the qubit design samples used for the measurements presented in this thesis.



(a) M84XMC1



(b) M84XM4



(c) M65B1

Figure B.1: The other three chip designs on which some of the measurements in this thesis have been performed. Q denotes a qubit, RR denotes a readout resonator, FL denotes a flux line, DL denotes a qubit drive line (charge line in Section 2.1.1), CR denotes a coupling resonator, and PF denotes a Purcell filter. The designs in (a) and (b) contain 2 qubits, and the blue arrows point to the location of the qubit not shown in these images. The design in (c) contains 4 qubits.

Acknowledgements

I would like to kindly thank Ants Remm for his great help and support as I was learning to understand and navigate the PycQED structure. I am also very grateful for his always-valuable input during PycQED development. I would also like to warmly thank Dr. Christian Kraglund Andersen, Johannes Heinsoo, and Ants Remm for their help and guidance in the lab. The experimental results presented in this paper could not have been successfully measured without their extensive knowledge about the entire experimental setup, and their experience with circuit QED measurements.

I would like to express my sincere gratitude to Professor Andreas Wallraff for the privileged opportunity to pursue this project with QuDev. I am also very grateful to Dr. Christian Kraglund Andersen for his kind guidance, his constructive feedback, and overall great support throughout this entire project. Thank you also to all the other members of the group, who have been always warm and helpful towards me. In particular, I thank Yves Salathé for everything he has taught me about FPGAs and how they are used in the lab. I gained a deeper understanding about digital data acquisition, and this knowledge has helped me time and again throughout this project.

Lastly, I would like to thank Professor Leonardo DiCarlo and his team for starting this collaboration, and for their warm support during the PycQED development.

References

[1] The Economist. *The Economist*. 1843. URL: http://www.economist.com/news/essays/21717782-quantum-technology-beginning-come-its-own#s-3 (visited on 06/19/2017) (cit. on p. 1).

[2] IBM Computer Manufacturing Company. *IBM Quantum Experience*. 2016. URL: https://www.research.ibm.com/ibm-q/ (visited on 06/19/2017) (cit. on p. 1).

[3] Simon J. Devitt. "Performing quantum computing experiments in the cloud". In: *Phys. Rev. A* 94 (3 2016), p. 032329. DOI: 10.1103/PhysRevA.94.032329. URL: http://link.aps.org/doi/10.1103/PhysRevA.94.032329 (cit. on p. 1).

[4] Christine Corbett Moran. "Quintuple: a Python 5-qubit quantum computer simulator to facilitate cloud quantum computing". In: *arXiv:1606.09225 [quant-ph]* (June 2016). arXiv: 1606.09225. URL: http://arxiv.org/abs/1606.09225 (visited on 07/06/2016) (cit. on p. 1).

[5] Microsoft Research. *Microsoft Research Station Q*. 2005. URL: https://stationq.microsoft.com/ (visited on 06/19/2017) (cit. on p. 1).

[6] QuTech. *Microsoft and TU Delft collaboration started*. 2017. URL: https://qutech.nl/microsoft-and-tu-delft-collaboration-started/ (visited on 06/19/2017) (cit. on p. 1).

[7] Tim Menke. "Realizing a Calibration Program for Superconducting Qubits". MA thesis. ETH Zurich, Aug. 2013. URL: http://qudev.phys.ethz.ch/content/science/Documents/semester/Tim_Menke_Semester_Thesis_130829.pdf (cit. on pp. 1, 14, 39, 47).

[8] Andreas Landig. "Software for arbitrary single qubit and qutrit gate calibration". MA thesis. ETH Zurich, Nov. 2013. URL: http://qudev.phys.ethz.ch/content/science/Documents/semester/Andreas_Landig_semesterthesis_131020.pdf (cit. on pp. 1, 11–13, 47).

[9] Damian Steiger and Thomas Hner. *ProjectQ*. 2016. URL: https://projectq.ch/ (visited on 06/19/2017) (cit. on p. 1).

[10] D. S. Steiger et al. "ProjectQ: An Open Source Software Framework for Quantum Computing". In: *arXiv:1612.08091* (2016). URL: https://arxiv.org/abs/1612.08091 (cit. on p. 1).

[11] Robert S. et al. "A Practical Quantum Instruction Set Architecture". In: *arXiv:1608.03355* (2017). URL: http://arxiv.org/abs/1608.03355 (cit. on p. 1).

[12] TutorialsPoint. *Learn Python Programming Language*. 2006. URL: https://www.tutorialspoint.com/python/python_classes_objects.htm (visited on 07/20/2017) (cit. on pp. 1, 15, 16).

[13] Marek Pechal. "Microwave photonics in superconducting circuits". PhD thesis. ETH Zurich, 2016. URL: http://qudev.phys.ethz.ch/content/science/Documents/phd/MarekPechal_PhDThesis_v4_160928.pdf (cit. on pp. 3, 4, 6–8, 10–14).

[14] Alexandre Blais et al. "Cavity quantum electrodynamics for superconducting electrical circuits: An architecture for quantum computation". In: *Phys. Rev. A* 69 (6 2004), p. 062320. DOI: 10.1103/PhysRevA.69.062320. URL: https://link.aps.org/doi/10.1103/PhysRevA.69.062320 (cit. on pp. 3, 4, 6).

[15] Matthias Baur. "Realizing quantum gates and algorithms with three superconducting qubits". PhD thesis. ETH Zurich, Mar. 2012. URL: http://qudev.phys.ethz.ch/content/science/Documents/phd/PhD_Thesis_Baur_Matthias.pdf (cit. on pp. 3, 6–8, 13, 14, 22, 31, 35).

[16] J. Majer et al. "Coupling superconducting qubits via a cavity bus". In: *Nature* 449.7161 (Sept. 2007), pp. 443–447. ISSN: 0028-0836. DOI: 10.1038/nature06184. URL: http://dx.doi.org/10.1038/nature06184 (cit. on p. 4).

[17] Eyob A. Sete, John M. Martinis, and Alexander N. Korotkov. "Quantum theory of a bandpass Purcell filter for qubit readout". In: *Phys. Rev. A* 92 (1 2015), p. 012325. DOI: 10.1103/PhysRevA.92.012325. URL: http://link.aps.org/doi/10.1103/PhysRevA.92.012325 (cit. on p. 4).

[18] Jens Koch et al. "Charge-insensitive qubit design derived from the Cooper pair box". In: *Phys. Rev. A* 76.4, 042319 (2007), p. 042319. DOI: 10.1103/PhysRevA.76.042319. URL: http://link.aps.org/abstract/PRA/v76/e042319 (cit. on pp. 4, 6).

[19] L. Steffen et al. "Deterministic quantum teleportation with feed-forward in a solid state system". In: *Nature* 500 (2013), pp. 319–322. DOI: 10.1038/nature12422. URL: http://www.nature.com/nature/journal/v500/n7462/full/nature12422.html?WT_ec_id=NATURE-20130815 (cit. on p. 5).

[20] A. Wallraff et al. "Strong coupling of a single photon to a superconducting qubit using circuit quantum electrodynamics". In: *Nature* 431 (2004), pp. 162–167. DOI: 10.1038/nature02851. URL: http://www.nature.com/nature/journal/v431/n7005/full/nature02851.html (cit. on pp. 4, 6).

[21] L. DiCarlo et al. "Demonstration of two-qubit algorithms with a superconducting quantum processor". In: *Nature* 460.7252 (July 2009), pp. 240–244. ISSN: 0028-0836. DOI: 10.1038/nature08121. URL: http://dx.doi.org/10.1038/nature08121 (cit. on p. 4).

[22] D. I. Schuster et al. "AC Stark shift and dephasing of a superconducting qubit strongly coupled to a cavity field". In: *Phys. Rev. Lett.* 94.12 (Apr. 2005), p. 123602. DOI: 10.1103/PhysRevLett.94.123602. URL: http://link.aps.org/abstract/PRL/v94/e123602 (cit. on p. 7).

[23] D. Vion et al. "Rabi oscillations, Ramsey fringes and spin echoes in an electrical circuit". In: *Fortschritte der Physik* 51 (2003), pp. 462–468. URL: http://www3.interscience.wiley.com/cgi-bin/abstract/104528217/ABSTRACT (cit. on p. 8).

[24] Simon Storz. "Spectroscopy Automation and Sample Characterization of Superconducting Qubits". MA thesis. ETH Zurich, 2016. URL: http://qudev.phys.ethz.ch/content/science/Documents/master/Simon_Storz_MastersThesis.pdf (cit. on pp. 8, 10).

[25] Rodney Loudon. *The Quantum Theory of Light.* Oxford U, 2000 (cit. on p. 10).

[26] C. J. Foot. *Atomic Physics.* Oxford University Press, 2007 (cit. on p. 10).

[27] J. M. Gambetta et al. "Analytic control methods for high-fidelity unitary operations in a weakly nonlinear oscillator". In: *Phys. Rev. A* 83.1 (Jan. 2011), pp. 012308–13. DOI: 10.1103/PhysRevA.83.012308 (cit. on p. 14).

[28] TutorialsPoint. *Learn C++ Programming Language.* 2006. URL: `https://www.tutorialspoint.com/cplusplus/cpp_classes_objects.htm` (visited on 07/15/2017) (cit. on pp. 15, 16).

[29] D. I. Schuster. "Circuit Quantum Electrodynamics". PhD thesis. Yale University, 2007 (cit. on p. 22).

[30] Andrew Collette. *HDF5 for Python.* 2013. URL: `http://www.h5py.org/` (visited on 06/21/2017) (cit. on p. 22).

[31] The HDF Group. *Hierarchical Data Format Version 5 (HDF5) Technologies.* 2016. URL: `https://www.hdfgroup.org/why-hdf/` (visited on 06/21/2017) (cit. on p. 22).

[32] American Physical Society. *APS Physical Review Letters Contributors Guidelines.* 2017. URL: `https://journals.aps.org/prl/info/infoL.html` (visited on 06/21/2017) (cit. on p. 24).

[33] JD Long. *Principal Component Analysis (PCA) vs Ordinary Least Squares (OLS): A Visual Explanation.* 2010. URL: `https://www.r-bloggers.com/principal-component-analysis-pca-vs-ordinary-least-squares-ols-a-visual-explanation/` (visited on 06/20/2017) (cit. on p. 28).

[34] Richard G. Lyons. *Understanding Digital Signal Processing.* 3rd. Upper Saddle River, NJ, USA: Prentice-Hall, 2011. ISBN: 978-0-13-702741-5 (cit. on p. 39).

[35] F. Motzoi et al. "Simple Pulses for Elimination of Leakage in Weakly Nonlinear Qubits". In: *Phys. Rev. Lett.* 103.11, 110501 (2009), p. 110501. DOI: `10.1103/PhysRevLett.103.110501`. URL: `http://link.aps.org/abstract/PRL/v103/e110501` (cit. on p. 45).

[36] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information.* Cambridge University Press, 2000. ISBN: 521635039 (cit. on p. 47).

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):

SINGLE-QUBIT GATES CALIBRATION IN PYCQED USING SUPERCONDUCTING QUBITS

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
BALASIU	STEFANIA

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, 17.08.2017	[illegible]

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