MASTER'S THESIS

A Categorical Order Theory of Pulse Scheduling in Gate-Based Quantum Computing

Via a Time and Frequency Ordering of Integrable Functions

AXEL E. ANDERSSON

Department of Engineering Mathematics and Computational Science
Department of Microtechnology and Nanoscience
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2022

A Categorical Order Theory of Pulse Scheduling in Gate-Based Quantum Computing
Via a Time and Frequency Ordering of Integrable Functions
AXEL E. ANDERSSON

Supervisor: Miroslav Dobsicek, Microtechnology and Nanoscience
Examiner: Jonas Bylander, Microtechnology and Nanoscience

Master's Thesis 2022
Interdisciplinary work in Engineering Mathematics and Computational Science, and
Microtechnology and Nanoscience

Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Approved seminar presenting degree project on June 3rd 2022.
Approved in plagiarism search on June 10th 2022.
Approved report on June 10th 2022.

Typeset in LaTeX
Diagrams produced with `tikzcd`
Printed by Chalmers Reproservice
Gothenburg, Sweden 2022

Abstract

We define a preorder on a vector space of complex valued integrable functions on the non-negative real numbers. This preorder is then used to develop a scheduling theory for microwave pulse schedules with an application for quantum computer experiments on superconducting circuits. The scheduling theory is further developed in a categorical framework using a subcategory of **Ord**, the category of preordered sets and order-preserving mappings between them.

The developed theory is then applied to create a Python library which translates IBM OpenPulse schedules to Quantify schedules. Further, this library was then used to conduct single qubit characterisation experiments. Performed experiments include: resonator spectroscopy, two-tone spectroscopy, Rabi oscillation, relaxation time (T_1) and qubit state discrimination experiments.

Acknowledgements

Firstly, I want to thank Miroslav Dobsicek for many insightful discussions and for giving me the opportunity to write this thesis. I also want to thank Eleftherios Moschandreou and Giovanna Tancredi for interesting discussions, technical advice, general guidance throughout this work and most of all for their patience with my simultaneously pedantic and occasionally ignorant input. Without their help I would surely not have been able to write this thesis. Furthermore, I also want to thank Jonas Bylander for trusting me with this work in the MC2 department, given that my background is in engineering mathematics and computer engineering and not physics or nanoscience. I also want to thank Anton Frisk Kockum, Laura García-Álvarez, Pontus Vikstål and Giulia Ferrini for holding a very interesting course in quantum computation - since this was how I originally stumbled into this field.

Finally, I want to thank Aila Särkkä, Jeffrey Steif, Per Salberger, and Håkan Andreasson for their wonderful lectures in spatial statistics, integration theory, modern algebra, and functional analysis respectively. They were by far my favourite lectures throughout my five years at Chalmers.

Contents

1

Introduction

The quantum circuit model abstracts away from the underlying physical implementation of gates and measurements on a quantum computer [38]. This abstraction is useful because it allows quantum algorithms to be developed independently of particular quantum computers. Under the hood, quantum operations are typically implemented with classical time-dependent control signals coupled to quantum systems. The signals are also often referred to as *pulses*, to emphasize the desire to make them as brief as possible. Control hardware such as arbitrary waveform generators, flux sources, and lasers emit these pulses to orchestrate the synchronous emission of calibrated control fields [45, p. 31]. The purpose of these control fields is to manipulate qubits of the target quantum computer in a way which implements the computation that we are interested in. Finding the optimal way to design and schedule these pulses is an active area of research [20, 26, 35, 47]. Software development in this research field increases the performance of existing control stacks. This motivates a hardware & software "co-design" [23] in order to extract the maximum performance from contemporary quantum computers [45, 47].

Recently, there is a growing interest in generalising upon the control flow of the software used in the instrument orchestration. The interest stems from the fact that existing quantum assembly languages either incorporate too high-level constructs to be directly implemented by a micro-architecture, or are too low-level to be able to provide a comprehensive abstraction of the quantum hardware which can support the required control flow [30]. This has led to an intermediary abstraction layer of quantum computation to take shape [30, 38, 47] wherein quantum operations are associated to their control pulses, but the control pulses are abstracted from the control instruments. A particular boon of introducing such a layer is that quantum computing experiments which rely on time dependent dynamics, such as characteri-

sation of decoherence and crosstalk [18], dynamical decoupling [2, 7, 46], dynamically corrected gates [14], and gate parallelism [41] can be specified unambiguously independent of control hardware.

One such hardware agnostic descriptive language is called OpenPulse, which was developed by IBM [32, 38]. The main contribution of this thesis is a Python library which translates OpenPulse schedules to Quantify schedules. Quantify [49] is a Python based data acquisition platform focused on quantum computing and solid-state physics experiments, built on top of QCoDeS [48]. The reason Quantify is useful to us is because it has good support for QBLOX devices, which are new quantum hardware bought by Chalmers.

The translation that the library provides is underpinned by an order theory of complex valued integrable functions on the real number line. Specifically, a preordered function space is constructed which allows one to generate graphs from pulse schedules. These graphs are used as stepping stones in the translation process.

Moreover, a category of pulse schedules is constructed wherein pulse schedules for large-scale quantum computer experiments can be generated with ease. The categorical formulation is an independent result, and it is not needed for the translation of OpenPulse schedules to Quantify schedules. It is included in the thesis with the hope that such a formulation could potentially assist in a future co-design theory of quantum computer experiments. An important distinction that should be made is that this category differs from the categorical quantum mechanical foundations such as those constructed by Coecke, Heunen and Kissinger [10, 24, 34]. These constructions are for studying the physical processes involved in quantum information theory, such as e.g. entanglement and state teleportation. We will not incorporate the Hilbert spaces of quantum information theory.

A closer cousin of the theory considered in this thesis is signal processing and control theory [6] with the difference that most classical control theory do not incorporate non-linear subsystems, let alone quantum subsystems. The closest related domain is probably quantum control theory which is recent but rapidly developing [11, 12, 27].

The reader is assumed to be familiar with engineering concepts, such as basic signal processing, the Fourier transform and LC circuits. The remaining sections of this chapter describe some background material in quantum computation, specifically focused on quantum computers containing superconducting qubits that are controlled with microwave pulses. Readers with background in quantum computer engineering, physics or RF engineering will most likely find these topics very familiar.

In Chapter 2, the scheduling theory is developed. It is primarily based order theory and category theory and uses set-theoretic arguments. Readers of this chapter will benefit from previous exposure to order theory, in particular.

In Chapter 3, the implementation of the library is presented. Chapter 3 includes a very limited amount of code, and as such previous knowledge of Python is not strictly required to follow its development. However, readers will benefit from familiarity with standard programming concepts.

In Chapter 4, the results of standard quantum computer experiments which were conducted using the library are presented. Quantum computer engineers, specifically those working with superconducting qubits, will most likely be familiar with these experiments. Readers with other backgrounds would benefit from first reading the background sections in Chapter 1, which hopefully cover the minimum needed material to understand these experiments.

1.1 The Superconducting Qubit

There is already a lot of excellent literature written about qubits and their role in quantum computation [17, 33, 35]. Therefore, only the basic idea is covered in this chapter and it is in every sense an informal description. There are many kinds of qubits, but only the superconducting qubit is covered here.

Recall from electrical engineering the concept of an LC circuit, a.k.a. tank circuit, that is, a circuit with a capacitor and an inductor coupled to each other, without

any other components. If we measure the voltage across any component, say, the capacitor, the measured signal will be zero assuming that there is no initial voltages in either of the components. If, however, we start the circuit in some initial state, say, by charging the capacitor, and then measure the same voltage, then the signal will now be oscillatory. The capacitor stores energy in its electric field and the inductor stores energy in its magnetic field. Hence, the voltage signal is oscillatory because the energy is oscillating between between being stored in the capacitor and the inductor. The period of this oscillation is directly related to the physical characteristics of the circuit, such as the material of the wire, the inductance of the inductor, and the capacitance of the capacitor. The corresponding frequency of oscillation is known as the *characteristic frequency* of the system, denoted by ω.

The voltage signal that we measure depends directly on the characteristic frequency of the system. It also depends on the initial state of the circuit, although after some time the signal will decay to zero since energy is "lost" to the environment mostly due to heat transfer. This phenomenon will occur with classical electrical components.

When an LC circuit is made using a superconducting material, then we get what is known as a *quantum harmonic oscillator* (QHO) [35, p. 4]. The lowest energy level of such a circuit is known as the ground state and the energy levels above the ground state are known as excited states. A famous result in physics states that quantum harmonic oscillators have quantized energy levels and moreover the energy levels are equidistantly spaced $\hbar\omega$ apart [35, p. 4], where $\hbar$ is the reduced Planck's constant. The creation of an encoding which assigns meaning to the energy levels is essentially the basis of quantum computation. A QHO can manually be put into an excited state by externally affecting the voltage across its components with a control pulse sent at the characteristic frequency of the circuit. In quantum computation with superconducting qubits, this is done with microwave control pulses. However, the equidistant spacing of the energy levels becomes a problem, because such a control pulse could excite other the energy states in the system [35, p. 5]. It turns out that the equidistant energy levels are a consequence of using linear circuit components. If the inductor in the LC circuit is replaced with what is known as a *Josephson junction* (JJ), which is essentially an inductor with a non-linear profile, then we can force the energy levels of the QHO to be not equidistant, see Figure 1.1.

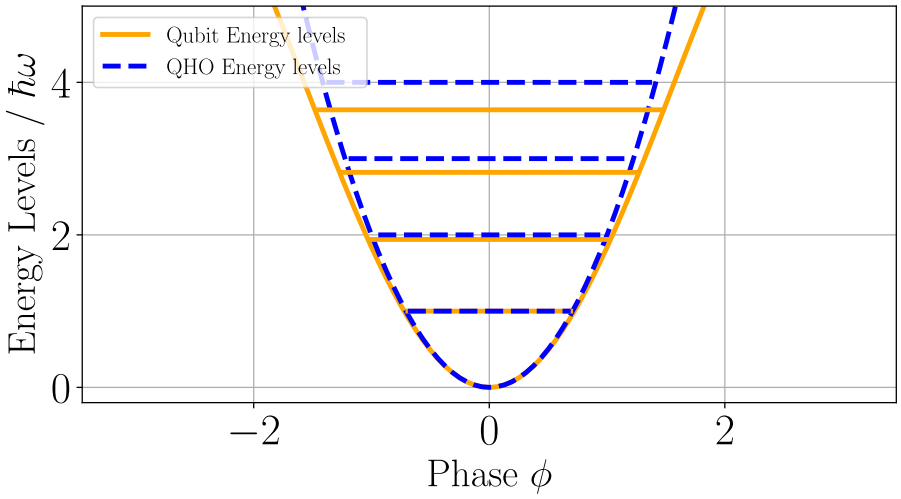


Figure 1.1: Energy levels of an example QHO (blue, dashed) and the corresponding energy levels of a qubit (orange, filled). Note that the former are spaced in integer multiples of $\hbar\omega$, whereas the latter are not equidistantly spaced.

The resulting circuit is what is meant by a superconducting *qubit*. In particular, this means that the characteristic frequency is no longer the same for all energy levels, due to the anharmonicity introduced by the non-linear inductance. We can therefore address individual pairs of energy levels with control pulses. The two lowest energy levels in a qubit are usually called $|0\rangle$ and $|1\rangle$ and form the two-level computational basis of a qubit. Note that we could use any of the energy levels as our computational basis. However, usually the lowest two energy levels are used because they have the highest separability, and moreover the qubit will naturally go to the ground state $|0\rangle$. A qubit state can be represented with the Bloch sphere, visualized in Figure 1.2. The Bloch sphere representation is useful because it provides a geometric understanding of a qubit state.

Importantly, we have that the qubit can be in a superposition of energy levels $|0\rangle$ and $|1\rangle$, which is written $\alpha\,|0\rangle + \beta\,|1\rangle$ for $\alpha, \beta \in \mathbb{C}$. A measurement on this qubit state (in the computational basis) yields the result 0 with probability $|\alpha|^2$ and the result 1 with probability $|\beta|^2$ [33, p. 7]. If $\alpha\,|0\rangle + \beta\,|1\rangle$ is a qubit state, then we must have that $|\alpha|^2 + |\beta|^2 = 1$, because a qubit is a two level system. Note that

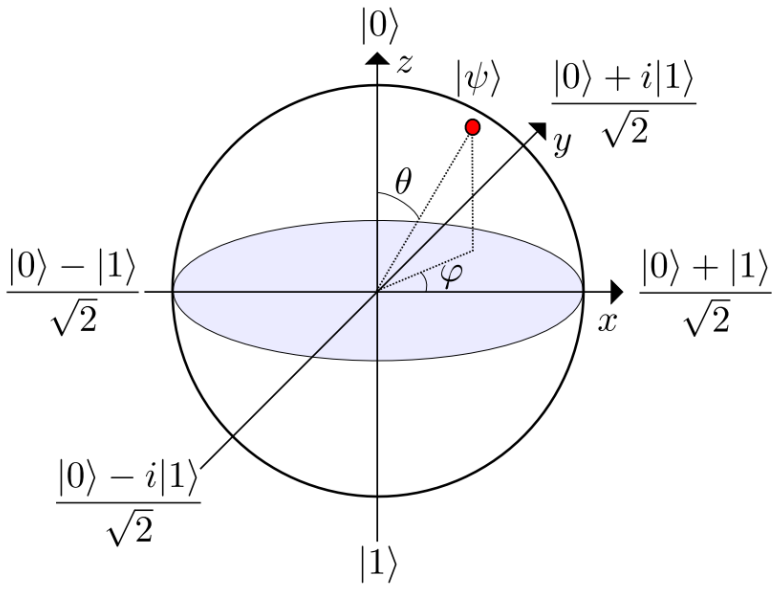


Figure 1.2: Example qubit state in red. Image from [33, p. 7]. To convert an arbitrary superposition of $|0\rangle$ and $|1\rangle$ to a point on the sphere, one can use the mapping $|\psi\rangle = \cos\frac{\theta}{2}|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}|1\rangle$.

the state of a superconducting qubit is not stationary relative to a fixed axis, as it would appear on the Bloch sphere because the energy in the qubit circuit is always oscillating between the JJ and the capacitor. In Figure 1.2, the z-axis is used as the energy level quantization axis, so this oscillation would correspond the qubit state constantly precessing around the z-axis [35, p. 12] at the qubit characteristic frequency[1].

The development above tells us how to control qubit states, but it does not tell us how read qubit states. This is done by connecting each qubit to a *resonator*. A resonator is simply a conducting mechanical device (without any electrical components) which has some particular characteristic frequency of its own. The purpose of the resonator is simply to "absorb" control pulses sent to it on a *readout line*. The qubit and the resonator are connected together, so they therefore also have a combined

[1]The "qubit characteristic frequency" of a qubit is the frequency which induces the $|0\rangle \to |1\rangle$ excitation.

characteristic frequency. If the qubit circuit is stimulated with a control pulse at its characteristic frequency transmitted separately to the qubit via a *drive line*, then it will excite the qubit to some superposition of $|0\rangle$ and $|1\rangle$. While excited, the characteristic frequency of the qubit/resonator pair will be shifted. If a control pulse is then sent on the readout line at the characteristic frequency of the qubit/resonator pair, then the absorption of the pulse will be different from if the qubit was in the ground state. If this measurement procedure is repeated and averaged, then the qubit state can be statistically estimated. This procedure is called *dispersive readout* [35, p. 35]. A consequence of dispersive readout is the destruction of the qubit state; when we send the second control pulse on the readout line, then that pulse will influence the oscillation of the qubit/resonator pair, which causes the qubit to collapse to either $|0\rangle$ or $|1\rangle$. The next two sections cover how to send control pulses at specific frequencies and what the word "absorption" means in this context and how it can be measured as a signal.

1.2 Modulation

In order to make a qubit resonate, we must send a pulse which has the same frequency as the $|0\rangle \rightarrow |1\rangle$ excitation frequency. This is quite easy if we do not care about the shape of our pulse, since we can simply send a sinusoidal signal which oscillates at the characteristic frequency. However, if we want to send a pulse with a specific shape, then situation calls for a more sophisticated method. Let $m(t)$ be a signal, which we call the *modulation* or *baseband* signal - this is the signal with our desired shape. The modulation signal contains some information, like a message. The process of modulating a *carrier* signal ϕ with a baseband signal is called *modulation*. The idea is that the carrier signal will transmit our arbitrary signal shape, but at a higher frequency. For example, we can say the same word using a low pitched voice and a high pitched voice and still have our word be understood by another person. In this example, m is the word that we want to say and ϕ is the sound wave produced by our vocal chords travelling through the air coming out of our lungs. The process of receiving a modulated carrier and obtaining the informative baseband signal from it

is known as *demodulation*. In the voice example, demodulation would be a listener close by perceiving our voice with their eardrums and noting what we say.

In electronics, the medium is simply a cable instead of air, like a coaxial cable, and the modulation signal m is a voltage signal, which encodes some information, like the voltage across a resistor. The signal m can also be a current signal but in this thesis we use "signal" and "voltage signal" synonymously. The carrier signal ϕ is simply another signal which is periodic with some frequency ω_c. Of course, periodicity of ϕ is a simplification and is not strictly required. Although, usually ϕ is chosen to be sinusoidal by virtue of the simplifications which arise from the trigonometric identities.

There are many ways to modulate ϕ with m, but in general there are only two modulation techniques which can be used to modulate a sinusoidal carrier. The two techniques are known as amplitude modulation (AM) [6, p. 282] and exponential modulation (EM) [6, p. 289]. The former modulates the amplitude of ϕ as a function of m and the latter modulates the argument of ϕ as a function of m. In both cases we assume a sinusoidal carrier, say e.g. $\phi = \cos$, with constant frequency ω_c. Firstly, amplitude modulation is given by

$$\varphi_{AM}(t) = (A + m(t)) \cos(\omega_c t) \tag{1.1}$$

This type of modulation is also known as double sideband carrier (DSB) modulation [6, p. 279]. Here $A \in [0, \infty)$ is simply a way to transmit the carrier along with the modulating signal so that the receiver can demodulate the carrier and receive the informative baseband signal without generating their own carrier. When $A = 0$ we write DSB-SC to mean DSB with a suppressed carrier. It is a double sideband modulation because of the Fourier transform pair

$$m(t) \cos(\omega_c t) \overset{\mathcal{F}}{\longleftrightarrow} \frac{1}{2}[M(\omega - \omega_c) + M(\omega + \omega_c)] \tag{1.2}$$

which duplicates the spectrum of $m(t)$ at $\pm\omega_c$ in the frequency domain. These two terms in the spectrum are known as the left and right sidebands [6, p. 279] (LSB & RSB) of the modulated carrier. Secondly, we have exponential modulation, which is given by

$$\varphi_{EM}(t) = \cos(\omega_c t + \psi(t)) \tag{1.3}$$

where $\psi(t)$ is a measure of $m(t)$ that is obtained from an invertible linear operation on $m(t)$. A specific instance of EM is the well known frequency modulation (FM) which is when $\psi(t) = \int_{-\infty}^{t} m(\tau)d\tau$. In frequency modulation, the instantaneous frequency of the carrier wave is modulated and is proportional to the amplitude of $m(t)$. We need the operation to be invertible, because we need to be able to demodulate the carrier. In practice we either use AM or EM for any given modulation, not both on the same carrier. We will not cover modulation at further depth. The interested reader is referred to [6, Chapter 4] by B. P. Lathi. In this thesis we are mainly interested in AM. This is because only AM was used in the MC2 cryolab for the conduction of the experiments in Chapter 4, where AM will also be covered in more detail.

A domain specific detail of quantum computing is that the modulating signal $m(t)$ may be complex valued. In fact, all modulating signals that we will cover in this thesis will be complex valued as it is a tractable way to describe the control pulses which we send to the qubits. The concept of a complex valued signal brings us to the notion of *I/Q modulation*. By a complex signal, we mean a function of time where the value at each moment is a point in $\mathbb{C}$. Since we are specifically considering voltage signals we will naturally struggle to transmit complex voltages. In electronics, what is done instead is similar to the fact that there is a bijection from $\mathbb{C}$ to $\mathbb{R}^2$. Recall from linear algebra that the bijective mapping $a + ib \mapsto (a, b)$ can be accomplished by selecting a basis which spans $\mathbb{R}^2$ and then scaling the basis vectors appropriately. For the standard choice $e_0 = [1, 0]$, $e_1 = [0, 1]$ we simply compute $ae_0 + be_1$.

The same approach is used for complex signals. Instead of selecting two orthogonal vectors we select two orthogonal signals which together synthesize φ after amplitude modulation (instead of scaling). The standard basis choice is sin and cos with the same frequency ω_c. When these are amplitude modulated to match φ we call them the *in-phase* and *quadrature*[2] components of φ. Together they make an *I/Q pair*. If $\phi = \cos$ is chosen, then the amplitude modulation of the sin basis signal is the quadrature component of φ and vice versa for $\phi = \sin$. Although, note that this choice is completely arbitrary, since $\cos(t + \frac{\pi}{2}) = -\sin(t)$, $\forall t$. The terms in-phase

[2]Quadrature means offset by one-quarter cycle, or $\frac{\pi}{2}$ radians.

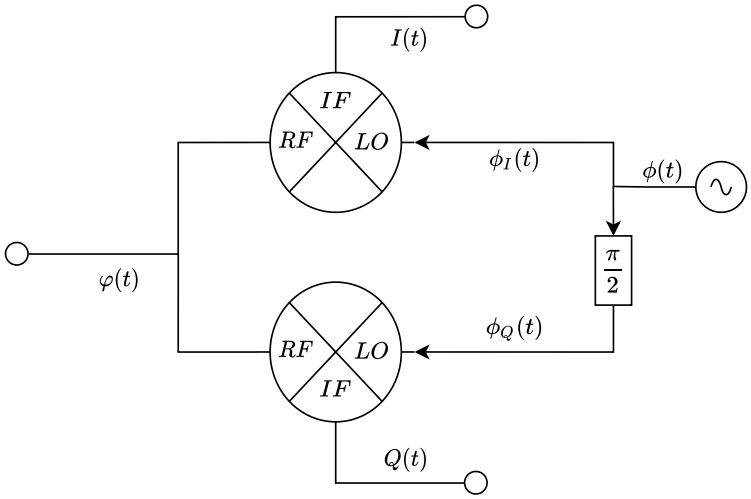


Figure 1.3: Schematic of an I/Q mixer. During modulation the ports of I and Q are input ports and the port of φ is an output port. During demodulation this schematic is run in reverse, that is, the ports of I and Q are output ports and the port of φ is an input port.

and quadrature are therefore essentially just a naming convention; it does not matter which one is which, but we have to choose one. We then transmit the real component of the complex signal as the I component and the imaginary as the Q component (or vice versa). An instrument which can synthesize complex signals from an I/Q pair is known as an I/Q mixer. For a schematic of an I/Q mixer, see Figure 1.3. During modulation the $\bigotimes$ symbol in Figure 1.3 is called a mixer and indicates a product of signals in the time domain, so when $\phi(t) = \cos(\omega_c t)$ we have

$$\begin{aligned} \varphi(t) &= I(t)\cos(\omega_c t) + Q(t)\cos(\omega_c t + \frac{\pi}{2}) \\ &= I(t)\cos(\omega_c t) - Q(t)\sin(\omega_c t) \end{aligned} \tag{1.4}$$

Say we have the incoming DSB-SC modulated signals

$$I(t) = m_I(t)\cos(\omega t + \theta_I) \tag{1.5}$$

$$Q(t) = m_Q(t)\sin(\omega t + \theta_Q) \tag{1.6}$$

then the signal at the output of the I/Q mixer will be

$$\begin{aligned} \varphi(t) = {} & m_I(t)\cos(\omega t + \theta_I)\cos(\omega_c t) \\ & - m_Q(t)\sin(\omega t + \theta_Q)\sin(\omega_c t) \end{aligned} \tag{1.7}$$

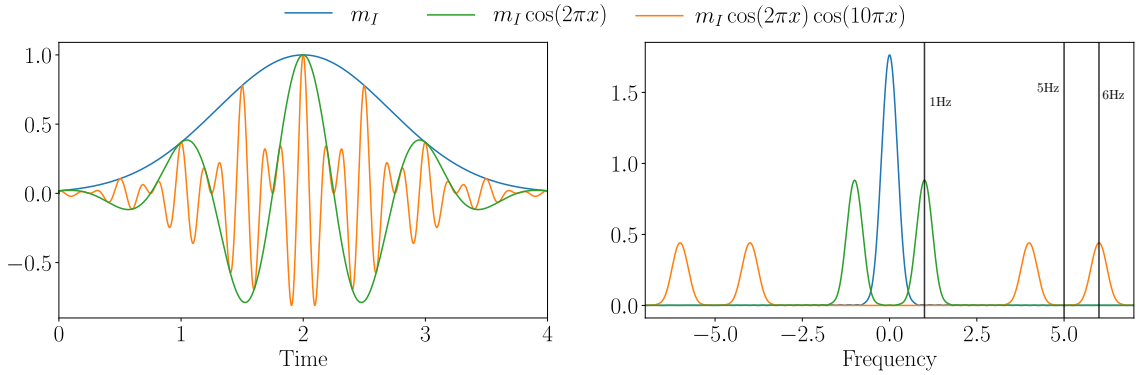


Figure 1.4: Example of the in-phase component of a twofold DSB-SC modulated signal.

if we apply the trigonometric product-to-sum identities, this simplifies to the following

$$\begin{aligned}\varphi(t) = \frac{m_I(t)}{2}\big[\cos((\omega-\omega_c)t+\theta_I)+\cos((\omega+\omega_c)t+\theta_I)\big] \\ -\frac{m_Q(t)}{2}\big[\cos((\omega-\omega_c)t+\theta_Q)-\cos((\omega+\omega_c)t+\theta_Q)\big]\end{aligned} \tag{1.8}$$

If $I(t)$ and $Q(t)$ are phase matched to be perfectly $\frac{\pi}{2}$ radians out of phase relative to each other, i.e. if $\theta_I = \theta_Q = \hat{\theta}$, then we can simplify even further to obtain

$$\begin{aligned}\varphi(t) = \frac{1}{2}\big[\cos((\omega-\omega_c)t+\hat{\theta})(m_I - m_Q)(t) \\ +\cos((\omega+\omega_c)t+\hat{\theta})(m_I + m_Q)(t)\big]\end{aligned} \tag{1.9}$$

We see that the spectrum of $(m_I \pm m_Q)(t)$ is placed at $\omega \pm \omega_c$ in the frequency domain, see e.g. Figure 1.4. A complex number is the sum of its real and imaginary components. Similarly, a complex signal is the sum of its in-phase and quadrature components, hence we are interested in the RSB in the spectrum of φ. By isolating the RSB with a filter we can therefore transmit our modulating signal $m_I + m_Q$ at a higher frequency $\omega + \omega_c$ which uniquely represents the complex signal $m(t) = m_I(t) + i m_Q(t)$. I/Q mixers used in the reverse order (current at input and current at output are swapped) decompose angle modulated complex signals into an I/Q pair. When used in this fashion we say that the I/Q mixer *demodulates* the carrier and produces the I/Q pair. We do not cover in depth how demodulation is performed and refer instead the reader to [22, Appendix C].

In the above analysis we assumed that $\theta_I = \theta_Q$. In practice, this is never the case due to small differences in circuitry between the mixers $\bigotimes_I$ and $\bigotimes_Q$. When this is the case, we say that the mixers $\bigotimes_I$ and $\bigotimes_Q$ are imbalanced. Specifically we say that (i) *phase error* is a source of imbalance. Other sources of imbalance are (ii) the *amplitude ratio of the mixers* as well as (iii & iv) *DC offsets* of the $I(t)$ and $Q(t)$ signals. Therefore, whenever one uses I/Q mixers to transmit complex signals one must always first calibrate these four values.

1.3 Measurement

We begin this section with showing what kind of device is being used, namely a quantum chip, illustrated in Figure 1.5.

Not all qubit chips have to look like Figure 1.5, this is just an example design. The main control interface of a qubit chip are through its *ports*. A port is simply a physical location on the chip where microwave pulses can enter or leave. Internally, each port is connected to a *line*. The line leads incoming pulses to the correct component and outgoing pulses to the correct port. In the schematic in Figure 1.5, only the port labeled b is an output port, namely the output port of the readout line.

In a quantum computer with superconducting qubits there are usually at least three different kinds of ports for every qubit chip: drive ports which are connected to drive lines, coupling ports connected to coupling lines, and readout ports connected to readout lines. These three kinds are usually separated because it is desirable to have a way to operate on qubits, entangle qubits, and measure qubits separately. It it is not necessary that each qubit has their own drive line or coupling line like in Figure 1.5, they can also be shared like the readout line.

We measure a qubit state by looking for the absorption of the readout stimulus by the readout resonator at its characteristic frequency, as explained in Section 1.1. We will call such stimuli *readout pulses*. In Figure 1.5, ports a and b indirectly corresponds to two cables coming out of the cryostat which we use to interface with the quantum

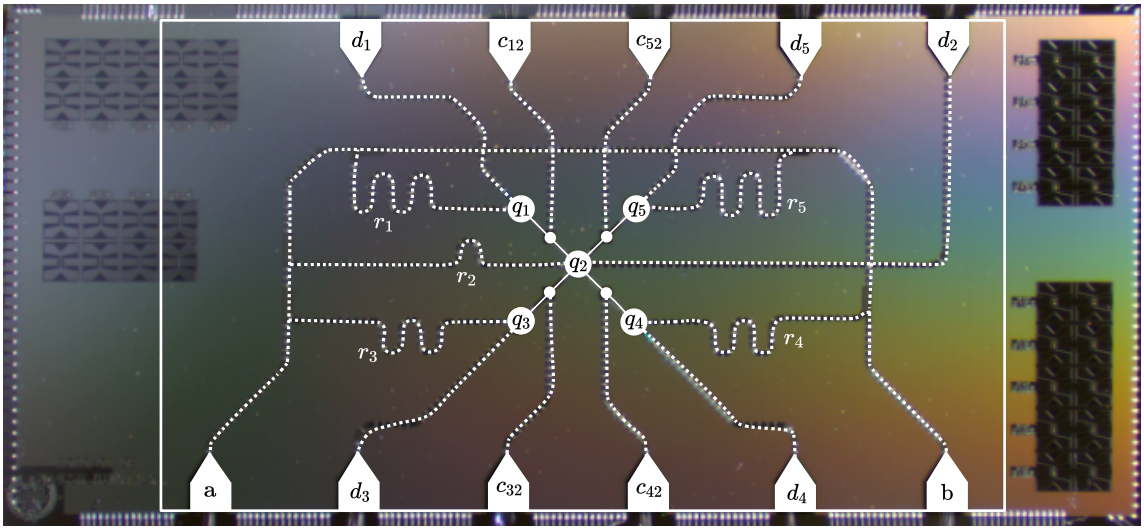


Figure 1.5: Butterfly design five qubit quantum chip with a superposed schematic. The five large circles labeled with q are the qubits. The readout resonators are the meander shaped lines labeled with r. The small circles are the couplers, so in this design, the coupling map is $\{\{q_1, q_2\}, \{q_3, q_2\}, \{q_4, q_2\}, \{q_5, q_2\}\}$. The remaining labels are labels of the chip's ports. This chip has four drive lines, one readout line, which is shared by all qubits, and four coupling lines. Device designed and measured by Christopher Warren. Photo courtesy of Christopher Warren.

computer. This is only an indirect correspondence, because in reality there are many components in-between the chip and the final output ports of the cryostat. Let $a_1(t)$ denote the readout pulse that we are sending to port a of the chip and $b_2(t)$ denote the pulse that exits the b port of the chip and subsequently propagates up out of the cryostat. From outside the cryostat, the interface looks like Figure 1.6.

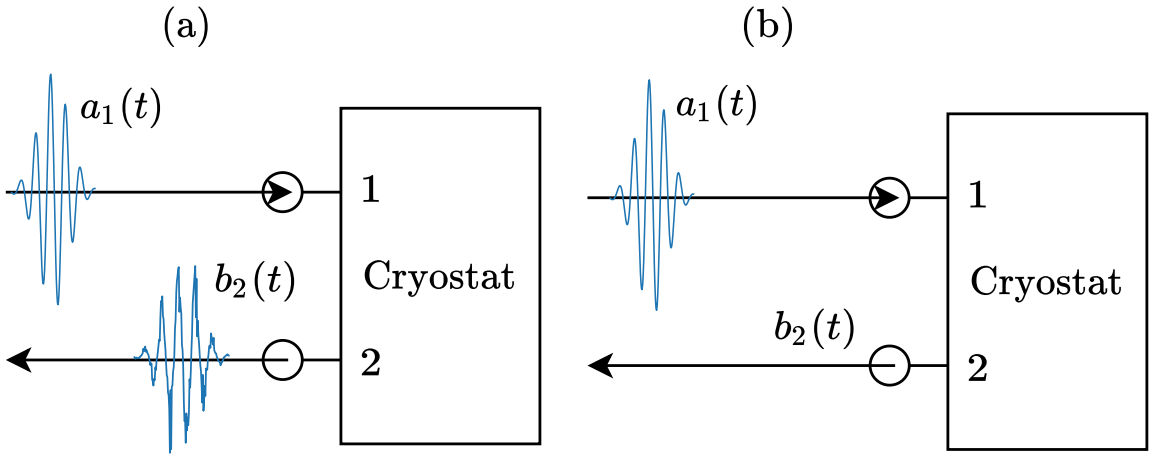


Figure 1.6: (a) A readout pulse is reflected out of the opposite port. There will of course be a slight delay between the readout pulse being transmitted and it being reflected out the other port, since it has to travel through the cryostat. This duration is known as the time of flight, and it can be experimentally measured for a given setup. (b) An equivalent readout pulse is later absorbed inside the cryostat, indicating a stateful change within the cryostat.

Measuring a qubit amounts to measuring $b_2(t)$. A single measurement of $b_2(t)$ is called a *shot*. Shots can be in the form of *traces* which is when all of $b_2(t)$ is measured and stored as data. Alternatively, shots can also be integrated to reduce the measurement to a singe complex number $\int b_2(t)dt$. The latter approach is often used in practice since it is a more compressed form of measurement - however, the actual shape of the trace is of course lost after integration. The qubit state can be determined from the trace by comparing it to the readout pulse. Absorption (Figure 1.6 (b)) of the pulse indicates that the characteristic frequency of the coupled system was not changed, and hence we can infer that the qubit was measured to be in the $|0\rangle$ state. A non-absorption (Figure 1.6 (a)) means that the characteristic frequency of coupled system was changed, and hence we can infer that the qubit was measured to be in an excited state. However, the trace is a stochastic process, so it is necessary to repeat the same measurement many times in order to get a high resolution image of the distribution of the state. To determine exactly which excited state the qubit was measured to be in, one can use a vector quantization method, such as k-means

clustering. Qubit state discrimination is covered in more detail in Section 4.5.

Furthermore, the qubit system must be ensured to be in the ground state in-between shots in order to maintain the assumption of statistical independence. Consequently, without technology to actively reset the qubit to the ground state, we have to wait for the qubit to relax on its own between each shot, which also increases the duration of the sampling procedure. These are the two primary reasons which constrain the minimum duration of measurement shots, both of which are being actively researched [9, 19, 29, 31]. Measurement and choice of readout pulse is discussed further in Section 4.1 and Section 4.2. For the development of the theory it suffices to know that a qubit measurement is essentially a complex number yielded from the integration of the trace in response to an I/Q modulated readout pulse over some fixed time window.

1.4 Pulse Programming

The aim of this section is provide some background about how quantum computers using superconducting qubits are programmed with microwave control pulses, without delving into how they are constructed. It is meant to explain the whole process in broad strokes.

The quantum circuit model of computation is a model wherein qubits are operated upon using quantum logic gates. Qubits are represented with horizontal lines and logic gates are represented as labeled boxes which the lines enter and exit, depicted in Figure 1.7 (a).

Algebraically, quantum logic gates are unitary matrices which serve as building blocks for quantum circuits, just like classical logic gates are building blocks of classical digital circuits. Many quantum gates are describable in terms of other quantum gates through logical equivalences. For example, the two circuits in Figure 1.7 (a), (b) are equivalent. To perform the computation described by a quantum circuit, we implement it with a schedule of microwave control pulses, visualized in Figure 1.7

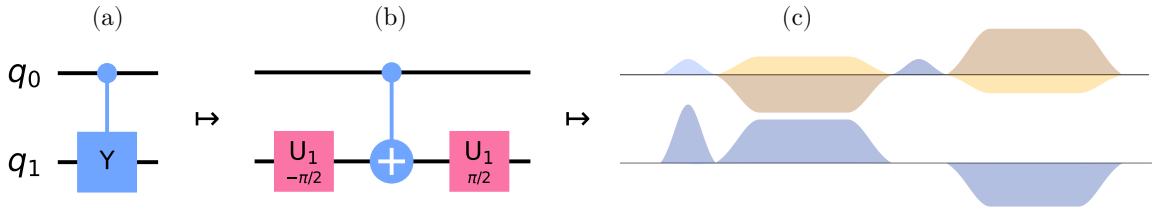


Figure 1.7: (a) A quantum circuit consisting solely of a controlled Y gate. (b) The circuit in (a) can be described with other gates. (c) The circuit in (b) is implemented with a calibrated pulse schedule where time dynamics are now specified. This schedule can be used to control the qubit.

(c). This schedule indicates how the emission of control fields over the qubits should be orchestrated. For simple gates, the schedule can sometimes be a single pulse. For more advanced gates, then sometimes very many pulses are needed. Moreover, the pulses need to be *calibrated* to a specific qubit, since each qubit is unique.

Pulse schedules can very easily be prepared manually with any vectorized math library such as e.g. `NumPy` [40]. However, the main virtue for using the gate-based model of quantum computation is that it makes it easy to make large, useful circuits by only dealing with unitary matrices [17, p. 171] and letting users ignore hardware specific details, such as calibrations.

Pulse schedules for certain gates are easy to calibrate, these are called *native* gates. This is because the quantum system is suitable for the operation that they define. For example, as we remarked in Section 1.1, for superconducting qubits the qubit state is constantly precessing around the z-axis in the Bloch sphere. Consequently, it is really easy to implement rotations around the z-axis on a superconducting qubit. All we need to do is alter the phase of all control pulses sent to that qubit after the time of rotation. Then, in a rotating frame [35, p. 11], (i.e. when the xy-axes are precessing around the z-axis but the state is still), the qubit state will have rotated around the z-axis. So a z-rotation on superconducting qubits is actually implemented with zero pulses.

If a set of gates can decompose *any* quantum logic gate then we call it *universal* [33, p. 10]. In the language of group theory, if a quantum computer has N qubits, then

a gate set is universal if it generates [13, p. 77] $U(2^N)$ under matrix multiplication. An example of a universal gate set for any quantum computer with a finite number of qubits is $\{\mathrm{H}, \mathrm{CX}, \mathrm{T}\}$ [33, p. 11], i.e. $U(2^N) = \langle\{\mathrm{H}, \mathrm{CX}, \mathrm{T}\}\rangle$. In classical logic this is analogous to how the `NAND` gate can be used to construct any classical circuit [25, p. 69] (although they might be very deep circuits). We will call the gate set for which a quantum computer has defined calibrated pulse schedules for the quantum computer's *basis set*. Calibrations change with time, so it is easiest to have a small basis set, with as many native gates as possible. It is of course preferable if the basis set is also universal.

The process of (a) designing a quantum circuit, (b) decomposing it against the basis set of some quantum computer, and (c) mapping the decomposed circuit to calibrated pulses is the compile chain of quantum computation with the gate-based model. Step (b) is sometimes called *unrolling* the gates of the circuit. We can of course unroll gates manually, but it is much more time efficient to let a classical computer do that. In order to do describe a quantum circuit to a computer we need a textual representation which is amenable to processing by computers. There are many textual intermediate representations of quantum circuits available, but one of the most common open source alternatives is IBM's OpenQASM [32, 38]. IBM has also developed a software development kit called Qiskit [43], which is a Python implementation of OpenQASM. Qiskit allows us to programmatically unroll quantum circuits described with OpenQASM through a process called *transpilation*[3]. Transpilers usually implement gate unrolling with multiple passes through known logical circuit equivalences (as expressed in OpenQASM) until convergence is reached within the target basis set. Transpilation is deemed out of scope for this thesis, since this thesis focuses exclusively on the pulse level (c). Recently, IBM also developed a submodule to Qiskit called OpenPulse. OpenPulse adds to the Qiskit compilation pipeline the capability to map a circuit to a pulse schedule as an intermediate representation, given that a target basis set is specified by some provider with a quantum computer [38, p. 2].

The main virtue of using an intermediate representation such as open OpenPulse is

[3]Transpilation is not the same as gate unrolling. Transpilation works with the textual representation of the circuit whereas gate unrolling is done in terms of quantum gates (unitary matrices). However, there is a very close correspondence.

that it is able to describe time dynamics of quantum computation that are unspecified in the circuit model. There is a similarity between pulse schedules and classical CPU timing diagrams [25, p. 149], however, CPU timing diagrams are outputs from a processor whereas pulse schedules are inputs to a quantum processor. Moreover, in a pulse schedule the signal level can be any complex number. The abstraction layer that OpenPulse defines is very simple, and stems from the introduction of a concept called a *channel*. The concept is comparable to a communications channel which is not a novel concept and its use can at least be traced back to C. Shannon (1948) [1, p. 381]. OpenPulse defines four kinds of channels, they are called *acquire channels*, *drive channels*, *coupling channels*, and *measure channels*. See Figure 1.8 for example.

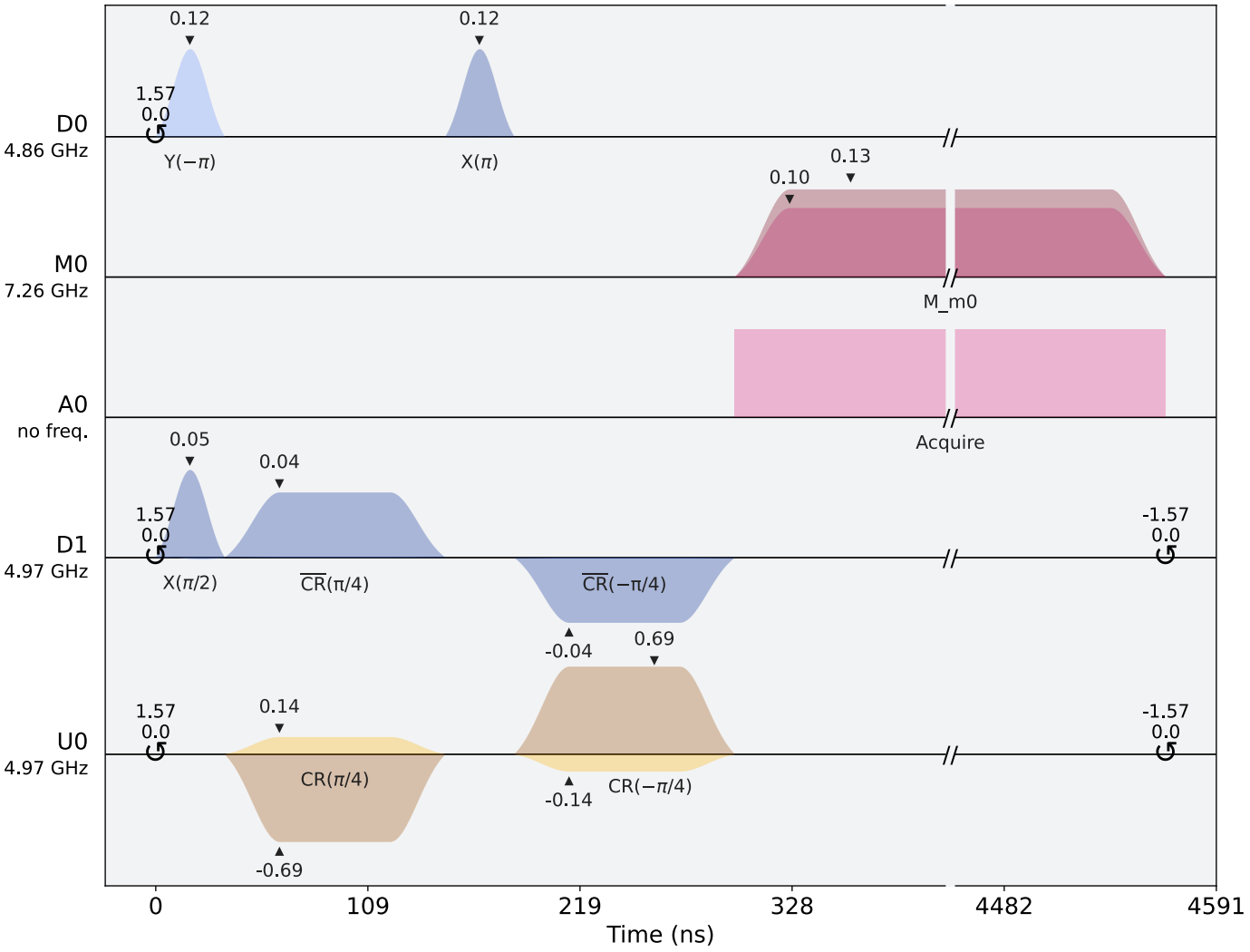


Figure 1.8: The OpenPulse schedule yielded from the OpenQASM textual representation of the quantum circuit in Figure 1.7 (b) with an explicit measurement at the end. The ↻ symbol denotes a change in the reference frame, such as a z-rotation. This schedule shows all four kinds of OpenPulse channels

- *acquire* channels, usually labelled with `A`.
- *drive* channels, usually labelled with `D`.
- *coupling* channels, usually labelled with `U`.
- *measure* channels, usually labelled with `M`.

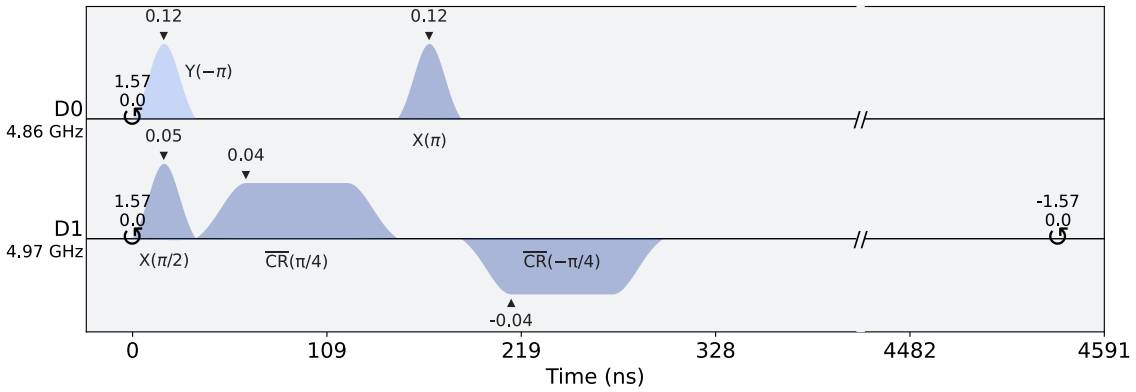


Figure 1.9: The drive channels of the qubits in the pulse schedule in Figure 1.8.

The naming of these three latter kinds suggests a correspondence to drive lines, coupling lines and measure lines. However, a channel is a completely virtual concept; it does not have to correspond to a physical object. A channel is essentially just a naming convention for subsets of frequencies. Consider e.g. the drive channels in Figure 1.9. The channels `D0` and `D1` have been assigned the frequencies 4.86 GHz and 4.97 GHz indicating the local oscillator frequencies of two carriers. The pulses displayed in Figure 1.9 are indicators for how a channel's carrier should be amplitude modulated at that moment in time, at the channel's assigned frequency. However, this description is completely independent of any hardware components. The hardware agnosticism that this induces is useful when conveying time dependent information which is actually virtual in nature, such as what integration window to use when integrating a trace, as discussed in the last section. In fact, this is exactly what the acquire channel in OpenPulse is designed to do, displayed in Figure 1.10. Indeed, note that the acquire channel `A0` does not have an assigned frequency.

The second facet of Qiskit OpenPulse is a sizeable collection of parametric template schedules which can be used for standard quantum computer experiments. Importantly, these schedules do not have a gate-based analogues, because they rely on time dependent instructions. Some examples are dynamical decoupling [2] schedules, optimal control schedules [12], error mitigation schedules, and single qubit characterisation schedules [38].

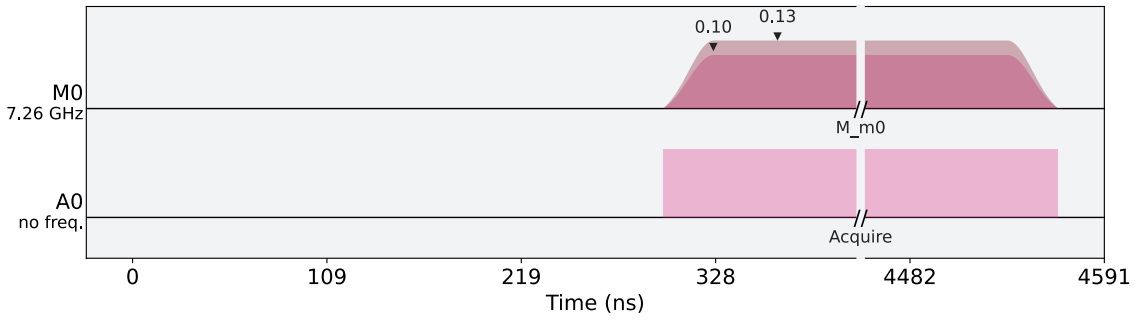


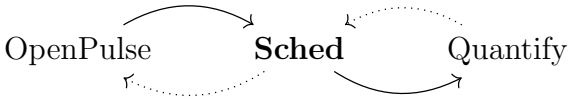
Figure 1.10: The measure channel and the acquire channel of Figure 1.8. The acquire channel is not assigned a frequency, because it simply dictates when to integrate. The two overlapping pulses in the measure channel indicate real and imaginary components of the modulating signal for a carrier transmitting a readout pulse.

In a certain sense, a pulse schedule is a bit like sheet music; its information needs to be programmatically assigned to instruments before it has meaning as a quantum computer experiment. To accomplish this, we use instrument orchestration software. There exists multiple commercial and free software packages for interfacing with quantum hardware, such as QCoDeS, pyCQED, qKIT, Labber, and Quantify [35, 37, 48, 49]. With the use of the Qiskit transpiler, any quantum circuit described with OpenQASM is also executable on Chalmers' quantum computers, using the Python library presented in Chapter 3 of this thesis. Although, a caveat for this is that one needs to maintain calibrations for a universal gate set, which was deemed out of scope for this thesis. This caveat is discussed more in the final chapter.

2

Theory

We saw in Chapter 1 that a quantum computer is programmed with control pulses. The goal of this chapter is to be able to describe all pulse schedules in a unifying categorial framework. Using this category, which we call **Sched**, we will have a solid ground for translation between OpenPulse and Quantify schedules, because in such a mathematical framework they are semantically the same thing.



We now give a brief roadmap of this Chapter which will lead us to our goal. As a begininning, we define what we mean by a pulse. Subsequently, we show that the set of all pulses is a vector space and that all carriers amplitude modulated by pulses are in fact pulses as well. Using Fourier analysis, we then build a set that we call the $\text{chan}_{\mathcal{A}}$ set of a pulse which determines the frequency domain occupancy of a pulse. The $\text{chan}_{\mathcal{A}}$ set is parameterised by a namespace $\mathcal{A}$ which has an interpretation as frequency intervals centered around the characteristic frequencies of qubits or resonators sitting on the same qubit chip.

Using order theory, we then introduce a preorder relation on the set of all pulses, based on the time and frequency occupancy of a pulse. This preorder relation allows us to generate graphs from arbitrary pulse schedules. Finally, we provide some structure preserving transformations of these graphs which allows one to tractably generate large executable schedules such as N-dimensional parametric sweep schedules (in the form of graphs). The practical use of these graphs is that they can serve as stepping stones in a translation process between OpenPulse schedules and Quantify schedules.

The main results of this chapter have been marked out in the left margin with a right triangle symbol ▷. This is to provide readers who are mainly interested in the results with the option to skip parts which they are not necessarily interested in. These results are proven, hopefully unambiguously, using set-builder notation or direct argument. Longer arguments are marked out with "*Proof.*". When these were deemed complete, they were marked out with a filled in square symbol ■ on the right hand side of the page. Results obtained from counter examples, are marked with the symbol ⋫ in the left margin.

2.1 Defining a Pulse

Definition 2.1. (Pulse) A pulse is a function $f : \mathbb{R} \to \mathbb{C}$ such that

$$0 \leq \inf_{\overline{\mathbb{R}}} \operatorname{supp} |f| \leq \infty \text{ and } \int |f(x)|dx < \infty \tag{2.1}$$

Definition 2.2. ($\boldsymbol{p}$) The set $\boldsymbol{p}$ is defined as the set of all pulses.

Note that not all functions that are integrable are in $\boldsymbol{p}$. For instance, $\int |e^{-x^2}|dx = \sqrt{\pi}$, but $e^{-x^2} \notin \boldsymbol{p}$ because $\inf_{\overline{\mathbb{R}}} \operatorname{supp} |e^{-x^2}| = -\infty$.

▷ The set $\boldsymbol{p}$ is a function space (vector space where the elements are functions) with scalar multiplication and addition defined as

$$(\alpha f)(x) = \alpha f(x) \tag{2.2}$$
$$(f + g)(x) = f(x) + g(x) \tag{2.3}$$

for f, $g \in \boldsymbol{p}$ and $\alpha \in \mathbb{C}$.

Proof. We first show that $\boldsymbol{p}$ is closed with respect to (2.2). Take $f \in \boldsymbol{p}$, $\alpha \in \mathbb{C}$. If

$\alpha = 0$, then $\inf_{\overline{\mathbb{R}}} \text{supp } |\alpha f| = \inf_{\overline{\mathbb{R}}} \text{supp } \mathbf{0} = \infty$. If $\alpha \neq 0$, then

$$\begin{aligned} \text{supp } |\alpha f| &= \{x : |\alpha f(x)| > 0\} \\ &= \{x : |\alpha||f(x)| > 0\} \\ &= \{x : |f(x)| > 0/|\alpha|\} \\ &= \{x : |f(x)| \neq 0\} \\ &= \text{supp } |f| \end{aligned} \tag{2.4}$$

Furthermore

$$\int |\alpha f| dx = \int |\alpha||f| dx = |\alpha| \int |f| dx \tag{2.5}$$

$$|\alpha| \underbrace{\int |f| dx}_{< \infty} \implies \int |\alpha f| dx < \infty \tag{2.6}$$

Hence $f \in \boldsymbol{p} \implies \alpha f \in \boldsymbol{p}$, $\forall \alpha \in \mathbb{C}$. Next we show that $\boldsymbol{p}$ is closed with respect to (2.3). Take $f, g \in \boldsymbol{p}$. By the triangle inequality we have

$$|f(x) + g(x)| \leq |f(x)| + |g(x)|, \forall x \in \mathbb{R} \tag{2.7}$$

$$\begin{aligned} \text{supp } |f(x) + g(x)| &= \{x : |f(x) + g(x)| > 0\} \\ &\subseteq \{x : |f(x)| + |g(x)| > 0\} \\ &= \text{supp } |f(x)| + |g(x)| \end{aligned} \tag{2.8}$$

$$h \in \boldsymbol{p} \implies |h(x)| = 0, \forall x \in (-\infty, 0) \tag{2.9}$$

Hence $0 \leq \inf_{\overline{\mathbb{R}}} \text{supp } |f(x)|+|g(x)| \leq \infty$. Since $\text{supp } |f(x)+g(x)| \subseteq \text{supp } |f(x)|+|g(x)|$ we must also have that $0 \leq \inf_{\overline{\mathbb{R}}} \text{supp } |f(x) + g(x)| \leq \infty$. Furthermore

$$\begin{aligned} \int |f(x) + g(x)| dx \leq \underbrace{\int |f(x)| dx}_{< \infty} + \underbrace{\int |g(x)| dx}_{< \infty} \\ \implies \int |f(x) + g(x)| dx < \infty \end{aligned} \tag{2.10}$$

Hence $f, g \in \boldsymbol{p} \implies f + g \in \boldsymbol{p}$. The remaining conditions of a vector space [8, p. 2] follow from the fact that $\mathbb{C}$ is a vector space [8, p. 3]. ■

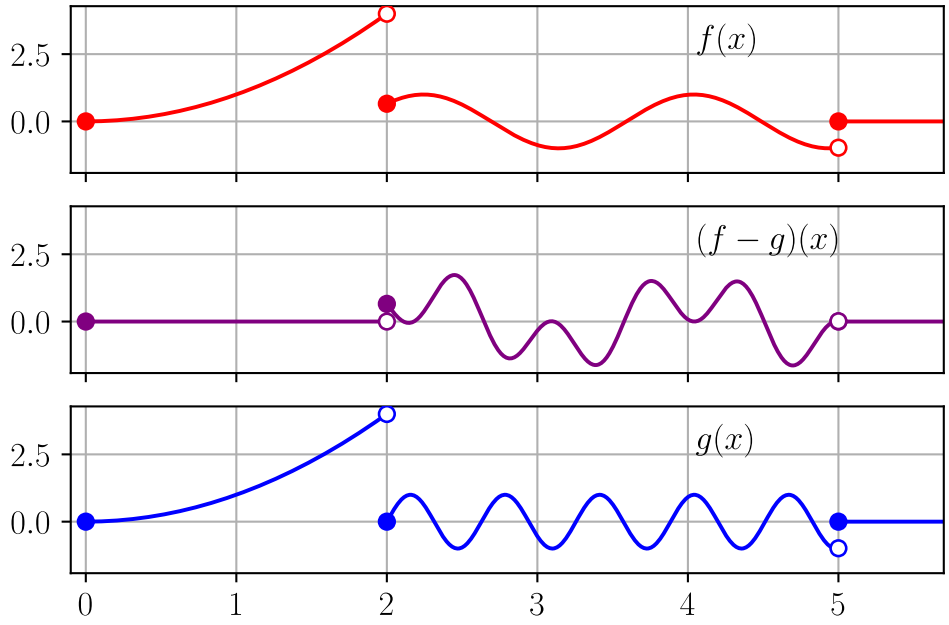


Figure 2.1: Here $\inf_{\overline{\mathbb{R}}} \operatorname{supp} |f| = \inf_{\overline{\mathbb{R}}} \operatorname{supp} |g| = 0$ but $\inf_{\overline{\mathbb{R}}} \operatorname{supp} |f-g| = 2$. We also have $\overline{\operatorname{supp}}\,|f| = \overline{\operatorname{supp}}\,|g| = (0,5)$ and $\overline{\operatorname{supp}}\,|f-g| = [2,5)$.

⊮ Note that $\operatorname{supp}|f(x)+g(x)| \neq \operatorname{supp}|f(x)| + |g(x)|$ in general. For example, see Figure 2.1. Given a pulse $f \in \boldsymbol{\mathcal{p}}$, we will call the quantity $\inf_{\overline{\mathbb{R}}} \operatorname{supp} |f|$ the *start time* of the pulse f. We will also use the notation $\overline{\operatorname{supp}}\,|f|$ to mean the convex hull of the support of $|f|$. Since all pulses are defined on $\mathbb{R}$, the convex hull of the support of $|f|$ is the smallest interval containing the support of $|f|$. The notation is necessary to distinguish from $\operatorname{supp}|f|$ because $\overline{\operatorname{supp}}\,|f| \neq \operatorname{supp}|f|$ in general. Take for example the pulse $g(x)$ in Figure 2.1, it has $g(2) = 0$, so $2 \notin \operatorname{supp}|g|$, but $2 \in \overline{\operatorname{supp}}\,|g|$. The pulse $\mathbf{0}$ never starts, because it is always zero. It has $\overline{\operatorname{supp}}\,\mathbf{0} = \emptyset$.

In quantum computer experiments the domain of our pulses $\mathbb{R}$ is interpretable as time. The dual of the time domain is the frequency domain, which we have not mentioned in this chapter so far. With this interpretation, the set $\overline{\operatorname{supp}}\,|f|$, $f \in \boldsymbol{\mathcal{p}}$ is the "space" that f occupies in the time domain. The dual notion would then be the "space" that f occupies in the frequency domain. However, such a dual set is less intuitive to develop and requires Fourier analysis. We begin by defining some

concepts which help us work in the frequency domain.

Recall first the concept of amplitude modulation, which was introduced in Section 1.2. Let the carrier be $\phi(x) = e^{i\omega_c x}$. If ϕ is amplitude modulated by a pulse f, then the amplitude modulated carrier is also a pulse. This follows from Euler's formula. Indeed, the amplitude modulated carrier and the original pulse have the same modulus

$$|f(x)\phi(x)| = |f(x)||e^{i\omega_c x}| = |f(x)|, \forall x \in \mathbb{R} \tag{2.11}$$

Because we can write $\cos(\omega_c x) = \frac{1}{2}[e^{-i\omega_c x} + e^{i\omega_c x}]$, this also applies for the standard choice when using the cosine function as a carrier.

▷ In other words, $f(x) \in \boldsymbol{p} \implies f(x)\cos(\omega_c x) \in \boldsymbol{p}$, $\forall \omega_c \in \mathbb{R}$.

Proof. Indeed, by the triangle inequality, we have

$$\begin{aligned} |f(x)\cos(\omega_c x)| &= |\frac{f(x)}{2}[e^{-i\omega_c x} + e^{i\omega_c x}]| \\ &\leq \frac{|f(x)|}{2}(\underbrace{|e^{-i\omega_c x}|}_{=1} + \underbrace{|e^{i\omega_c x}|}_{=1}) = |f(x)| \end{aligned} \tag{2.12}$$

Hence $0 \leq \inf_{\mathbb{R}} \operatorname{supp} |f(x)\cos(\omega_c x)| \leq \infty$ which we get from the containment

$$\operatorname{supp} |f(x)\cos(\omega_c x)| \subseteq \operatorname{supp} |f(x)| \tag{2.13}$$

Furthermore (2.12) also yields

$$\begin{aligned} \int |f(x)\cos(\omega_c x)|dx &\leq \underbrace{\int |f(x)|dx}_{<\infty} \\ \implies \int |f(x)\cos(\omega_c x)|dx &< \infty \end{aligned} \tag{2.14}$$

■

For the remaining part of the thesis we will no longer segregate carrier and pulse, because $\boldsymbol{p}$ clearly contains the set of all modulated carriers that we are interested in.

Furthermore, note that we always have the Fourier transform of $f \in \boldsymbol{p}$, by the integrability of pulses. In this thesis the definition $f(x) \overset{\mathcal{F}}{\longleftrightarrow} \int f(x)e^{-i\omega x}dx$ is used for the Fourier transform [6, p. 238], where the right hand side is a function of ω. When f is a slowly varying (with time) pulse, most of its spectral information will be located around $\omega \approx 0$ in the frequency domain. When f is a pulse modulated at some frequency $\omega \gg 0$, then we expect that the amplitude spectrum of f should have some large component around ω. For a general f we might have several large harmonic, or disharmonic components in the amplitude spectrum. For example, consider the function $f - g$ in Figure 2.1. This pulse is a composite pulse of two truncated sinusoids with different frequencies ω_f and ω_g. It can therefore be written as $(f-g)(x) = \cos(\omega_f x+\theta_f)-\cos(\omega_g x+\theta_g), \forall x \in \overline{\operatorname{supp}}\,|f-g|$. Its Fourier transform is approximately

$$(\hat{f} - \hat{g})(\omega) \approx \pi(\delta(\omega \pm \omega_f) - \delta(\omega \pm \omega_g)) \tag{2.15}$$

where $\delta(x)$ is the Dirac delta distribution. The corresponding amplitude spectrum is plotted in Figure 2.2. The Fourier transform in (2.15) is approximate because $(f - g)(t) = 0,\ \forall t \notin \overline{\operatorname{supp}}\,|f - g|$. In Figure 2.2 we see that there is a *peak* near the frequency of each sinusoidal component of the pulse. Additionally, there are extra peaks which are not at $\pm\omega_f$ or $\pm\omega_g$. These peaks are artefacts of the discontiuity of the pulse $f - g$. Consider for example a pulse with the $\operatorname{rect}(x)$ shape, where $\operatorname{rect}(x) \overset{\mathcal{F}}{\longleftrightarrow} \operatorname{sinc}(\omega/2)$ [6, p. 252], depicted in Figure 2.3. In fact, the amplitude spectrum in Figure 2.3 has infinitely many peaks as $\omega \to \pm\infty$. Pulses with discontinuities (such as f in Figure 2.3 or $f-g$ in Figure 2.1) can be described as piecewise functions. Piecewise functions can be described as sums of amplitude modulated rect functions. The Fourier transform of the rect function induces the sinc shape, and hence in general the amplitude spectrum of pulses with discontinuities will have infinitely many peaks. We will now define precisely what we mean by peak.

Definition 2.3. (Local maximum) Define the set

$$\begin{aligned} \operatorname*{locmax}_{x \in X} f &= \{x : \exists \epsilon > 0 : f(x) > f(\xi),\ \xi \in X \cap B_\epsilon(x)\} \\ &\text{where } B_\epsilon(x) = (x - \epsilon, x + \epsilon) \end{aligned} \tag{2.16}$$

where f is some real valued continuous function.

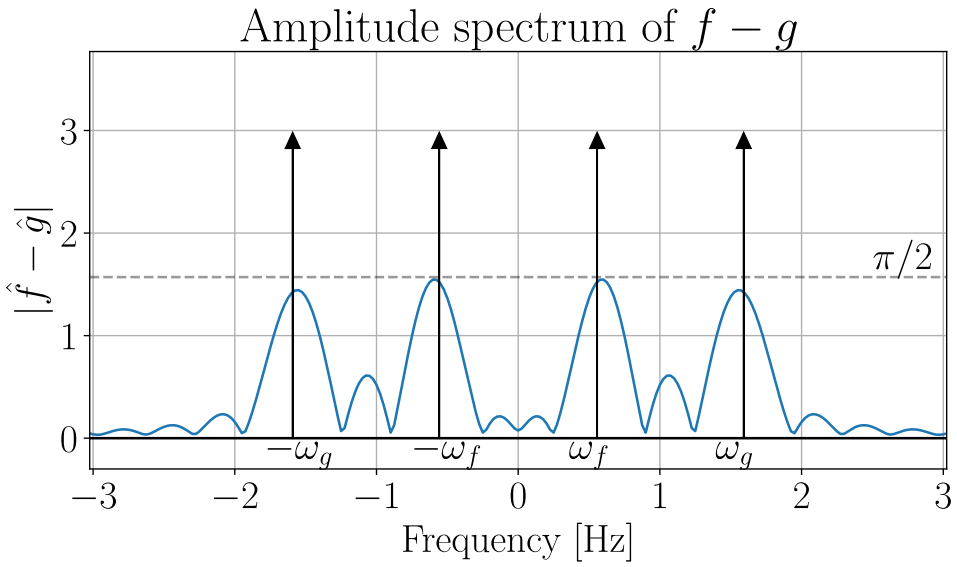


Figure 2.2: Amplitude spectrum (full, blue) of the pulse $(f - g)(x)$ in Figure 2.1. The modulus of theoretical Fourier transform of $\cos(\omega_f x + \theta_f) - \cos(\omega_g x + \theta_g)$ is plotted as black arrows, denoting Dirac δ distributions. The amplitude spectrum was computed with the Fast Fourier Transform (FFT).

The set $\underset{x \in X}{\operatorname{locmax}} f$ is called the set of strict local maxima of f over X. It is a construction borrowed from continuous optimization theory [39, p. 79].

⊮ Note that the set of local maxima is not defined for all pulses, because there exists pulses which have discontinuities, e.g. $f - g$ in Figure 2.1. Specifically, a peak is an element of the set $\underset{\omega \in \mathbb{R}}{\operatorname{locmax}} |\hat{f}(\omega)|$ where $f(x) \overset{\mathcal{F}}{\longleftrightarrow} \hat{f}(\omega)$, $f \in \boldsymbol{p}$. The set $\underset{\omega \in \mathbb{R}}{\operatorname{locmax}} |\hat{f}(\omega)|$ on the other hand is well defined in this context because the amplitude spectrum computed from the Fourier transform of any integrable function is real valued and uniformly continuous on $\mathbb{R}$.

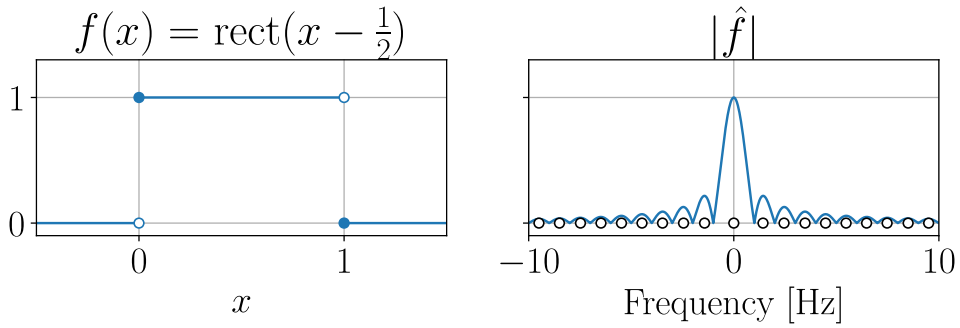


Figure 2.3: The locations of the peaks in the amplitude spectrum have been plotted with white markers.

▷ Some properties of locmax for any real valued continuous function $f : \mathbb{R} \to \mathbb{R}$ are

$$\text{(i)} \ \operatorname*{locmax}_{x \in X} \mathbf{0} = \emptyset \tag{2.17}$$

$$\text{(ii)} \ \operatorname*{locmax}_{x \in X} af = \operatorname*{locmax}_{x \in X} f, \ \ \forall a > 0 \tag{2.18}$$

$$\text{(iii)} \ f \geq 0 \implies \operatorname*{locmax}_{x \in X} f = \operatorname*{locmax}_{x \in \operatorname{supp} f} f \tag{2.19}$$

Proof. Property (i) holds because $\mathbf{0} \not> \mathbf{0}$ everywhere on X. Property (ii) holds because $af(x) > af(\xi), a > 0 \iff f(x) > f(\xi)$ for $>$ on $\mathbb{R}$. Property (iii) holds because if $f(x) \geq 0, \forall x$ then $\operatorname{supp} f = \{x : f(x) \neq 0\} = \{x : f(x) > 0\}$. Any zero z of f (i.e. $f(z) = 0$) cannot be a local maximum, because 0 is the least element in the image of $\mathbb{R}$ through f. Hence all local maxima of f must be in $\operatorname{supp} f$ when $f \geq 0$. ■

Property (iii) does not hold for $f \leq 0$ in general because some continuous functions ⊮ have local maxima which coincide with their zeros, like e.g. $-x^2$. The amplitude spectrum in Figure 2.2 has many peaks, but most of them are insignificant. This leads us to a definition of a *dominant* peak.

Definition 2.4. (Dominant peaks) Fix some $0 \leq \kappa < 1$. The set of dominant peaks of a pulse f is defined as

$$\operatorname{peaks}(f, \kappa) = \{\omega \in \operatorname*{locmax}_{\xi \in \mathbb{R}} |\hat{f}(\xi)| : |\hat{f}(\omega)| \geq \kappa \|\hat{f}\|_\infty\} \tag{2.20}$$

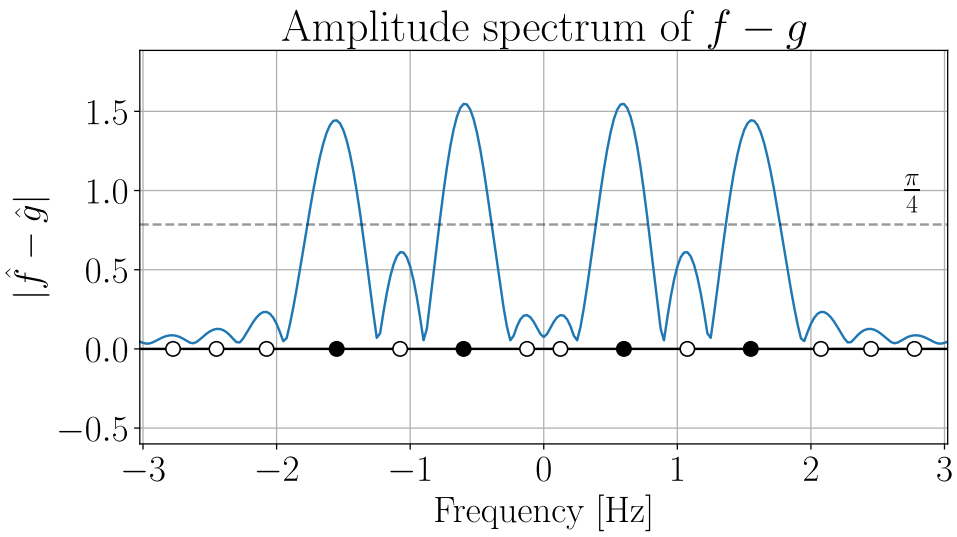


Figure 2.4: Amplitude spectrum (full, blue) of the pulse $(f - g)(x)$ in Figure 2.1. The non-dominant peaks have been plotted with white markers and the dominant peaks have been plotted with black markers. Here the κ threshold used was $\kappa = \frac{1}{2}$.

where $f \xleftrightarrow{\mathcal{F}} \hat{f}$ and $\|\hat{f}\|_\infty = \max_{\xi \in \mathbb{R}} |\hat{f}(\xi)|$ is the max norm.

By thresholding peaks we can partition the set of local maxima. For an example partition, see Figure 2.4. We want to define frequency occupancy of a pulse. The dominant peaks are what occupy the frequency domain. For example, if there were to be a small peak of some underlying component close to ω_f in Figure 2.4, then it would just merge together with the dominant peak at ω_f, making it appear more dominant. In other words, dominant peaks occupy more frequencies than just their peak frequency. We therefore partition the domain of $\hat{f}(\omega)$, i.e. $\mathbb{R}$, into equivalence classes which we call *channels*. We denote the partition by $\mathcal{A}$, and call it the *namespace*. For simplicity we assume every channel is an interval. For an example of an amplitude spectrum of a typical decaying pulse containing multiple sinusoidal components, with an associated namespace, see Figure 2.5. The intention here is to make dominant peaks occupy frequency channels. This leads us to the next definition.

Definition 2.5. ($\text{chan}_\mathcal{A}$) Let $\mathcal{A}$ be any interval partition of $\mathbb{R}$ and let $f \xleftrightarrow{\mathcal{F}} \hat{f}$ be a

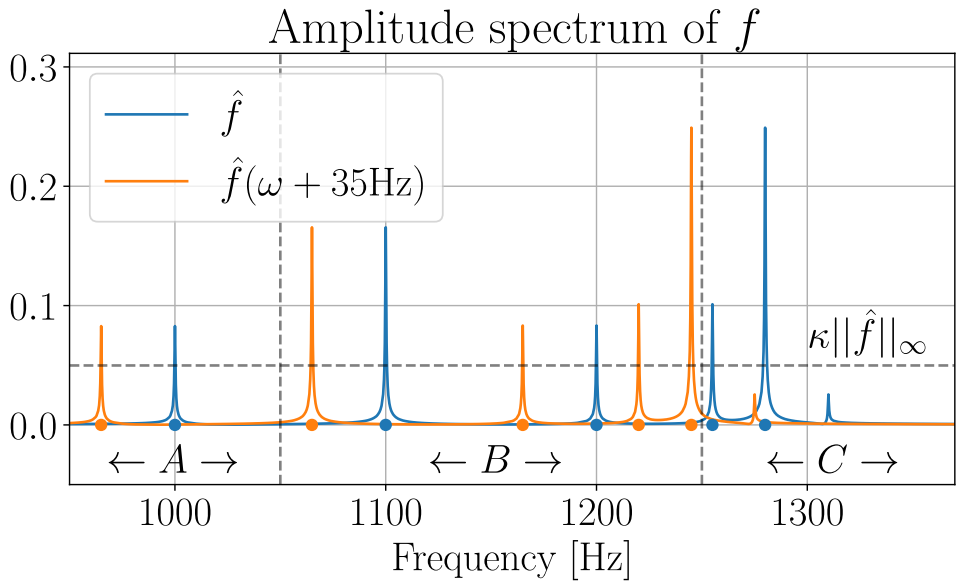


Figure 2.5: Here the namespace partition is $\mathcal{A} = \{A, B, C\}$ consisting of the channels $A = (-\infty, 1050\,\text{Hz})$, $B = [1050\,\text{Hz}, 1250\,\text{Hz})$, and $C = [1250\,\text{Hz}, \infty)$. The amplitude spectrum is plotted in blue and the amplitude spectrum shifted by $-35\,\text{Hz}$ is plotted in orange. The blue spectrum contains one peak in channel A, two peaks in channel B, and two peaks in C. The orange spectrum contains one peak in A and four peaks in B.

Fourier transform pair for some $f \in \boldsymbol{p}$. The set $\text{chan}_{\mathcal{A}} f$ is defined as

$$\text{chan}_{\mathcal{A}} f = \{[\omega]_{\mathcal{A}} : \omega \in \text{peaks}(f, \kappa)\} \tag{2.21}$$

For example, in Figure 2.5, then we have that $\text{chan}_{\mathcal{A}} f = \{A, B, C\}$. An important remark is that κ is an arbitrary threshold, but as $\kappa \to 0$ it will make the namespace meaningless since many pulses have peaks over all of $\mathbb{R}$. As $\kappa \to 1^-$ fewer channels will be included in $\text{chan}_{\mathcal{A}}$. Therefore, the threshold κ essentially decides a scale of how dominant a peak needs to be (relative to the other peaks in the spectrum) in order to occupy a channel in the namespace.

▷ We now list some properties of $\text{chan}_{\mathcal{A}}$ which holds $\forall f \in \boldsymbol{p},\ \forall \alpha \in \mathbb{C}\backslash\{0\}$.

$$\text{(i) } \text{chan}_{\mathcal{A}}(\mathbf{0}) = \emptyset \tag{2.22}$$

$$\text{(ii) } \text{chan}_{\mathcal{A}}(\alpha f) = \text{chan}_{\mathcal{A}} f \tag{2.23}$$

$$\text{(iii) } \text{chan}_{\mathcal{A}}(f(x - x_0)) = \text{chan}_{\mathcal{A}} f \tag{2.24}$$

$$\text{(iv) } \text{chan}_{\mathcal{A}}(f e^{i\omega_0 x}) = \{[\omega + \omega_0]_{\mathcal{A}} : \omega \in \text{peaks}(f, \kappa)\} \tag{2.25}$$

Proof. Property (i) holds because $\mathbf{0} \overset{\mathcal{F}}{\longleftrightarrow} \mathbf{0}$, $\underset{\xi \in \mathbb{R}}{\text{locmax}}\, \mathbf{0} = \emptyset$. Property (ii) holds by the linearity of the Fourier integral, $|\alpha \hat{f}| = |\alpha||\hat{f}|$ and moreover $\underset{x \in X}{\text{locmax}}\, |\alpha| f = \underset{x \in X}{\text{locmax}}\, f$ for $\alpha \in \mathbb{C}\backslash\{0\}$ and

$$|\alpha||\hat{f}(\omega)| \geq \kappa \|\alpha \hat{f}\|_\infty = \kappa |\alpha| \|\hat{f}\|_\infty \iff |\hat{f}(\omega)| \geq \kappa \|\hat{f}\|_\infty \tag{2.26}$$

Property (iii) follows from the fact that if $f \overset{\mathcal{F}}{\longleftrightarrow} \hat{f}$ then $f(x - x_0) \overset{\mathcal{F}}{\longleftrightarrow} \hat{f}(\omega) e^{-i\omega x_0}$ and $|e^{-i\omega x_0}| = 1$, $\forall \omega x_0 \in \mathbb{R}$. In other words (iii) states that time shifting of a pulse does not affect its channels. We can prove property (iv) directly; let $f \overset{\mathcal{F}}{\longleftrightarrow} \hat{f}$, then

$$f e^{i\omega_0 x} \overset{\mathcal{F}}{\longleftrightarrow} \hat{f}(\omega - \omega_0) \tag{2.27}$$

The spectrum $|\hat{f}(\xi - \omega_0)|$ is just $|\hat{f}(\xi)|$ shifted to the right by ω_0, so all peaks of $|\hat{f}(\xi)|$ have been shifted to the right by ω_0 as well. Hence, any dominant peak ω which was in channel $[\omega]$ before the shift is now in channel $[\omega + \omega_0]$. ■

For the example in Figure 2.5, the pulse corresponding to the orange spectrum does not have the same $\text{chan}_{\mathcal{A}}$ set as the pulse corresponding to the blue spectrum, since a frequency shift of $-35\,\text{Hz}$ has "moved" the dominant peaks that were in channel C over to channel B. This observation raises the question of what the maximal possible frequency shift amount of a pulse is, before it changes channel. We can find this quantity from the minimum distance from any peak to any channel boundary. Define the set

$$\text{bnd}(\mathcal{A}) = (\{\underset{\mathbb{R}}{\inf} A : A \in \mathcal{A}\} \cup \{\underset{\mathbb{R}}{\sup} B : B \in \mathcal{A}\})\backslash\{\pm\infty\} \tag{2.28}$$

which is simply the set of all finite boundaries in the namespace. For example in Figure 2.5 then $\text{bnd}(\mathcal{A}) = \{1050\,\text{Hz}, 1250\,\text{Hz}\}$. The unsigned maximum frequency shift of a pulse, denoted ω_{max} is then given by

$$\omega_{max}(\mathcal{A}, \kappa, f) = \inf_{\overline{\mathbb{R}}} \{|\omega - b| : \omega \in \text{peaks}(f, \kappa), b \in \text{bnd}(\mathcal{A})\} \tag{2.29}$$

The quantity $\omega_{max}(\mathcal{A}, \kappa, f)$ is the largest lower bound on the distance to any boundary. For the trivial partition $\mathcal{A} = \{\mathbb{R}\}$ we have that $\omega_{max}(\mathcal{A}, \kappa, f) = \inf_{\overline{\mathbb{R}}} \emptyset = \infty$, $\forall f$ which indicates that any amount of frequency shifting is allowed, because we can never leave the channel. For the example in Figure 2.5 we have $\omega_{max} = 5\,\text{Hz}$. Note that this does not tell us the direction to the closest boundary, as a signed maximum frequency shift would (no absolute value in (2.29)). However, since all channels considered in this thesis are approximately equal size large intervals, we settle with this simple description.

2.2 Ordering Pulses

With both time and frequency occupancy of a pulse well defined, we can now discuss how to order pulses in a such a way that that we can schedule them. To do that, we first need to be able to relate them. There are two important relationships to consider. The first relationship is whether the pulse f was transmitted before the pulse g. The second relationship is whether f and g were transmitted on the same channel.

Definition 2.6. (Simultaneity) Let $f, g \in \boldsymbol{p}$. The pulses f and g are said to be simultaneous i.f.f. $\overline{\text{supp}}\,|f| = \overline{\text{supp}}\,|g|$. When this is true we write $f \cong g$.

▷ Simultaneity is an equivalence relation on $\boldsymbol{p}$, which follows from the equality of sets. The equivalence classes are the set of pulses which are simultaneous, i.e. the pulses whose absolute values have support with the same convex hull. If $\overline{\text{supp}}\,|f| \cap \overline{\text{supp}}\,|g| \neq \emptyset$ but $f \ncong g$, then we say that f and g are *partially* simultaneous. For example, see Figure 2.6

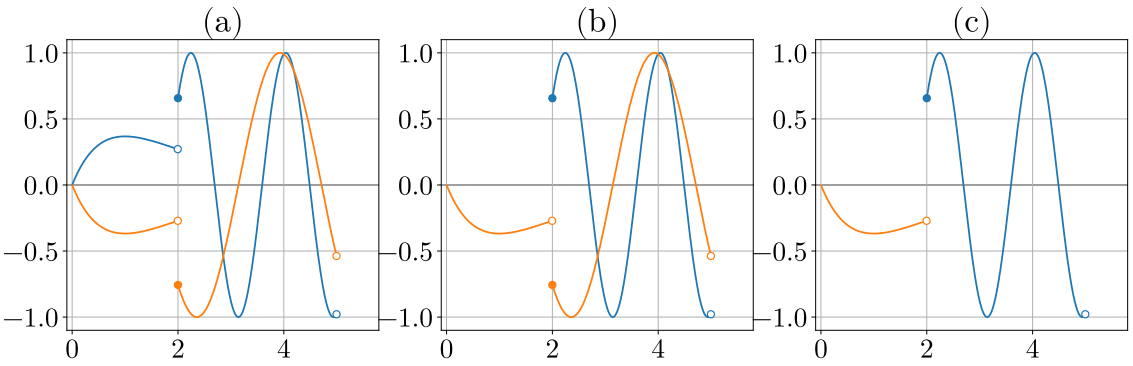


Figure 2.6: Examples of (a) Simultaneous pulses. (b) Partially simultaneous pulses. (c) Orange pulse precedes blue pulse, non-simultaneous pulses.

Definition 2.7. (Precedence) Let $f, g \in \boldsymbol{p}$. The pulse f is said to precede g i.f.f. $x < \inf_{\overline{\mathbb{R}}} \operatorname{supp} |g|$, $\forall x \in \overline{\operatorname{supp}}\,|f|$. If this is true we write $f \prec g$.

▷ Precedence is a strict partial order on $\boldsymbol{p}$.

Proof. Irreflexivity of $\prec$ follows from that $\inf_{\overline{\mathbb{R}}} \operatorname{supp} |g|$ is a lower bound on $\operatorname{supp} |g|$, hence if $f = g$, then by definition $x \in \overline{\operatorname{supp}}\,|f|$ cannot be strictly smaller. We now prove that $\prec$ is transitive. Let $f, g, h \in \boldsymbol{p}$. We have

$$f \prec g \prec h \implies \begin{cases} x < \inf_{\overline{\mathbb{R}}} \operatorname{supp} |g|, \ \forall x \in \overline{\operatorname{supp}}\,|f| \\ y < \inf_{\overline{\mathbb{R}}} \operatorname{supp} |h|, \ \forall y \in \overline{\operatorname{supp}}\,|g| \end{cases} \tag{2.30}$$

By definition of infimum, we have $\inf_{\overline{\mathbb{R}}} \operatorname{supp} |g| \leq y, \forall y \in \overline{\operatorname{supp}}\,|g|$. So, $\forall x \in \overline{\operatorname{supp}}\,|f|$, $\forall y \in \overline{\operatorname{supp}}\,|g|$, we then get that

$$\begin{aligned} x < \inf_{\overline{\mathbb{R}}} \operatorname{supp} |g| \leq y < \inf_{\overline{\mathbb{R}}} \operatorname{supp} |h| \\ \implies x < \inf_{\overline{\mathbb{R}}} \operatorname{supp} |h| \end{aligned} \tag{2.31}$$

All transitive irreflexive relations are necessarily asymmetric, so $\prec$ is a strict partial order on $\boldsymbol{p}$. ■

We now combine the last two relations into one.

Definition 2.8. ($\preceq$) Fix any interval partition $\mathcal{A}$ of $\mathbb{R}$ and some sensitivity threshold $0 \leq \kappa < 1$. Let $f, g \in \boldsymbol{p}$. We write that $f \preceq g$ i.f.f. the following conditions are true

$$\text{(i) } f \cong g \text{ or } f \prec g \tag{2.32}$$

$$\text{(ii) } \text{chan}_{\mathcal{A}} f = \text{chan}_{\mathcal{A}} g \tag{2.33}$$

▷ The relation $\preceq$ is a preorder on $\boldsymbol{p}$.

Proof. Let $f, g \in \boldsymbol{p}$. If $f = g$, then $f \cong g$ by reflexivity of $\cong$ and $\text{chan}_{\mathcal{A}} f = \text{chan}_{\mathcal{A}} g$ by set equality, so $\preceq$ is reflexive. Let $f, g, h \in \boldsymbol{p}$. We immediately have, from set equality

$$\begin{aligned} f \preceq g \preceq h \implies & \begin{cases} (f \prec g \text{ or } f \cong g) \text{ and } \text{chan}_{\mathcal{A}} f = \text{chan}_{\mathcal{A}} g \\ (g \prec h \text{ or } g \cong h) \text{ and } \text{chan}_{\mathcal{A}} g = \text{chan}_{\mathcal{A}} h \end{cases} \\ & \implies \text{chan}_{\mathcal{A}} f = \text{chan}_{\mathcal{A}} h \end{aligned} \tag{2.34}$$

Note that $f \prec g \implies f \ncong g$ and $f \cong g \implies f \not\prec g$, i.e. we cannot have that a pulse f precedes g while they are simultaneous. Consequently, the cases

$$\text{(i) } f \cong g \cong h \implies f \cong h \tag{2.35}$$

$$\text{(ii) } f \cong g \prec h \implies x < \inf_{\overline{\mathbb{R}}} \text{supp } |h|, \forall x \in \overline{\text{supp}}\, |g| = \overline{\text{supp}}\, |f| \tag{2.36}$$

$$\text{(iii) } f \prec g \cong h \implies x < \inf_{\overline{\mathbb{R}}} \text{supp } |g| = \inf_{\overline{\mathbb{R}}} \text{supp } |h|, \forall x \in \overline{\text{supp}}\, |f| \tag{2.37}$$

$$\text{(iv) } f \prec g \prec h \implies f \prec h \tag{2.38}$$

suffice to show that $\preceq$ is transitive. Case (i) and (iv) we already have from transitivity of $\cong$ and $\prec$ respectively. Case (ii) and (iii) both yield the definition of $f \prec h$. ■

⊮ Not all pulses relate under $\preceq$. For example, the two pulses in Figure 2.6 (b) are distinct, but do not relate; in general partially simultaneous pulses never relate under $\preceq$. A consequence of this is that $\preceq$ is not connected.

What we have now is a vector space $\boldsymbol{p}$ endowed with a preorder structure $\preceq$. The
▷ relation $\preceq$ additionally satisfies homogeneity.

Proof. Homogeneity of the relation means that $f \preceq g \implies \alpha f \preceq \alpha g, \forall \alpha \in \mathbb{C}$. Indeed if $\alpha \neq 0$, we have that $\operatorname{supp}|\alpha f| = \operatorname{supp}|\alpha||f| = \operatorname{supp}|f|$ and $\operatorname{chan}_{\mathcal{A}} \alpha f = \operatorname{chan}_{\mathcal{A}} f$. The case when $\alpha = 0$ is trivial because we already know $\mathbf{0} \preceq \mathbf{0}$. Hence $f \preceq g \implies \alpha f \preceq \alpha g$ by set equality. ■

⋫ However, $\preceq$ is not translation invariant and therefore $\preceq$ is not compatible with the function space structure [16, p. 140]. Indeed, $f \preceq g \not\implies f+h \preceq g+h, \forall f, g, h \in \boldsymbol{p}$. Take for example $f \neq \mathbf{0}$, $h = -f$, $g = 2f$. Clearly $f \preceq 2f$ since $\operatorname{supp}|f| = \operatorname{supp}|2f|$ and $\operatorname{chan}_{\mathcal{A}}|f| = \operatorname{chan}_{\mathcal{A}}|2f|$, and $f + h \in \boldsymbol{p}$, $g + h \in \boldsymbol{p}$ since $\boldsymbol{p}$ is a vector space, but $\mathbf{0} \not\preceq f$ in general, take e.g. any f such that $\operatorname{chan}_{\mathcal{A}} f \neq \emptyset$.

It is well known that all preorders admit a directed graph diagram where there are objects[1] for each element of the preordered set and there are arrows which go from element A to element B i.f.f. they relate under the preorder. Consider for example Diagram 2.7. The diagram of a preorder can contain very many arrows, and we often instead construct their Hasse diagrams. The Hasse diagram of a preorder is the smallest directed graph that it admits in the sense that it contains the fewest number of arrows necessary to maintain the transitivity of the relation. It is a compressed version of the "full" diagram and it is a result of a computation called *transitive reduction* which has been studied extensively [3, 4, 21]. In the case of directed acyclic graphs (DAG), the transitive reduction is unique. Preorders can in general have cycles, which makes the transitive reduction of their diagrams not necessarily unique. A possible transitive reduction of Diagram 2.7 is given in Diagram 2.8. Note that Diagram 2.8 is one of sixteen possible Hasse diagrams for this preorder because there are two strongly connected components, and they can be connected to each other in sixteen different ways. Diagrams like Diagram 2.8 and 2.7 are specific cases of a general construction known as a *quiver*[2], which are directed graphs where there can be any number of arrows between objects [5]. Although, in the quiver of

[1] Other common names for the objects of a graph are: nodes, vertices, points, or monads.
[2] Also known as a *multidigraph*.

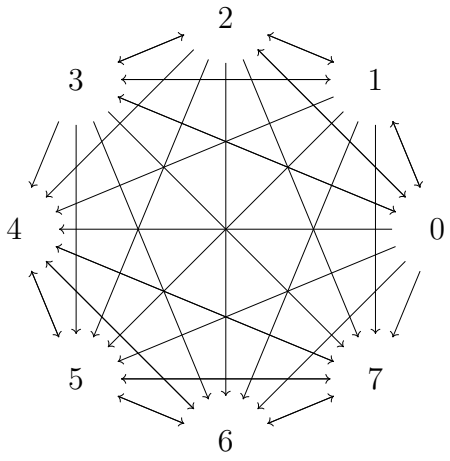


Figure 2.7: Directed graph of the preordered set $\mathbb{Z} \cap [0, 7]$ with the ordering xRy i.f.f. $\lfloor x/4 \rfloor \leq \lfloor y/4 \rfloor$. The preorder $(\mathbb{Z} \cap [0, 7], R)$ is a subset of the preorder $(\mathbb{Z}, R)$ and hence this diagram is a subgraph of the directed graph of $(\mathbb{Z}, R)$. Reflexive arrows omitted.

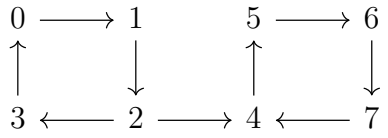


Figure 2.8: Hasse diagram of the preordered set $\mathbb{Z} \cap [0, 7]$ with the ordering xRy i.f.f. $\lfloor x/4 \rfloor \leq \lfloor y/4 \rfloor$. Reflexive arrows omitted.

a preorder there can be at most one arrow $A \to B$, since A either relates to B or does not (no arrow). We therefore have a representation of $(V, \preceq)$ as a quiver, for any $V \in \mathcal{P}(\boldsymbol{p})$ where $\mathcal{P}(\boldsymbol{p})$ denotes the power set of $\boldsymbol{p}$. While this is unimportant for our technical treatment, it is a significant realisation in a representative sense, because a quiver can be understood as the underlying structure of a category.

2.3 The Schedule Category

A *small category* is a mathematical construct which contains a set of identifiable objects together with some set of rules for how to draw arrows between those objects.

Formally, categories can be generally defined for classes[3] however we will strictly consider only the case of small classes. Therefore the following formal definition is for a small category. In the rest of the thesis the unqualified use of the word "category" refers to this definition.

Definition 2.9. (Small category) A small category $\mathcal{C}$ consists of

(i) A set $\mathrm{ob}(\mathcal{C})$ whose elements are called *objects*.

(ii) A set $\hom(\mathcal{C})$ whose elements are called *morphisms*.

(iii) A *source* mapping $\mathrm{src} : \hom(\mathcal{C}) \to \mathrm{ob}(\mathcal{C})$.

(iv) A *target* mapping $\mathrm{trg} : \hom(\mathcal{C}) \to \mathrm{ob}(\mathcal{C})$.

(v) An associative binary operation $\circ$ on $\hom(\mathcal{C})$.

such that $\exists \mathrm{id}_x \in \hom(\mathcal{C}) : \mathrm{id}_x \circ f = f,\ \forall f \in \hom(\mathrm{src}(f), x)$ and $g \circ \mathrm{id}_x = g,\ \forall g \in \hom(x, \mathrm{trg}(g)),\ \forall x \in \mathrm{ob}(\mathcal{C})$. Here $\hom(A, B)$ is the subset of morphisms with the same source $A \in \mathrm{ob}(\mathcal{C})$ and target $B \in \mathrm{ob}(\mathcal{C})$, called the homset of morphisms between A and B. The element $\mathrm{id}_x \in \hom(\mathcal{C})$ is known as the *identity morphism* of object x.

In the last section we showed that the pair $(\boldsymbol{p}, \preceq)$ is a preorder. We already know that each preorder admits a quiver, so in some sense we already have a category made for us, we just have to flesh it out.

To illuminate how this is done, consider again the preorder in Diagram 2.8. While this quiver is not a category, we can generate a category from it; the "full" quiver in Diagram 2.7 which includes all the transitive arrows is in fact its generated category. This category has elements of the set $\mathbb{Z} \cap [0, 7]$ as objects, and has the relation R as its homset. Its source mapping is defined as $\mathrm{src}((a, b)) = a,\ \forall (a, b) \in R$ and the target mapping is defined as $\mathrm{trg}((a, b)) = b,\ \forall (a, b) \in R$. The binary operation $\circ$ is defined

[3] A class is a generalization of a set which avoids Russell's paradox by disallowing containment of a class inside another class. A class which is actually a set is called a *small class*.

as $(b,c) \circ (a,b) = (a,c),\ \forall((a,b),(b,c)) \in R \times R$. We know that $\hom(\mathbb{Z} \cap [0,7])$ is closed under $\circ$ by the transitivity of R. Finally, all numbers have identity morphisms by the reflexivity of R. The categorical name for this "full" quiver is the *free* category generated by the quiver in Diagram 2.8.

The free category of a quiver is "free" in the sense that we can obtain it simply by concatenating arrows together whenever the target of one arrow is the source of the next. We saw that the structure of the free category generated by Diagram 2.8 only depended on the reflexivity and transitivity properties of the preorder on the set and not on its definition. Hence, if we have any preorder, then that preorder admits some Hasse diagram which in turn generates its free category. Of course we can skip the intermediary Hasse diagram and directly generate the preorder's free category. The Hasse diagram is simply a useful tool for drawing preorders by hand.

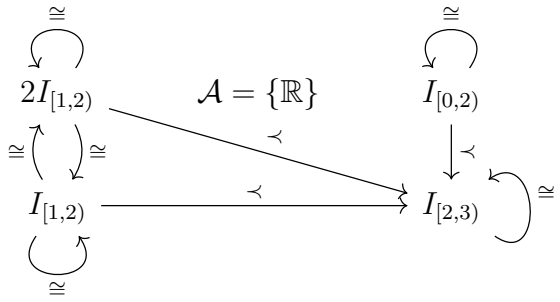


Figure 2.9: An example of a freely generated category from a subset of pulses. Here the pulses are indicator functions. The namespace is trivial and $\kappa = 0$.

Since $(\boldsymbol{p}, \preceq)$ is a preorder, it is of course possible to generate its free category. In fact, we could generate a free category for any subset of $\boldsymbol{p}$, such as the one in Diagram 2.9. However, there is a less trivial category which we can make using $(\boldsymbol{p}, \preceq)$ which is more useful, namely the category of preordered subsets of $\boldsymbol{p}$.

Definition 2.10. (Schedule category) **Sched** is the category where the object set is $\text{ob}(\textbf{Sched}) = \mathcal{P}(\boldsymbol{p})$ and morphisms are the set of order-preserving (monotone) functions between subsets of $\boldsymbol{p}$, with respect to $\preceq$, that is

$$F \in \hom(\textbf{Sched}) \iff (f \preceq g \implies F(f) \preceq F(g), \forall f, g \in \text{src}(F)) \tag{2.39}$$

Since the morphisms in **Sched** are functions, we have that its source mapping is defined as $\mathrm{src}(F) = \mathrm{dom}(F),\ \forall F \in \hom(\mathbf{Sched})$ and its target mapping is defined as $\mathrm{trg}(F) = \mathrm{cod}(F),\ \forall F \in \hom(\mathbf{Sched})$. Where $\mathrm{dom}(F)$ and $\mathrm{cod}(F)$ are the domain and codomain of F respectively. The binary operation $\circ$ is simply function composition, which is associative. Furthermore, each $V \in \mathrm{ob}(\mathbf{Sched})$ has the identity morphism $\mathrm{id}_V(x) = x$.

▷ **Sched** is a category.

Proof. Function composition of two order-preserving functions is order-preserving and the identity map leaves order unchanged. **Ord** is the category that has preordered sets as objects and order-preserving functions as morphisms. **Sched** restricts the choice of order to $\preceq$ and the preordered sets to subsets of $\boldsymbol{p}$, hence **Sched** is a subcategory of **Ord**. ■

We call the objects in **Sched** for *schedules*. Additionally, note that $\preceq$ is not antisymmetric, we have that $f \preceq g$ and $g \preceq f \implies f = g, \forall f, g \in \boldsymbol{p}$. This introduces a notion of congruence between similar schedules. Consider for instance two singleton schedules $P = \{\cos(t) I_{[0,1]}\}$ and $Q = \{\frac{1}{2}\cos(t) I_{[0,1]}\}$. P and Q are essentially the same schedule, but they contain different pulses. We can easily come up with an invertible mapping $F : P \to Q$ which satisfies $F(f) = g, \forall f \in P$, such as e.g. $p(x) \mapsto \frac{1}{2}p(x)$ with inverse mapping $p(x) \mapsto 2p(x)$. We can also show that F is a morphism in **Sched**; to do so we must show that it preserves $\preceq$. It does so because $\mathrm{supp}\,|\frac{1}{2}p| = \mathrm{supp}\,|p|$ and $\mathrm{chan}_{\mathcal{A}}\frac{1}{2}p = \mathrm{chan}_{\mathcal{A}}p, \forall p \in P$. Hence $F \in \hom(\mathbf{Sched})$. Since F is invertible, we say that it is an *isomorphism* and that P and Q are *iso-*
⊮ *morphic* schedules. A category is called *skeletal* if isomorphic objects are necessarily identical so another result is that **Sched** is not skeletal.

Not all functions which are definable on schedules are morphisms in **Sched**. Take
⊮ for instance a function F defined as the map $p(x) \mapsto p(x+1)$. We have $F \notin \hom(\mathbf{Sched})$ because $\mathrm{trg}(F) \not\subseteq \boldsymbol{p}$ in general. Indeed, $F(I_{[0,1]})$ is not a pulse, since it has a negative start time. Hence F is not a morphism from any schedule which contains $I_{[0,1]}$, because its order cannot be preserved between subsets of $\boldsymbol{p}$.

Moreover, there also exists functions which are defined between subsets of $\boldsymbol{p}$ but still do not preserve the order of $\preceq$. We have already seen examples of such functions when we proved that $\preceq$ is not translation invariant. For example let $F : P \to Q$ for $P, Q \subseteq \boldsymbol{p}$ defined by $p(x) \mapsto (p - I_{[0,1]})(x)$. Let $f = I_{[0,1]}$ and $g = I_{[2,3]}$. Clearly $f \preceq g$. We have $F(f) = \mathbf{0}$ and $F(g) = -f + g$ and therefore $F(f) \not\preceq F(g)$ because $\text{chan}_{\mathcal{A}} F(g) \neq \emptyset$. Hence, $F \notin \hom(\mathbf{Sched})$ because the order of $P = \{f, g\}$ is not preserved.

We now list two families of functions that are morphisms on any schedule in **Sched**. These are based off of the constructions developed in the last two sections.

$$\text{(delay) } D_\tau : p(x) \mapsto p(x - \tau), \tau \geq 0 \tag{2.40}$$

$$\text{(complex scaling) } S_a : p(x) \mapsto ap(x), a \in \mathbb{C} \tag{2.41}$$

Proof. For each of the above functions, we show that they preserve $\preceq$ for all subsets of $\boldsymbol{p}$ (i.e. that they are morphisms in **Sched**). First, for the *delay* function

$$f \preceq g \implies \begin{cases} \inf_{\overline{\mathbb{R}}} \text{supp}\,|f| \leq \inf_{\overline{\mathbb{R}}} \text{supp}\,|g| \\ \text{chan}_{\mathcal{A}} f = \text{chan}_{\mathcal{A}} g \end{cases} \tag{2.42}$$

$$\inf_{\overline{\mathbb{R}}} \text{supp}\,|D_\tau h| = \inf_{\overline{\mathbb{R}}} \text{supp}\,|h(x - \tau)| = \inf_{\overline{\mathbb{R}}} \text{supp}\,|h| + \tau, \forall h \in \boldsymbol{p} \tag{2.43}$$

$$\text{(2.42) and (2.43)} \implies \inf_{\overline{\mathbb{R}}} \text{supp}\,|D_\tau f| \leq \inf_{\overline{\mathbb{R}}} \text{supp}\,|D_\tau g| \tag{2.44}$$

$$\text{(2.24) and (2.44)} \implies D_\tau \in \hom(\mathbf{Sched}) \tag{2.45}$$

Second, for the *complex scaling* function, we have that $S_a \in \hom(\mathbf{Sched})$ follows from homogeneity of $\preceq$. In fact, when $a \in \mathbb{C}\backslash\{0\}$, then S_a is an isomorphism. In this case its inverse is $S_{\frac{1}{a}}$. ■

If we delay the start time of all pulses in a schedule, then the result is also a schedule with the order preserved. Similarly, if we multiply every pulse in a schedule by a non-zero complex number, then the resulting schedule is ordered the same way, and moreover we can revert the transformation by dividing by the same number. These are very intuitive results, but it is nice to see them formally fall out from the theory.

▷ Another important type of morphism is the frequency shift morphism. Perhaps unsurprisingly, we define it as

$$\text{(frequency shift) } F_\omega : p(x) \mapsto p(x)e^{i\omega x} \tag{2.46}$$

Let $V \subseteq \boldsymbol{p}$ be a schedule such that $\exists\omega : |\omega| < \omega_{max}(\mathcal{A}, \kappa, f)$, $\forall f \in V$. When this is true, we say that V is *modulatable* by ω. We now prove that F_ω is a morphism for schedules V which are modulatable by ω.

Proof.

$$F_\omega f = fe^{i\omega x} \tag{2.47}$$

$$\operatorname{supp}|fe^{i\omega x}| = \operatorname{supp}|f||e^{i\omega x}| = \operatorname{supp}|f|,\ \forall x \tag{2.48}$$

$$\operatorname{chan}_\mathcal{A} fe^{i\omega x} = \{[\xi + \omega]_\mathcal{A} : \xi \in \operatorname{peaks}(f, \kappa)\} \tag{2.49}$$

$$|\omega| < \omega_{max}(\mathcal{A}, \kappa, f) \implies [\xi \pm \omega]_\mathcal{A} = [\xi]_\mathcal{A},\ \forall \xi \in \operatorname{peaks}(f, \kappa) \tag{2.50}$$

$$|\omega| < \omega_{max}(\mathcal{A}, \kappa, f) \implies \operatorname{chan}_\mathcal{A} fe^{i\omega x} = \operatorname{chan}_\mathcal{A} f \tag{2.51}$$

$$(f \preceq g \implies F_\omega f \preceq F_\omega g,\ \forall f, g \in V) \implies F_\omega \in \hom(\mathbf{Sched}) \tag{2.52}$$

■

The existence of the frequency shift morphism in a homset between two schedules in **Sched** depends on the namespace $\mathcal{A}$, the κ threshold, and the source schedule. If e.g. the trivial namespace $\mathcal{A} = \{\mathbb{R}\}$ is used, then every schedule is modulatable by any ω. The F_ω morphism does not exist from schedules V where $\exists f \in V : |\omega| \not< \omega_{max}(\mathcal{A}, \kappa, f)$ where $\mathcal{A}$ is some non-trivial namespace. For example, in Figure 2.5, we have that the $\operatorname{chan}_\mathcal{A} f \neq \operatorname{chan}_\mathcal{A} F_\omega f$ for $\omega = -35\,\text{Hz}$, which means that the order of V is not preserved by F_ω in these cases. F_ω is also an isomorphism on schedules modulatable by ω because any schedule modulatable by ω is also modulatable by $-\omega$. This follows from the fact that $\omega_{max}(\mathcal{A}, \kappa, f)$ is an unsigned quantity.

▷ ⋪

The singleton preordered sets, such as e.g. $P = \{\cos(t)I_{[0,1]}\}$ and $Q = \{\frac{1}{2}\cos(t)I_{[0,1]}\}$ are *terminal objects* of **Sched**. Terminal here means that for every schedule X there exists exactly one monotone function from X to that terminal object. This function is the surjective function which maps all pulses in the source schedule to the same

pulse in the target schedule. We know that these surjections are morphisms because the order of the source schedule is always preserved as one pulse is always ordered, by the reflexivity of $\preceq$. We have already shown that there exists an isomorphism between P and Q given above. In fact all terminal objects in **Sched** are isomorphic to each other. An isomorphism between terminal objects always exists because the surjective function which maps all pulses in the source schedule to the same pulse is invertible on terminal objects. We can invert it between terminal objects because there is only one pulse to map.

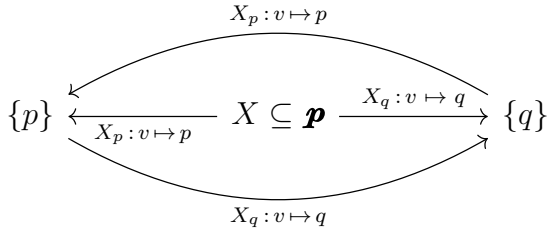


Figure 2.10: All terminal objects in **Sched** are isomorphic.

In Diagram 2.10, $\{p\}$, $\{q\}$ and X_p, X_q are the unique surjective morphisms from X to $\{p\}$ and $\{q\}$ respectively. We know that these are unique, because for any other morphism $F : X \to \{p\}$ for any $X \subseteq \boldsymbol{p}$ we have $X_p = F$ by function equality, since $\text{src}(F) = \text{src}(X_p)$, $\text{trg}(F) = \text{trg}(X_p)$ and $X_p(v) = F(v)$, $\forall v \in X$. So what we are saying is that for any singleton schedule $\{p\} \subseteq \boldsymbol{p}$ we have $\text{card}(\hom(X, \{p\})) = 1$, $\forall X \subseteq \boldsymbol{p}$, which is exactly the definition of terminal object. The same argument follows for every singleton schedule.

In category theory, every notion has a dual notion, because we can always flip the arrows. The dual notion of a terminal object, is an *initial object* [36, p. 53]. While not all categories have both terminal objects and initial objects, we have that the initial object in **Sched** is the empty schedule $\emptyset \subseteq \boldsymbol{p}$. This is because there exists a unique function which goes from $\emptyset$ to any schedule, namely the function which maps nothing to the pulses in the target schedule. These are peculiar functions because they can never be evaluated, but they are technically mappings because they map all elements in the source schedule (no pulses) to a uniquely determined pulse in the target schedule (whichever). Moreover, they are morphisms in **Sched** because they

clearly preserve the order of the source schedule, since there is no order to preserve. The empty set is also to the initial object of **Ord**.

In summary, we now have a vector space $\boldsymbol{p}$, endowed with a preorder $\preceq$. This allows us to generate graphs from any schedule. Take e.g. the schedule in Figure 1.8. If we use the same labels for the pulses as the ones in the Figure, and fix a namespace such that it has channels centered around the indicated frequency spanning 20 MHz with say $\kappa = 0.5$, then it has the diagram

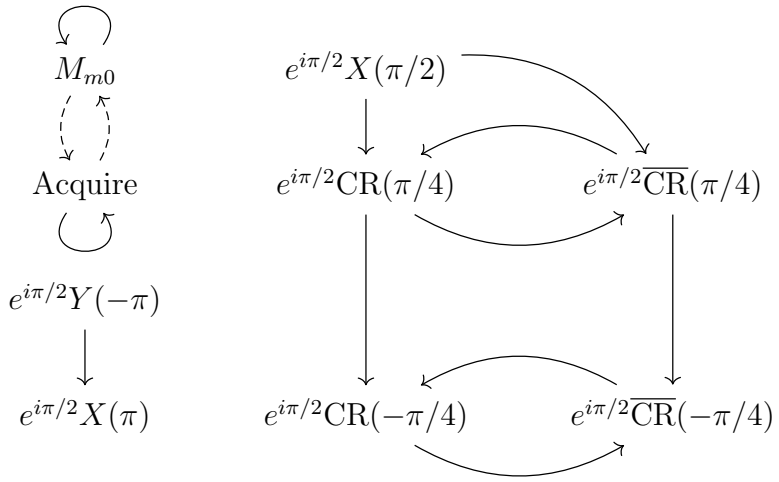


Figure 2.11: The last two reference frame changes of `D1` and `U0` have been omitted because there are no pulses after these operations. Acquire channels are virtual in the sense that they do not correspond to a frequency. Therefore it is up to interpretation how to relate "acquire pulses" with other pulses. There are two straightforward ways to do so; the first way is to say that acquire pulses are partially simultaneous pulses to readout pulses on measure channels. The second way is to say that, if a is an acquire pulse, then $\mathrm{chan}_{\mathcal{A}} a = \emptyset$, which would relate it to the zero pulse $\mathbf{0}$. For the translation purposes of this thesis, both are valid. For the implementation, the former option was chosen. This is discussed further in Section 3.2.

We have also constructed a category **Sched**, which allows us to categorise families of order preserving mappings between schedules. This category allows us to generate large graphs, from pulse schedules.

3

Implementation

This chapter details how the scheduling theory developed in Chapter 2 was used to implement a small server written in Python 3.9 which can schedule and execute arbitrary pulse schedules on quantum computers using superconducting qubits. The following paragraph is an overview on the functionality of the server.

The server, called `tergite-quantify-connector` (`tqc`), is a RESTful server which accepts serialised Qiskit OpenPulse schedules sent to it as `POST` requests and translates them to Quantify schedules, which are then executed on a quantum computer via the use of QBLOX instruments such as the Pulsar QRM, Pulsar QCM, or the QBLOX Cluster. It can also orchestrate any instrument with a QCoDeS driver such as e.g. the Rohde & Schwarz SGS100A SGMA RF source. The `tqc` server was implemented to be a part of the Wallenberg Centre for Quantum Technology (WACQT) software stack called *Tergite*, whose development is led by Miroslav Dobsicek at Chalmers. With this, pulse schedules can be transmitted to it from any computer on the campus intranet with Qiskit OpenPulse installed. Moreover, it can also be accessed from external networks. For a scheme of some different possible interfacing methods to `tqc`, see Figure 3.1. External job queuing has not yet been rigorously tested, but with the help of Miroslav, a successful X-gate experiment was ran on a Chalmers quantum computer using `tqc` from the LUMI super computer in Finland.

When `tqc` receives an incoming job, it assigns it a universally unique identification number (UUID), which is registered in the Tergite database. This UUID uniquely identifies the experimental data associated to an experiment. To each experiment there are two pieces of data: a hierarchical data format (HDF) binary file containing measurement data and a unicode logfile. Both of these are associated to the UUID and are accessible via Tergite. The logfile contains information of the computation

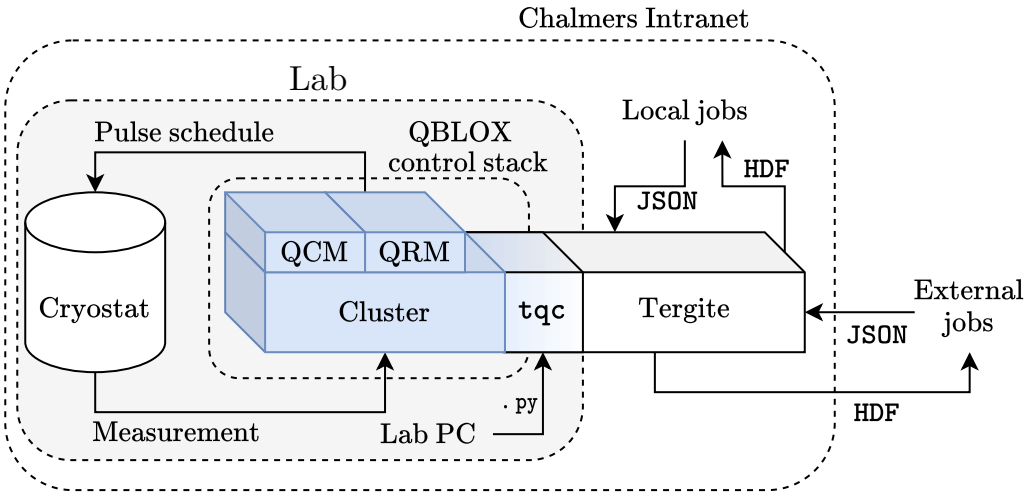


Figure 3.1: Interfacing with a Chalmers quantum computer using `tqc`. Jobs are serializations of OpenPulse schedules which are transmitted to Tergite as `JSON` payloads. All measurement data is stored in an HDF file, which is accessible through Tergite. It is also possible to use `tqc` directly in a laboratory setting by creating OpenPulse schedules locally with Python in e.g. Jupyter notebooks.

as it progresses, such as the current frequency of all local oscillators in the instrument orchestration and their currently running assembly programs. Furthermore, significant effort was placed in making the logfiles compressed and human readable.

The results of a completed quantum computer experiment can be accessed via a `GET` request to a Tergite server, which either retrieves the full HDF binary data file or just the essential data, such as e.g. whether a qubit was measured to be in $|0\rangle$ or $|1\rangle$. The `tqc` can also be used as a library when conducting experiments locally. In this mode, it additionally provides an interface for the HDF binary data file, which can parse certain experimental results as Quantify `xarray` [28] datasets, thus making it portable with the Quantify analysis classes. Of course, if desired the HDF data can also be accessed directly with a HDF library such as `h5py` [44]. The storage file memory layout is detailed further in Section 3.3.

The `tqc` server, which is detailed further in Section 3.2, utilizes the scheduling theory of Chapter 2 in the sense that the translation is implemented as a parsing of a schedule

graph generated from an OpenPulse schedule.

We now give an overview of the hardware connected to the `tqc` server, the architecture of the server, the general program flow of the translational mechanism, and the output storage format of the experiment data. Effort was put into including as little code as possible in the thesis because of the transient nature of code. Instead, we describe the functions of the library by describing its component modules, their intended purpose, and how they relate to one another. However, working code examples for how to repeat all measurement in this thesis are provided in the Appendix. The library is fully open source, and has been made available on the MC2 code repository.

3.1 Hardware

All experiments in this thesis were conducted using the cryostat "Pingu" at the MC2 department. Specifically the B-side of Pingu was used. At the time when the experiments were conducted, the chip installed in Pingu B had two qubits (with respective readout resonators). Only one of the qubits was working, but both resonators were functional. Due to these limitations, we will only cover the hardware necessary to control and measure a single qubit. In particular, this means that we will not cover the experimental setup necessary for entangling computations. Moreover, during the initial stage of the work, only the Pulsar QCM and the Pulsar QRM out of the ordered QBLOX modules had arrived, and so only these were available for use. The Pulsar QCM & QRM are more than enough for single qubit experiments however, so this was not an issue. The Pulsar QCM & QRM both have a bandwidth of $[0, 300\,\mathrm{MHz}]$. Consequently, we cover an experimental setup which utilizes external analog mixers, and we do not cover a setup which uses the RF modules of the QBLOX Cluster, for example. It was known already (from measurements by Eleftherios Moschandreou and Christian Križan) that the $|0\rangle \rightarrow |1\rangle$ excitation frequency for the working qubit was in the $4.7\,\mathrm{GHz}$ range, and that both the resonators were in the $6.3\,\mathrm{GHz}$ range, so the mixers were needed for frequency upconversion. Two of the I/Q mixers used were of the Rohde & Schwarz SGS100A SGMA RF model, specifically the mixers A

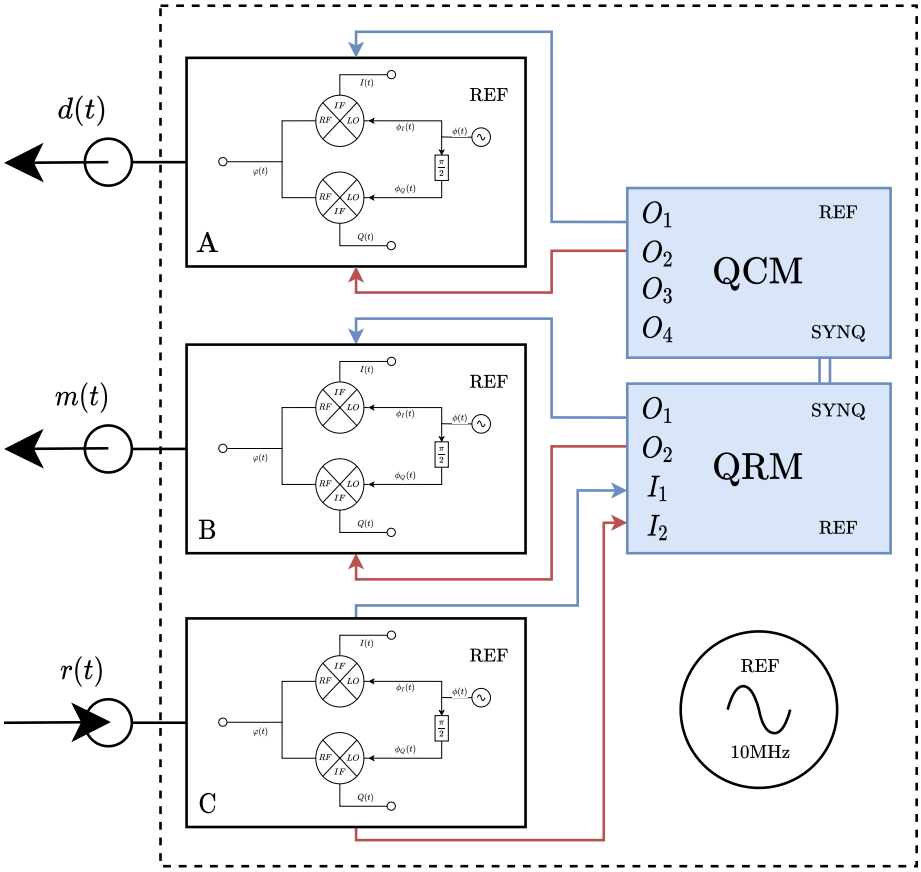


Figure 3.2: Experimental setup used for all experiments in this thesis. Cryostat not included in picture.

and B in Figure 3.2 were SGS100A mixers. The demodulating I/Q mixer C was a custom build.

In Figure 3.2 the label $d(t)$ corresponds to the signal in the cable which leads to the drive port of the working qubit in the two-qubit chip of Pingu B. The label $m(t)$ corresponds to the signal in the cable which leads to the readout port of the qubit chip and the label $r(t)$ corresponds to the cable carrying the reflection coming out of the cryostat. All qubits were connected to the same readout line, so both resonators in Pingu B could be addressed with $m(t)$.

The local oscillator frequencies $\omega_A,\ \omega_B$ of the analog mixers were set using QCoDeS commands via the Quantify instrument orchestration. Both d and m were modulated in two stages. First they were amplitude modulated inside the QCM and QRM. Each is modulated to some frequency in the IF frequency range $[0, 300\,\text{MHz}]$. Pulse schedules d and m are then further modulated to the their respective target frequencies using the analog I/Q mixers A and B respectively.

Both I/Q mixers A and B were imbalanced. Mixer balancing was performed by first sending a very long pulse to the imbalanced mixers, which were connected to a spectrum analyzer. The mixers were then balanced by adjusting the amplitude ratio, I and Q DC-offsets, and phase until the lower sideband and the LO peak dissapeared below the noise level and only the upper sideband remained.

The reflection $r(t)$ is demodulated in two stages in the inverse fashion relative to d and m. After demodulation, the baseband signal of the reflection is integrated inside the QRM in order measure $b_2(t)$ over the integration window specified in the pulse schedule, which yields a digital complex number stored in the memory of the QRM. The QRM can also store entire traces.

The real component of the pulse schedules was associated to the in-phase component (blue) $I(t)$, which was assigned the index 1 for the outputs and inputs of the QCM and QRM. Conversely, the imaginary component (red) was associated to $Q(t)$ and index 2 respectively. All instruments were connected to a 10 MHz reference clock for synchronization. The SYNQ ports of the QRM and QCM were also connected with a USB cable in order for them to use the proprietary SYNQ protocol of QBLOX devices. There were also attenuators and filters along some signal paths, but they have been omitted from the figure as the core mechanism is the same without them. All instruments in the orchestration were communicated to using an Ethernet connection over the cryolab local subnet.

3.2 Server

The `tqc` server is a RESTful Python server implemented asynchronously with a FastAPI module sitting on top of the Quantify instrument orchestration platform. The quality of it being RESTful roughly means that there is no state maintained in-between jobs sent to the server so that every job is treated identically. The FastAPI module allows this interfacing to occur over the web. Every job is forked as a new thread, so every job is executed within its own local context. However, the drivers used by the control stack need to be globally accessible in shared memory between threads, since new threads should not have to restart the hardware drivers. Therefore, after every job, the server always fully resets the control hardware to a well defined state using the QBLOX `reset` command in order to maintain statelessness between jobs.

The `tqc` server consists of two modules, the connector & the scheduler. The connector is extremely minimal. It can accept certain `GET` requests which provide information about the current devices connected to `tqc`. It can also accept `POST` requests of OpenPulse schedules in the form of a `JSON` structure called a `QObj` [32], which is a `JSON` formatting scheme created by IBM Qiskit. A `JSON` structure is essentially just text with a very specific formatting, and the `QObj` just amounts to parametric descriptions of functions with specified time dynamics.

When a `QObj` job is received by the connector, it first registers the job with a Tergite UUID. Additionally, a local Quantify identification number is generated and associated to the job, called a TUID. The TUID is used by `tqc` to store data locally. It can also be used to find HDF files in a laboratory context using the Quantify core dataset handling helper functions; a particularly useful helper function is `locate_experiment_container` which will return the directory of a completed `tqc` job on the lab PC. The `QObj` structure is then de-serialized into a Python dictionary and is passed directly to the scheduler. The scheduler is the module of `tqc` which implements the translation mechanism between OpenPulse and Quantify schedules.

The scheduler consists of three Python classes, called `Experiment`, `Program`, and `Instruction`. Each of these is implemented as a container class. This just means that the only thing that happens during instance construction is assignment of attributes. These attributes are then accessed using the *memoization* pattern [36, p. 57]. Memoization was implemented using the Python `functools` function decorators `@functools.cache` and `@functools.property`. Methods which were not instance dependent were decorated with `@staticmethod`. The program flow arises from a creation of an `Experiment` object whose `schedule` property is called by control hardware when it is ready to compile and execute a schedule. This causes a cascade of function calls which implement the translational mechanism. Since many of the properties in the scheduler classes use the `@functools.cache` decorator, most of the translation just amounts to cache lookups. We now describe in closer detail the program flow of the scheduler.

The translational mechanism of the scheduler uses a simplified version of the preorder relation $\preceq$ which was developed in Chapter 2. The simplification is that the Fourier transform was not used to compute the $\text{chan}_{\mathcal{A}}$ set. Instead, the OpenPulse channel was used to determine frequency occupancy. This was done for two reasons. The first is that for the simple experiments conducted in this thesis, we do not need an algorithmic way to determine frequency occupancy. All schedules used had at most four pulses so it is easy to manually generate and analyse the schedules. The second reason is that when conducting experiments with only one qubit there is no interference in the frequency domain that we need to worry about, especially when it has a dedicated drive cable. Specifically, the preorder was hardcoded to only check whether two pulses belonged to the same OpenPulse channel (by checking for string equality of the channel names), instead of computing the Fourier transform and checking (2.33).

This simplification limits the available pulses that one can utilize. The limitation stems from that the OpenPulse channel does not actually restrict the frequency of the pulse, it only indicates a procedure. For example, say we want send a control pulse targeted at a specific characteristic frequency. If the modulating signal itself has some frequency, then the modulated carrier will be offset by that frequency, as we know from (1.9). Hence, the resulting pulse might be off-target or even play in

the wrong channel.

The above example might seem contrived, but it has significant implications due to *spectral crowding* [20, 26]. As more superconducting qubits are added to a quantum computer, knowledge of the exact frequency that a pulse addresses - in relation to other pulses in a schedule - becomes important to consider. Especially, in the case of many qubits coupled together, since then there are a large number of closely spaced transitions that need to be avoided during quantum gate operations [26, p. 1]. However, this partial implementation of the theory is justified in this context, because we are just dealing with one qubit.

We now cover the computational procedure of the scheduler. The order of operations of the scheduler are as follows.

1. First, the de-serialized `QObj` received from the connector is converted into a set of pulses by constructing an `Instruction` instance from each pulse.
2. The `Instruction` constructor parses the pulse so that all instructions have assigned `name`, `t0` (start time), `ch` ($\text{chan}_{\mathcal{A}}$, i.e. OpenPulse channel name in this case), and `duration` attributes.
3. The set of `Instruction`s is passed to the `Experiment` constructor.
4. During object instantiation, the `Experiment` constructor uses the preorder $\preceq$ to build a graph from the `Instruction`s which is stored as an attribute. Additionally, the relative time between two pulses $g.\texttt{t0} - (f.\texttt{t0} + f.\texttt{duration})$ is computed and associated to each edge as data. Except from the $\text{chan}_{\mathcal{A}}$ simplification, the graph of an instantiated `Experiment` is equivalent to the schedule graphs developed in Chapter 2 (with extra data associated to each edge). The theory in Chapter 2 did not specify that pulses had to have finite durations. However, all pulses that we use in practice have finite duration, so the relative time will always be finite.
5. When the hardware requests a Quantify schedule, then the graph in the `Experiment` class is traversed to build a `Program` instance. At the beginning of the traversal, the `Program` class is essentially a wrapper class around an empty Quantify schedule. When the graph traversal terminates, then the `Program` instance

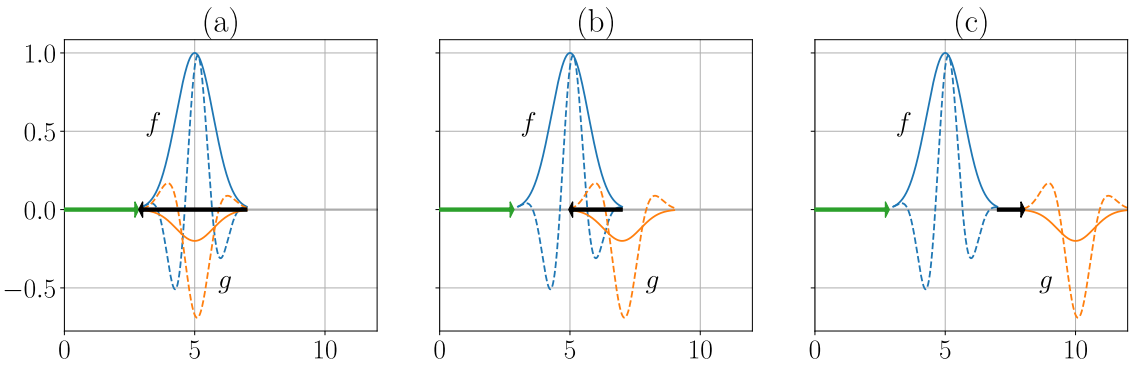


Figure 3.3: The three possible configurations for two successive pulses f, g on an OpenPulse channel. Here the real component of the pulses is plotted in with a full line and the imaginary component with a dashed line. (a) f and g are simultaneous. (b) f and g are partially simultaneous. (c) f precedes g. Cases (a) and (b) can be identified with a negative relative time between a pulse and its reference pulse, denoted by the black arrow. The green arrow originates from the pulse preceding f. If such a pulse does not exist, then it originates from the initial object.

contains the translated Quantify schedule. Using the sampled waveforms produced by Qiskit, all pulses are interpolated and mapped to an object which is compilable to Q1ASM by the Quantify instrument orchestration. The `Program` class also deals with some special Quantify related cases, such as how to signal to the control stack that a measurement should begin.

6. The constructed `Program` parsed from the `Experiment` graph is returned to the `tqc` connector module, which simply compiles and runs it with Quantify instrument orchestration routines. The connector will execute compiled `Programs` on whatever QBLOX instruments are currently connected to `tqc`. This is why `tqc` works on any QBLOX hardware.

Quantify schedules are (currently) different from OpenPulse schedules mainly because they use different conventions for the meaning of "start time". In OpenPulse, the start time of a pulse is given in relation to the absolute time of the schedule whereas in Quantify the start time is given in relation to a reference pulse. The parsing of the `Experiment` graph (step 5. above) is therefore implemented by first

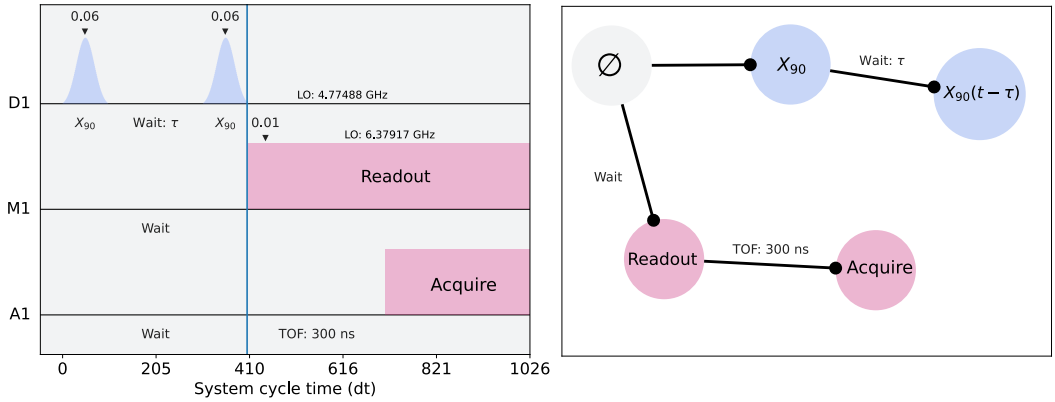


Figure 3.4: An example schedule and its simplified (no Fourier transform) `Experiment` graph. Another option is to define the relative time between the start times and associate this to the edge weight. If this is done, then we cannot identify simultaneous pulses solely from the edge weights, because we need to supply the duration information.

creating a reference pulse at the beginning of the Quantify schedule in `Program` which has zero duration. This object is named the *initial object*. The name is meant to highlight that it behaves similarly to the initial object in **Sched**, from Chapter 2. The first pulse which is played on any channel then has the initial object as a reference pulse. Subsequent pulses on any channels simply use the preceding pulse as a reference pulse, with respect to the preorder relation $\preceq$ that generated the `Experiment` graph. The relative duration between any pulse and its preceding reference pulse is known because of step 4. in the above procedure. In other words, we specify the reference pulse, if it exists, and delay it by the edge weight of step 4. For an illustration, see Figure 3.3.

Note that the relative duration may be negative when two pulses are simultaneous or partially simultaneous. This will happen, for instance, when specifying measurement operations, because the integration window can be treated as a pulse on a virtual acquisition channel. This is useful, because it allows one to specify integration weights very easily by simply modulating the pulse in the acquire channel.

If the reference pulse does not exist for a given pulse in a channel, then we say that the reference pulse is the initial object. In this case, we know that this is the

first pulse on a channel, so we set the relative time to the initial object as some constant value, called the *buffer time*. The buffer time is the amount of time that we should wait before running a new schedule, and it corresponds intimately with the decoherence time (see Section 4.4) of the qubits. In this thesis, a buffer time of $300\,\mu$s was used as a default setting. Another way to interpret the initial object is as the head of a linked list, where each channel is a linked list, for readers familiar with that.

For an example of a parsing of an `Experiment` graph into a Quantify schedule, see Figure 3.4, which generates the following Quantify schedule:

```
2022-03-29 18:10:04,930 - INFO - Timing table:
port      clock         is_acquisition       abs_time    duration  operation
--------  ------------  ----------------  -----------  ----------  ------------------------------
          cl0.baseband  False             0              0         initial_object-cl0.baseband-0
          d1            False             0.0003         0         setf-d1-0
drive0    d1            False             0.0003         1e-07     gaussian-d1-0
          m1            False             0.0003         0         setf-m1-0
          m1            False             0.0003         2.04e-07  delay-m1-0
          d1            False             0.0003001      4e-09     delay-d1-100
drive0    d1            False             0.000300104    1e-07     gaussian-d1-104
readout0  m1            False             0.000300204    3.5e-06   constant-m1-204
readout0  m1            True              0.000300504    3e-06     ssb_integration_complex-m1-504
```

The compiled Q1ASM program for this schedule is included on the next page. Both of these output extracts are from the logging submodule of the `tqc` connector. The logging submodule keeps track of the currently executing Q1ASM program of all sequencers of all instruments in the instrument orchestration and the corresponding schedules which generated those programs. Moreover, in order to compress the logfiles, the symmetric difference between the last executed Q1ASM program and the currently executing program is computed and only the lines which are changed in-between iterations are stored in the log file.

```
2022-03-29 18:10:04,918 - INFO - Q1ASM program:
qrm:
    seq1:
0:              wait_sync           4
1:              upd_param           4
2:              set_mrk             15                  # set markers to 15
3:              wait                4                   # Latency correction of 0 ns.
4:              move                2000,R0             # iterator for loop with label start
5:      start:
6:              reset_ph
7:              upd_param           4
8:              wait                65532               # auto generated wait (300000 ns)
9:              wait                65532               # auto generated wait (300000 ns)
10:             wait                65532               # auto generated wait (300000 ns)
11:             wait                65532               # auto generated wait (300000 ns)
12:             wait                37872               # auto generated wait (300000 ns)
13:             wait                204                 # auto generated wait (204 ns)
14:             set_awg_gain        917,0               # setting gain for constant-m1-204
15:             play                0,1,4               # play constant-m1-204 (3500 ns)
16:             wait                296                 # auto generated wait (296 ns)
17:             acquire             1,0,4
18:             wait                3196                # auto generated wait (3196 ns)
19:             loop                R0,@start
20:             set_mrk             0                   # set markers to 0
21:             upd_param           4
22:             stop
qcm:
    seq1:
0:              wait_sync           4
1:              upd_param           4
2:              set_mrk             15                  # set markers to 15
3:              wait                4                   # Latency correction of 0 ns.
4:              move                2000,R0             # iterator for loop with label start
5:      start:
6:              reset_ph
7:              upd_param           4
8:              wait                65532               # auto generated wait (300000 ns)
9:              wait                65532               # auto generated wait (300000 ns)
10:             wait                65532               # auto generated wait (300000 ns)
11:             wait                65532               # auto generated wait (300000 ns)
12:             wait                37872               # auto generated wait (300000 ns)
13:             set_awg_gain        851,0               # setting gain for gaussian-d1-0
14:             play                0,1,4               # play gaussian-d1-0 (100 ns)
15:             wait                96                  # auto generated wait (96 ns)
16:             wait                4                   # auto generated wait (4 ns)
17:             set_awg_gain        851,0               # setting gain for gaussian-d1-104
18:             play                0,1,4               # play gaussian-d1-104 (100 ns)
19:             wait                3596                # auto generated wait (3596 ns)
20:             loop                R0,@start
21:             set_mrk             0                   # set markers to 0
22:             upd_param           4
23:             stop
```

For example, if we were to increase the delay τ in the schedule in Figure 3.4 by 104 ns and execute it immediately after, then only the following lines of the Q1ASM program are printed in the log file

```
2022-03-29 18:10:06,721 - INFO - Q1ASM program:
qrm:
    seq1:
13: ---         wait                204                 # auto generated wait (204 ns)
13: +++         wait                308                 # auto generated wait (308 ns)
14: ---         set_awg_gain        917,0               # setting gain for constant-m1-204
14: +++         set_awg_gain        917,0               # setting gain for constant-m1-308
15: ---         play                0,1,4               # play constant-m1-204 (3500 ns)
15: +++         play                0,1,4               # play constant-m1-308 (3500 ns)
qcm:
    seq1:
16: ---         wait                4                   # auto generated wait (4 ns)
16: +++         wait                108                 # auto generated wait (108 ns)
17: ---         set_awg_gain        851,0               # setting gain for gaussian-d1-104
17: +++         set_awg_gain        851,0               # setting gain for gaussian-d1-208
18: ---         play                0,1,4               # play gaussian-d1-104 (100 ns)
18: +++         play                0,1,4               # play gaussian-d1-208 (100 ns)
```

The marking `+++` indicates that a new line has been added to the current program, comparing with the last and the marking, `---` indicates that a line has been removed. In the case where the symmetric difference between the last output and the new output is larger than the output itself, then simply the new output is printed. The only (non-comment) change between this and the last output is that we wait for 104 ns longer between the two X_{90} pulses on line 16 of the QCM and consequently have to wait for 104 ns longer until the readout pulse is transmitted on line 13 of the QRM. A similar symmetric difference is computed for the timing table (the Quantify schedules) in the log file, but that has been omitted here, for brevity.

The logging submodule also keeps track of the current `Program` settings. The important settings are: the buffer time (which has already been mentioned), the measurement level, and the measurement return type.

The measurement level is an integer which specifies the amount of post-processing

which should be applied to the measurement data. If the measurement level is zero, then the raw traces are returned, and no post-processing is performed. If the measurement level is one, then all traces will be integrated. Currently, there is no way to specify integration weights, but as has already been mentioned it is straightforward to implement this by modulating a virtual acquire channel. If the measurement level is two, then this indicates that the the readout will be classified with a binary discriminator before it is returned to the user. This has only been partially implemented with a hardcoded discrimination for the working qubit in Pingu B. However, this is not a lasting solution, and was done simply to conduct a state discrimination experiment (covered in Section 4.5). Implementing a full discriminator module for `tqc` was deemed out of scope for this thesis and is a possible extension of this work, which is discussed further in the last chapter. Measurement levels higher than two currently have no assigned meaning.

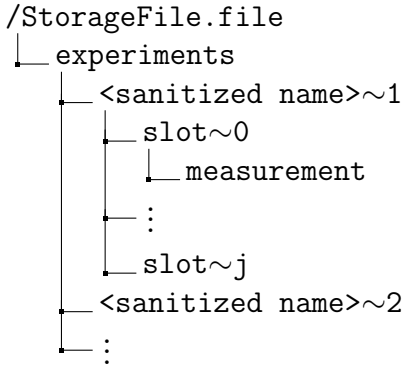
The measurement return setting specifies whether the measurement results should be returned as averages over all shots or as single shots. Shot averaging is performed on board the QRM[1]. The combination of measurement level zero and single shot measurements is disabled. It is disabled because the combination is currently not supported by the Quantify scheduler (version 0.6.0) and will give an error when attempted. The reason is presumably because it requires a three-dimensional memory array, which would consume a lot of memory.

Both the measurement level and the measurement return can be specified at a user level by specifying them as keyword arguments to the function transmitting the `QObj` job to `tqc` at the Qiskit level (i.e. using the Tergite Qiskit connector), see the Appendix for further detail. All experiments included in a single `QObj` job will have the same `Program` settings and the default settings are integrated measurement level with average measurement return. Moreover, the HDF storage file will be differently formatted depending on the `Program` settings. We now give an overview of how the storage file is implemented.

[1]The Quantify setting for average measurement return is currently called `BinMode.AVERAGE` and the setting for single shot measurement return is currently called `BinMode.APPEND`.

3.3 Storage

The HDF file produced by `tqc` after a successful job is very minimal and only consists of two groups. The first group is called `experiments` and the second group is called `header`. The `experiments` subgroup contains some finite number of subgroups, one for each experiment in the `QObj`.



The name of the subgroup is a sanitized version of the name assigned to the corresponding schedule at the Qiskit level. For instance, if we make a schedule called "My Test" in Qiskit, then its sanitized name will be `my_test∼i` where `i` is the index (starting from 1) of the experiment as it appeared in the `QObj`. The reason to sanitize names in this fashion is to maintain a correspondence to the OpenPulse schedule while simultaneously filtering schedule names which are illegal `h5py` subgroup names, such as e.g. those containing the forward slash character. Note that the renaming is injective even if two schedules in the `QObj` have the same Qiskit schedule name, because we include the experiment index in the name.

Each sanitized experiment subgroup then has some finite number of subgroups, called *slots*. A slot contains a single HDF dataset which contains the measurement of the qubit assigned to the slot. This dataset effectively corresponds to the classical register slot in OpenQASM. Currently, it is only possible to store qubit q measurements in slot q, but this is not strictly a necessity and a slot re-mapping can easily be implemented, if so desired. The measurement dataset is always a matrix, and can be written with notation from Section 1.3; let $b_2^{(ik)}(t)$ be the k^{th} trace of the i^{th} measurement of

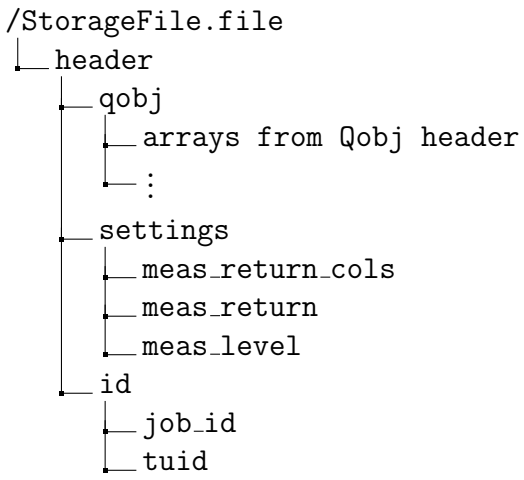
qubit q and let A_i be the integration window for the measurement. The value of the measurement matrix at the i^{th} row and the j^{th} column is

	Single shots	Averaged shots
Trace	$\times$ (Too much data)	$\frac{1}{n}\sum_{k}^{n} b_2^{(ik)}(t_j)$
Integrated	$\int_{A_i} b_2^{(ij)}(t)dt$	$\frac{1}{n}\sum_{k}^{n} \int_{A_i} b_2^{(ik)}(t)dt$

Figure 3.5: The actual values of the memory depends on the sampling time of the control stack. For the QBLOX devices used in this thesis, the time between two consecutive samples was 4 ns. For average traces, $b_2^{(ik)}(t_j)$ denotes the j^{th} sample of the k^{th} trace.

Note that when using integrated measurement level and averaged measurement return, then the columns of the measurement matrix are all equal, so we can compress it down to a single column vector. The reason for this organization is that it ensures that the storage file is always rectangular in memory, which simplifies processing. A consequence of this is that if a trace readout measurement level is specified, then all traces need to have the same length or some will have invalid samples in the measurement matrix. However, this is currently also a restriction in Quantify, since Quantify will give a run-time error for unequal length traces, so it is an inconsequential design choice at the moment for the current application.

There are three subgroups inside `header`, called `qobj`, `settings`, and `id`.



The `id` group just contains both the components of the UUID of the job as the names of empty subgroups, since this is an easy way to store strings inside HDF files.

The `settings` subgroup contains three datasets which dictate the measurement settings of the job. Each dataset `meas_level`, `meas_return`, and `meas_return_cols` contains a single integer. The first two encode the measurement level and measurement return settings of `tqc` respectively. The integer value `meas_return_cols` is automatically inferred by `tqc`. It is simply the number of columns in each of the experiment datasets. This number serves only to alleviate the parsing of HDF files.

The `qobj` subgroup also is meant to alleviate the parsing of storage files. Any array that was sent in the metadata of the `QObj` payload for the job corresponding to this storage file's UUID is included as a dataset in `qobj`. The purpose of this is to simplify the plotting of parametric sweeps. For example, say we send 600 schedules which implement a frequency sweep, where one schedule gives one data point for a single frequency. In order to analyse the measurement data, we parse the storage file and plot the measurement data, but to plot it against the correct frequencies we need to know which frequencies were included in the sweep. In other words, we just store the independent variable in the header. The reason this has to be done by array passing instead of by inference from schedules is because we should be able to plot against

any variable, not just the ones we can infer from the schedules inside the `QObj`.

Quantify has implemented their own dataset design for storing measurement data acquired using QBLOX instruments. The Quantify dataset design depends on the use of coordinates in an `xarray` [28]. This is very useful when conducting parametric sweeps, because one can affix independent variables (termed *settables* in Quantify) and dependent variables (termed *gettables* in Quantify) and then relate these to the coordinates. This simplifies greatly the plotting of parametric sweeps. In the case when graphs generated from OpenPulse `QObj`ects directly correspond to N-dimensional parametric sweeps, then we can create a mapping from storage files to the Quantify datasets. A rudimentary mapping[2] has been implemented for the most common case when integrated measurement level is chosen and the measurement return is set to average over all shots. However, in general, it is not possible to use a Quantify dataset to store the measurement output from `tqc`. The reason for why it is impossible is because the set of graphs generated from the experiments of a `QObj` do not necessarily relate to one another along the same dimensions. For example, we could send all single qubit characterisation experiments in a single `QObj`, but the independent variables across these experiments are different, and as such there is no straightforward way to associate them to coordinates in the Quantify sense. This is also the reason why a new HDF wrapper was designed.

[2]The implementation is the method `StorageFile.as_sweep` in the `tqc` repository.

4

Experiments

This chapter contains the experimental results that were obtained from single qubit characterisation experiments conducted on a quantum chip with two superconducting qubits in Pingu B. As mentioned in Section 3.1, only one qubit was working but both readout resonators were functional. Unless explicitly specified, all experiments below were averaged over 2000 shots and all experiments also used a buffer time of 300 μs. The data has been made available in an MC2 repository and each TUID presented associates to a HDF storage file and an experiment logfile generated by `tqc`. All metadata collected by `tqc`, such as exact time of each experiment is contained in the logfile. The schedules for all experiments were created using OpenPulse on a laptop without Quantify installed and transmitted to `tqc` as `QObj`ects over the Chalmers intranet. All experiments were translated by and executed using the `tqc` server connected to QBLOX quantum control hardware as displayed in Figure 3.2. Every measurement was performed with the integrated measurement level and average measurement return settings, except for the final state-discrimination experiment, which used single shots measurement return. Hence, when the term "measurement" is used in this chapter it specifically means integrated trace.

The single qubit characterisation experiments are all in the form of one or two dimensional parametric sweeps, meaning that all parameters are fixed except for one or two, which are linearly incremented over some interval. Since the integrated measurement level was used, all one dimensional sweeps can be plotted as functions of the sweep parameter and all two dimensional sweeps can be plotted as images, with each axis corresponding to the two sweep parameters. For every sweep two instances of the OpenPule schedule which generated the `Experiment` graphs are shown which differ only in their parameterisations.

4.1 Resonator Spectroscopy

The goal of the resonator specroscopy sweep is to determine if the readout resonator is working as intended. A working readout resonator can be established if the magnitude of the measurement drops to a very small value close to the designed frequency of the resonator with a frequency response that matches the theoretical profile. Since the purpose of the readout resonator is to absorb readout pulses when its coupled qubit is in the ground state, it should effectively behave similar to a notch filter. Two dimensional resonator spectroscopies are presented below of both the readout resonators in Pingu B. Additionally, a one dimensional multiplexed resonator spectroscopy is presented, which was conducted in parallel on both resonators simultaneously.

As we scan the frequency, the step size can be arbitrarily small. Hence, the only non-trivial choice is the readout pulse shape. Many shapes are valid, but the rectangular shape is the simplest. We choose a rectangular pulse which lasts for $3.5\,\mu$s.

We also must choose an amplitude for the readout pulse. We know that we must have enough power to distinguish between "absorbed" and "not absorbed". In the two dimensional sweeps presented, we sweep both the amplitude and the frequency.

The first resonator spectroscopy is shown in Figure 4.1 which is a concatenation of two datasets ($\texttt{tuid}_0$ and $\texttt{tuid}_1$). The data from $\texttt{tuid}_0$ was only averaged over 300 shots and gathered with $30\,\mu$s buffer time. The data from $\texttt{tuid}_1$ was gathered with $3\,\mu$s buffer time. This was done to make the experiments go a bit faster. A short buffer time is much less consequential in a resonator spectroscopy, because we do not have to wait for the qubit to go to the ground state. Two sweeps were conducted in order to get a higher resolution image, and to show that the `tqc` HDF parser can concatenate datasets with the same dimensions. The amplitude was swept from $0\,\text{mV}$ to $50\,\text{mV}$ with stepsize $2\,\text{mV}$. The frequency was swept from $6.199\,\text{GHz}$ to $6.229\,\text{GHz}$ with stepsize $0.1\,\text{MHz}$. Note that there are far more frequencies in the sweep than amplitudes. In order to easily view the image the pixels must therefore be made rectangular. Furthermore, Gaussian interpolation has been applied to the

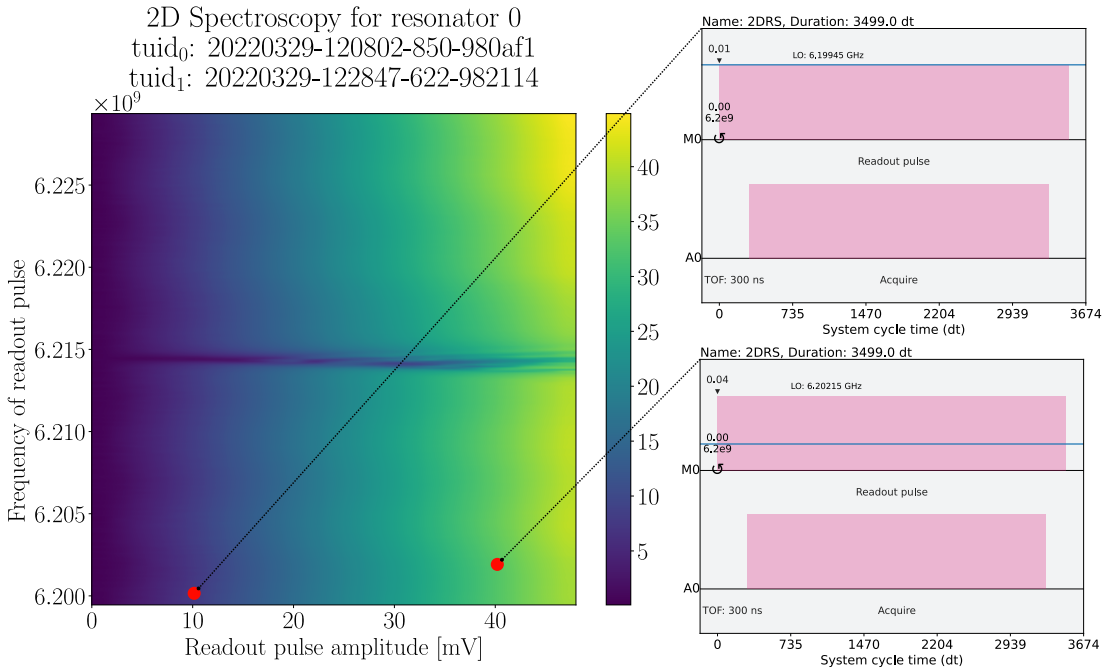


Figure 4.1: Left hand side: Amplitude of the measurement signal as a function of readout pulse amplitude and carrier frequency. Right hand side: Two OpenPulse schedules which generated two points in the plot to the left. The points have been marked out with red markers. The amplitude of the readout pulse in the lower schedule is larger, as indicated by the blue reference bar.

image.

The image is as expected because the amplitude of the measurement drops to a very small quantity at a very narrow frequency and is large elsewhere, proportional to the amplitude of the readout pulse, as indicated by the color bar. In Figure 4.1 we see what is called a resonator *punchout* at approximately 14 mV. The larger the amplitude of our readout pulse, the larger the separability of our qubit state will be, so it is important to maximize the amplitude of the readout pulse, before the punchout.

The same experiment was conducted for the other resonator, which is plotted in Figure 4.2. In this experiment the frequency was swept from 6.364 GHz to 6.394 GHz

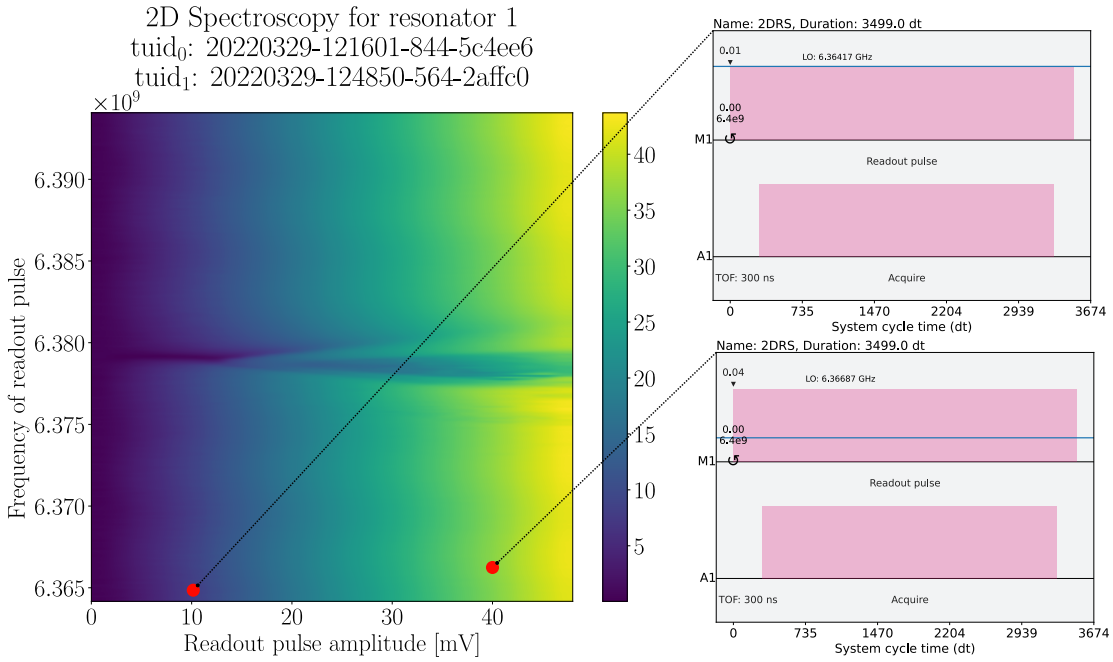


Figure 4.2: Left hand side: Amplitude of the measurement signal. Right hand side: Two OpenPulse schedules which generated two points in the plot to the left. The points have been marked out with red markers. The amplitude of the readout pulse in the lower schedule is larger, as indicated by the blue reference bar.

with step size 0.1 MHz. The amplitude step size was the same as the in Figure 4.1. Note that the image of this sweep clearly differs from the previous, which is to be expected since it is caused by a different physical component. In Figure 4.2 the punchout is more clear, and occurs around 14 mV as well.

It is a bit difficult to determine the characteristic frequency of these plots, because it is not that easy to determine the theoretical profile of a two dimensional resonator spectroscopy. We instead want to fit the one dimensional theoretical resonator profile against the data. Effectively, this amounts to looking at an amplitude cross section of these two figures.

We could simply extract a cross section from these figures, but instead a multiplexed one dimensional resonator spectroscopy was conducted, which was transmitted in

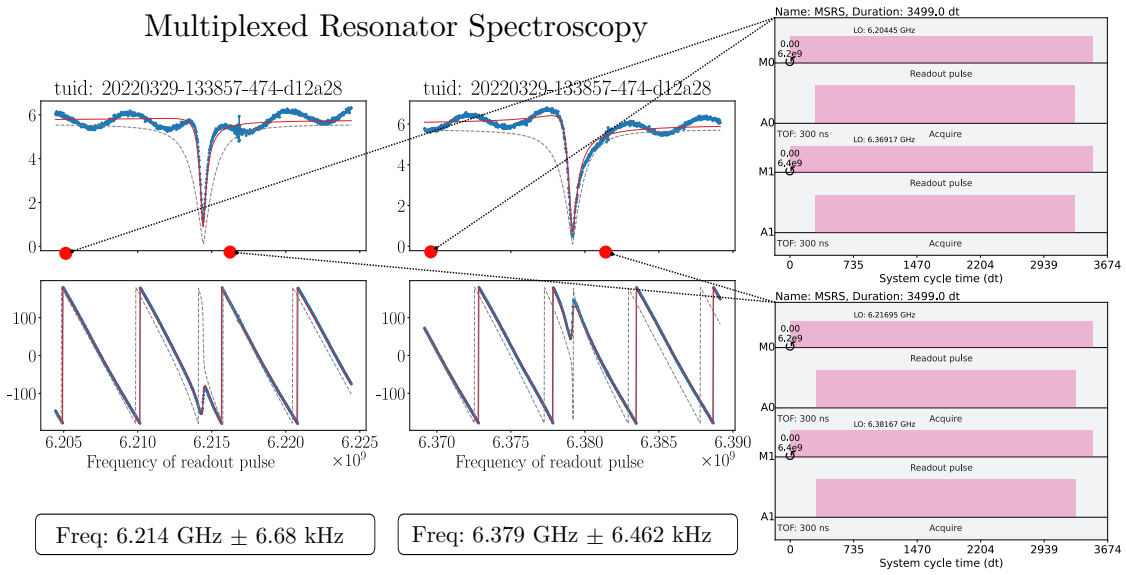


Figure 4.3: Left hand side: Amplitude and phase plots of the measurement signal. The frequency axis is shared and the phase units are in degrees, the troughs are found by fitting a theoretical resonator profile to the data. Right hand side: Two OpenPulse schedules which correspond to two frequencies each in the plots to the left. In each plot, there is one marker for each resonator. A buffer time of 30 μs was used.

parallel on both resonators simultaneously. This was done to show that `tqc` is capable of handling multiplexed schedules. The amplitudes of the readout pulses were halved so that each readout pulses both had amplitudes of 7 mV. By halving the amplitude of the pulses, we ensure that we keep the amplitude of the total signal below the punchout of both resonators. The sweep is plotted in Figure 4.3. The frequencies in the left plot were swept from 6.204 GHz to 6.224 GHz and the frequencies in the right plot from 6.369 GHz to 6.389 GHz, both with stepsize 0.025 MHz.

Note that all of the subplots have the same TUID because all were ran in the same experiment and were measured simultaneously. In the upper subplot of Figure 4.3 we see that the amplitude of the measurement is around 6 mV, which corresponds approximately to one pulse each being transmitted into each resonator.

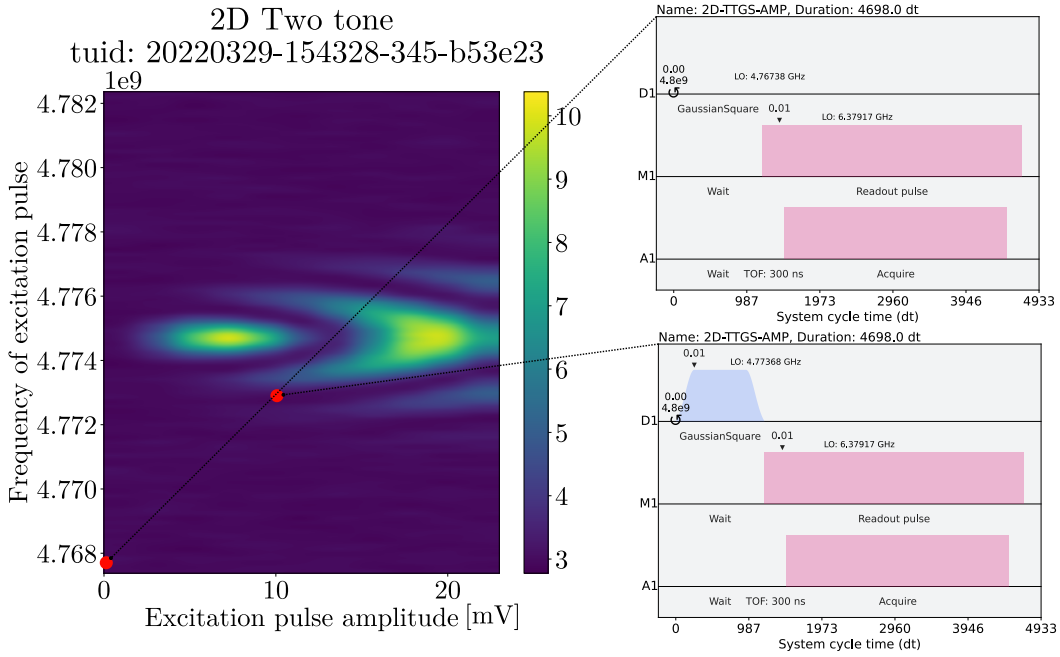


Figure 4.4: Left hand side: Amplitude of the measurement signal as a function of the excitation pulse amplitude and carrier frequency. Right hand side: Two OpenPulse schedules which generated two points in the plot to the left.

4.2 Two-tone Spectroscopy

The goal of the two-tone spectroscopy sweep is to verify that the qubit circuit is working, i.e. that we can excite a qubit in the ground state to some non-trivial superposition of $|0\rangle$ and $|1\rangle$. The etymology of the name "two-tone" stems from that we are sending two pulses; first we send a pulse at the characteristic frequency of the qubit circuit, and then we send a pulse at the characteristic frequency of the resonator. As described in Section 1.1 and Section 1.3, this procedure allows us to determine if the qubit became excited. We can determine that the qubit circuit is working if, when the frequency of the stimulus pulse is swept over the qubit channel, we see a large amplitude in the measurement signal at the $|0\rangle \rightarrow |1\rangle$ frequency. This means that the readout pulse was specifically not absorbed at this frequency.

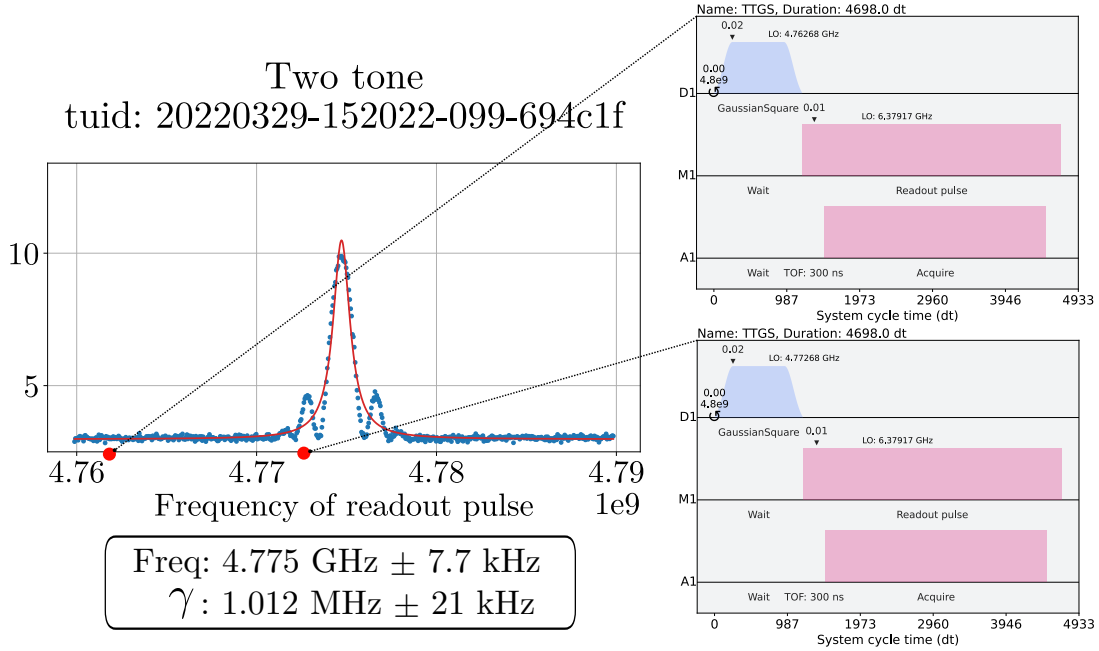


Figure 4.5: Left hand side: Amplitude of the measurement signal. The peak is found by fitting a Lorentzian distribution to the data. The parameter γ is the scale parameter of the Lorentzian. Right hand side: Two OpenPulse schedules which correspond to two frequencies in the plot to the left.

From the resonator spectroscopy we already have a good calibration of the readout pulse. We will keep using it for the remaining experiments. However now we are faced with a non-trivial choice of the stimulus pulse. We know that we want it to be short, but the exact duration is not important at the moment, because we just want to know if the qubit is working or not, so speed is not a priority. Lets choose a pulse of 1.2 μs width.

For this experiment we use a "GaussianSquare" function, which is just a rectangular pulse with a Gaussian smoothing instead of a discontinuity. The result is presented in Figure 4.4. The image is as expected, since we see that the readout pulse is fully reflected when the excitation pulse had amplitude 6.5 mV. The frequency was swept from 4.767 GHz to 4.782 GHz with stepsize 0.14 MHz. The amplitude was swept from 0.0 mV to 24.0 mV with stepsize 2.3 mV. The readout pulse was kept the same as

the one obtained from the resonator spectroscopy.

Similar to resonator spectroscopy, it is difficult to match the two dimensional sweep against a model, so we instead perform a one dimensional experiment corresponding to a cross section, which is what is shown in Figure 4.5. In this one dimensional case, the frequency of the excitation pulse was swept from 4.760 GHz to 4.790 GHz with stepsize 0.05 MHz. A peculiar feature of Figure 4.4 is that at around 12 mV the readout pulse is in fact absorbed, even at the qubit frequency. This is due to a phenomenon called a Rabi oscillation [35, p. 25], which brings us to the next experiment.

4.3 Rabi Oscillations

We can informally describe Rabi oscillations by returning to the Bloch sphere, from Section 1.1. Consider what would happen if we prepare a qubit to be in $|\psi\rangle = |0\rangle$ and drive it with so much power that the vector $|\psi\rangle$ goes up to $|1\rangle$ and then "overshoots" $|1\rangle$. It will then go to a "less excited" state, on the other side of the Bloch sphere. From a purely computational perspective, this is what we see in Figure 4.4. This is a phenomenon known as Rabi oscillations, which is the cyclic behaviour of a two-level quantum system in the presence of an oscillatory driving field. We will not cover the physics of this behaviour, and refer the reader instead to Krantz et. al. [35].

The Rabi oscillation experiment is a part of the single qubit characterisation experiments because the power which excites the qubit precisely to $|1\rangle$ (without overshoot) is of great computational interest. The power of a pulse depends on its shape, its duration and its amplitude. Conventionally, a Gaussian or a slightly modified Gaussian called a DRAG [15] pulse are used by virtue of optimality results for such shapes. We therefore use a Gaussian pulse shape for this experiment.

To determine the duration and the amplitude of the Gaussian which gives the excitation pulse the power that we need, we perform a two-dimensional sweep with these parameters as sweep parameters. The standard deviation of the Gaussian pulse was

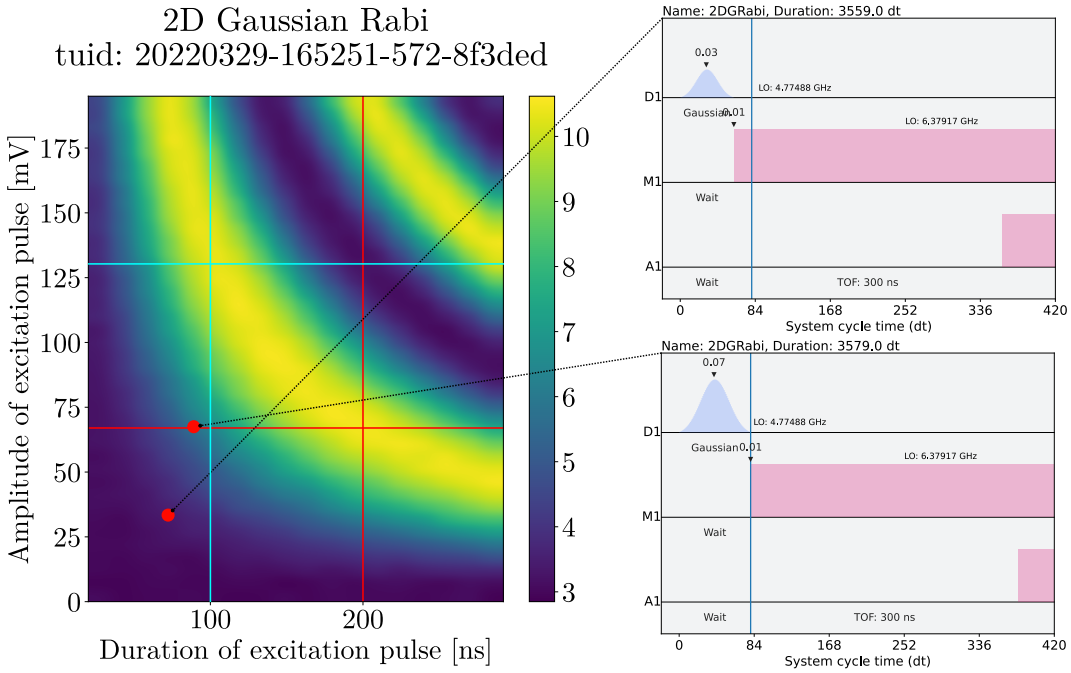


Figure 4.6: Left hand side: Amplitude of the measurement signal when both the duration and the amplitude of the excitation pulse were swept. The cyan and red axes indicate two possible parameterisations for an X-gate. Right hand side: Two OpenPulse schedules which generated two points in the plot to the left. The duration of the Gaussian pulse in the lower schedule is longer, as indicated by the fixed blue reference bar.

fixed to be a fifth of the pulse duration. The result of this sweep is plotted in Figure 4.6. The amplitude was swept from 0 mV to 200 mV with stepsize 5 mV and the duration was swept from 20 ns to 300 ns with stepsize 16 ns.

In Figure 4.6 any point along the first "front" of the plot on the left hand side constitutes a valid Gaussian excitation pulse (with σ as a fifth of the duration) because at these points we see a full reflection of the readout pulse.

We use this information to conduct a one dimensional Rabi oscillation experiment, so that we can precisely find the right amplitude needed. The one dimensional Rabi oscillation experiment is plotted in Figure 4.7. In this one dimensional sweep, the duration was fixed to 200 ns and the amplitude was swept from 0 mV to 300 mV

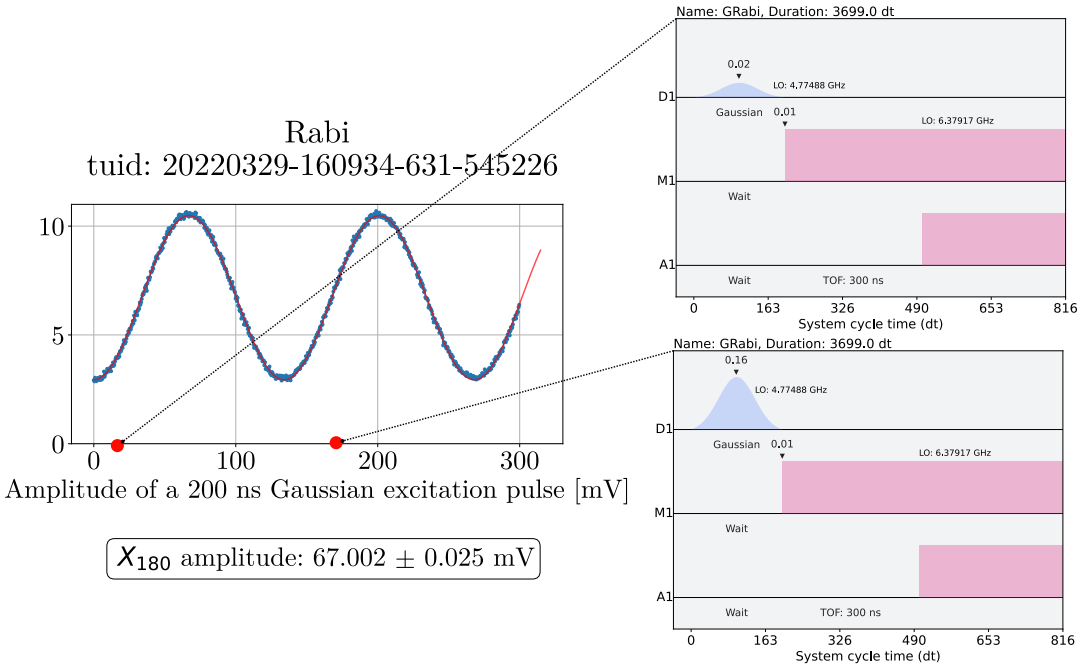


Figure 4.7: Left hand side: Rabi oscillation experiment where the amplitude was varied along the vertical red axis in Figure 4.6. We can find the $|0\rangle \rightarrow |1\rangle$ amplitude by fitting a sinusoidal function to the data. Right hand side: Two OpenPulse schedules which correspond to two amplitudes in the plot to the left.

with stepsize 0.45 mV. We could also use the second front for this parameterisation, but then the $|\psi\rangle$ state in the previous example would be going $1\frac{1}{2}$ times around the Bloch sphere, instead of just $\frac{1}{2}$ of a time. For this reason, the pulse required to go to $|1\rangle$ is called the X_{180} or often just the X pulse, since it corresponds to the quantum X-gate. Similarly, going halfway up the Bloch sphere to the equator is called an X_{90} pulse.

4.4 Relaxation time (T_1)

We know from Section 1.1 that a qubit will not stay in an excited state forever, and will decay down to $|0\rangle$. Since we can now create the $|1\rangle$ state with the X pulse, a

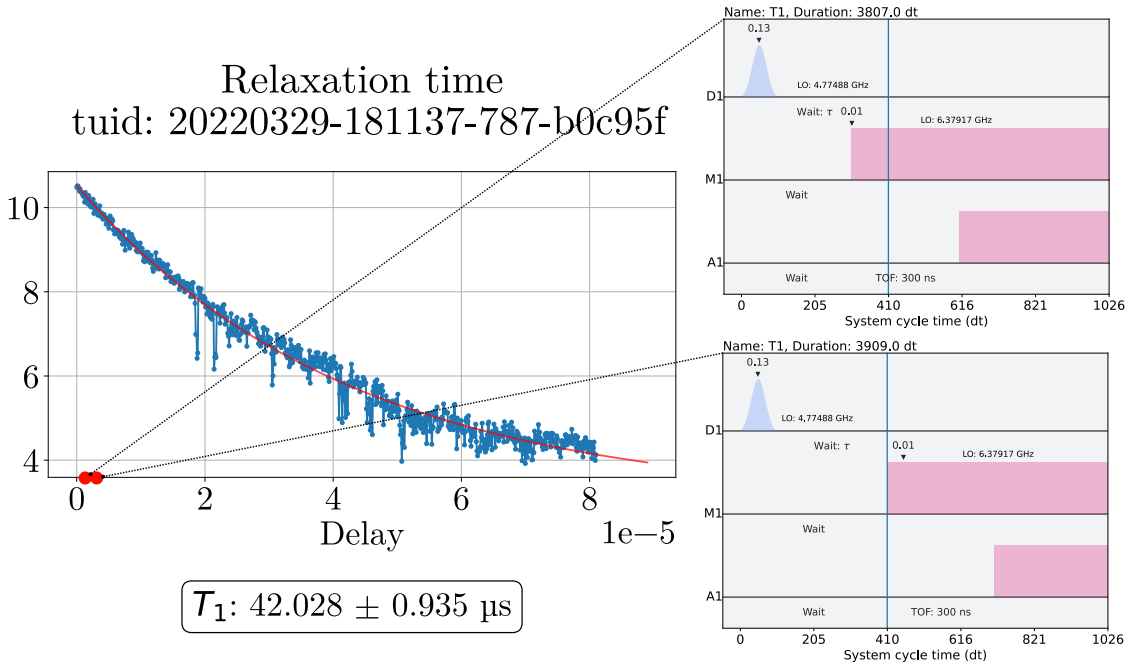


Figure 4.8: Left hand side: Amplitude of the measurement signal. The X_{180} pulse was defined as the pulse corresponding to the cyan axis in Figure 4.6. Right hand side: Two OpenPulse schedules which correspond to two delay times in the plot to the left. The measurement event occurs 102 ns later in the lower schedule than in the upper schedule, as indicated by the blue reference bar.

natural question to ask is: after an X pulse has been played, how long does it take for a qubit to go from $|1\rangle \to |0\rangle$? This is the question which the relaxation time (a.k.a. T_1) experiment answers by sweeping a delay parameter before measurement, after applying an X pulse. The results of such an experiment is plotted in Figure 4.8. The delay before measurement was swept from 4 ns to 81 μs with stepsize 102 ns. The T_1 time reported here is the coefficient in the expression $A\exp(\frac{-x}{T_1}) + C$. In other words, we find it by fitting a decaying exponential model to the data. The T_1 time is known as a *decoherence time*. There is another decoherence time called T_2 which is usually also measured. This experiment was attempted, but a mistake was noticed in the schedule implementing it at a very late stage of writing this thesis, so the T_2 experimental results have been omitted.

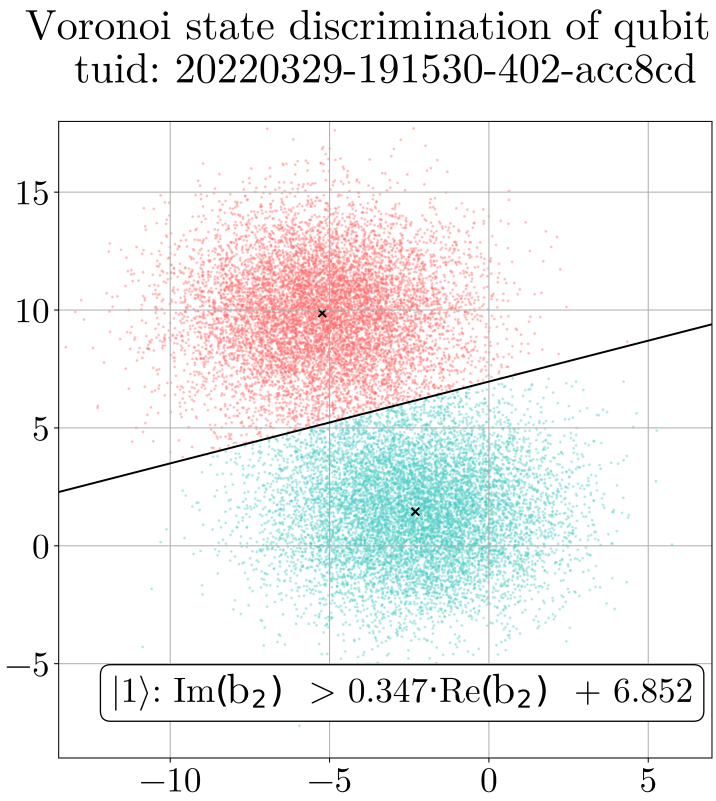


Figure 4.9: Qubit state discrimination between $|0\rangle$ (blue) and $|1\rangle$ (pink). The label b_2 refers to the measurement signal and each point is a single shot. In total there are 10k ground state shots and 10k excited state shots. The black crosses indicate cluster centers.

4.5 State Discrimination

The state discrimination experiment is the simplest single qubit experiment, but it depends on a good calibration of the readout pulse and the X pulse. In a state discrimination experiment, we simply want to test how separable the states $|0\rangle$ and $|1\rangle$ are. The experiment is conducted by first collecting single shot measurement data from when the qubit is in the ground state, and then collecting single shot measurement data from when the qubit is in the excited state and finally combining the two datasets. We then try to distinguish between $|0\rangle$ and $|1\rangle$ states in the merged

dataset using some algorithm.

When the measurement level is integrated, then this amounts to separating two clusters of points in the complex plane. There are many ways of identifying clusters of point processes. In this thesis a simple k-means clustering method was used. A separating line was then found by computing the Voronoi boundary of the cluster centers yielded from the k-means method. We can always do this for a general point cloud because the k-means clustering method results in a partitioning of the data space into Voronoi cells. This method was used because it can be used for arbitrary number of clusters, it does not depend on the location of the clusters, and moreover it is implemented easily using the Python library `SciPy` [42] which has fast implementations of both k-means clustering and Voronoi boundary computation. The discrimination can be seen in Figure 4.9. Note that the separation of the cluster centers is about 14 mV in modulus, which highlights the importance of selecting a good readout pulse. Using the discriminator reported in Figure 4.9 we find that $\mathbb{P}(|0\rangle \,|\text{no excitation pulse}) = 0.9334$ and $\mathbb{P}(|1\rangle \,|\text{excitation pulse}) = 0.9024$ for sample sizes of 10k shots each. It makes sense that the latter is less probable since a qubit will naturally go towards the ground state.

5

Discussion & Conclusion

In conclusion, we have presented a way to generate graphs from OpenPulse schedules. We have also showed how to parse these graphs as Quantify schedules which can consequently be compiled into executable programs by the Quantify platform. In so doing, we have implemented a suitable hardware abstraction layer of pulse scheduling which generalises upon the control flow of the software used in the instrument orchestration. The abstraction allows quantum computing experiments which rely on time dependent dynamics, such as characterisation of decoherence and crosstalk, dynamical decoupling, dynamically corrected gates, and gate parallelism to be specified unambiguously independent of control hardware. Another consequence of the abstraction is that any quantum circuit described with OpenQASM which is transpiled against a calibrated basis set of a Chalmers' quantum computer can now be executed on that quantum computer. The presented library is also implemented as a server, which allows offsite users to queue jobs to Chalmers' quantum computers. This capability was proven with a successful X gate test sent from the LUMI supercomputer in Finland. This is an important step towards building a software stack of a publicly accessible quantum computer due to the widespread use of OpenQASM.

A limitation of the thesis is that only certain single qubit experiments and one multiplexed experiment were conducted. However, with the library now in place, quantum computer experiments are easier to perform in large batches, and at larger scales with more qubits. An immediate extension of the thesis is to test the library with Ramsey oscillation experiments, dynamical decoupling experiments, and Quantum tomography experiments in the form of OpenPulse schedules.

Moreover, the hardware abstraction layer also serves as a good foundation for the automatic calibration of qubits. This is because automatic calibration routines can

now be described as feedback methods which read HDF data coming from `tqc` and in response en-queue OpenPulse schedules to `tqc` in the form of `QObjects`. Due to the hardware abstraction, this can all now be done without interfacing with the QBLOX control hardware. This even still applies if eventually the QBLOX control stack is replaced, given that a parsing method of `Experiment` graphs is written which complies with the new control stack's notion of a schedule.

Another extension of this thesis is to flesh out the HDF file which `tqc` returns, because it is a very barebones implementation with very rudimentary support for Quantify datasets. A more sophisticated HDF file would be needed for automatic calibration, for example. Furthermore, the state discriminator was hardcoded to suit the working qubit in Pingu B, and so a more generic state discrimination subroutine of `tqc` is also a necessary extension if multiple qubit and/or multiple measurement readouts are to be supported.

From a theoretical point of view, it would be interesting to consider how the category **Sched** can be used to describe a co-design theory [23] of quantum computer experiments. It is a construction which seems to be well suited to develop feasibility analyses of gate parallelism schedules in particular due to its order theoretic description. It could possibly also be used to formalise balancing problems of pulse schedules or resource theories of quantum computation. In other words, it opens a window to optimization routines of quantum computation in the time domain by serving as a flexible framework of scalable pulse scheduling.

Thank you for reading

Bibliography

1. Shannon, C. E. A mathematical theory of communication. *The Bell system technical journal* **27,** 379–423 (1948).
2. Hahn, E. L. Spin Echoes. *Physical Review* **80,** 580–594. doi:10.1103/physrev.80.580. https://doi.org/10.1103/physrev.80.580 (Nov. 1950).
3. Moyles, D. M. & Thompson, G. L. An Algorithm for Finding a Minimum Equivalent Graph of a Digraph. *Journal of the ACM* **16,** 455–460. doi:10.1145/321526.321534. https://doi.org/10.1145/321526.321534 (July 1969).
4. Aho, A. V., Garey, M. R. & Ullman, J. D. The Transitive Reduction of a Directed Graph. *SIAM Journal on Computing* **1,** 131–137. doi:10.1137/0201008. https://doi.org/10.1137/0201008 (June 1972).
5. Gabriel, P. Unzerlegbare Darstellungen I. *manuscripta mathematica* **6,** 71–103. doi:10.1007/bf01298413. https://doi.org/10.1007/bf01298413 (Mar. 1972).
6. Lathi, B. P. *Signal processing and linear systems* ISBN: 9780190651077 (Oxford University Press, New York, 1998).
7. Viola, L. & Lloyd, S. Dynamical suppression of decoherence in two-state quantum systems. doi:10.1103/PhysRevA.58.2733. eprint: arXiv:quant-ph/9803057 (1998).
8. Debnath, L. & Mikusinski, P. *Introduction to Hilbert spaces with applications* (Academic press, 2005).

9. Siddiqi, I. *et al.* Dispersive measurements of superconducting qubit coherence with a fast latching readout. *Physical Review B* **73.** doi:10.1103/physrevb.73.054510. https://doi.org/10.1103/physrevb.73.054510 (Feb. 2006).

10. Abramsky, S. & Coecke, B. Categorical quantum mechanics. eprint: arXiv:0808.1023 (2008).

11. Dirr, G. & Helmke, U. Lie theory for quantum control. *GAMM-Mitteilungen* **31,** 59–93 (2008).

12. Dong, D. & Petersen, I. R. Quantum control theory and applications: A survey. doi:10.1049/iet-cta.2009.0508. eprint: arXiv:0910.2350 (2009).

13. Durbin, J. *Modern algebra : an introduction* ISBN: 9780470384435 (Wiley, Hoboken, NJ, 2009).

14. Khodjasteh, K. & Viola, L. Dynamically error-corrected gates for universal quantum computation. *Physical review letters* **102,** 080501 (2009).

15. Motzoi, F., Gambetta, J. M., Rebentrost, P. & Wilhelm, F. K. Simple Pulses for Elimination of Leakage in Weakly Nonlinear Qubits. *Physical Review Letters* **103.** doi:10.1103/physrevlett.103.110501. https://doi.org/10.1103/physrevlett.103.110501 (Sept. 2009).

16. Narici, L. & Beckenstein, E. *Topological Vector Spaces* doi:10.1201/9781584888673. https://doi.org/10.1201/9781584888673 (Chapman and Hall/CRC, July 2010).

17. Nielsen, M. *Quantum computation and quantum information* ISBN: 9781107002173 (Cambridge University Press, Cambridge New York, 2010).

18. Gambetta, J. M. *et al.* Characterization of addressability by simultaneous randomized benchmarking. *Physical review letters* **109,** 240504 (2012).

19. Geerlings, K. *et al.* Demonstrating a Driven Reset Protocol for a Superconducting Qubit. *Physical Review Letters* **110.** doi:10.1103/physrevlett.110.120501. https://doi.org/10.1103/physrevlett.110.120501 (Mar. 2013).

20. Schutjens, R., Dagga, F. A., Egger, D. J. & Wilhelm, F. K. Single-qubit gates in frequency-crowded transmon systems. *Physical Review A* **88.** doi:10.1103/physreva.88.052330. https://doi.org/10.1103/physreva.88.052330 (Nov. 2013).

21. Clough, J. R., Gollings, J., Loach, T. V. & Evans, T. S. Transitive reduction of citation networks. *Journal of Complex Networks* **3,** 189–203. doi:10.1093/comnet/cnu039. https://doi.org/10.1093/comnet/cnu039 (Sept. 2014).

22. Jeffrey, E. *et al.* Fast accurate state measurement with superconducting qubits. *Physical review letters* **112,** 190504 (2014).

23. Censi, A. A mathematical theory of co-design. *arXiv preprint arXiv:1512.08055* (2015).

24. Coecke, B., Heunen, C. & Kissinger, A. Categories of quantum and classical channels. *Quantum Information Processing* **15,** 5179–5209 (2016).

25. Johansson, R. in *Grundläggande Datorteknik* (Studentlitteratur AB, 2016).

26. Barnes, E., Arenz, C., Pitchford, A. & Economou, S. E. Fast microwave-driven three-qubit gates for cavity-coupled superconducting qubits. *Physical Review B* **96.** doi:10.1103/physrevb.96.024504. https://doi.org/10.1103/physrevb.96.024504 (July 2017).

27. Deffner, S. & Campbell, S. Quantum speed limits: from Heisenberg's uncertainty principle to optimal quantum control. **50,** 453001. doi:10.1088/1751-8121/aa86c6. https://doi.org/10.1088/1751-8121/aa86c6 (Oct. 2017).

28. Hoyer, S. & Hamman, J. xarray: N-D labeled arrays and datasets in Python. *Journal of Open Research Software* **5.** doi:10.5334/jors.148. https://doi.org/10.5334/jors.148 (2017).

29. Walter, T. *et al.* Rapid High-Fidelity Single-Shot Dispersive Readout of Superconducting Qubits. *Physical Review Applied* **7.** doi:10.1103/physrevapplied.7.054020. https://doi.org/10.1103/physrevapplied.7.054020 (May 2017).

30. Fu, X. *et al.* eQASM: An Executable Quantum Instruction Set Architecture. *CoRR* **abs/1808.02449.** arXiv: 1808.02449. http://arxiv.org/abs/1808.02449 (2018).

31. Magnard, P. *et al.* Fast and Unconditional All-Microwave Reset of a Superconducting Qubit. *Physical Review Letters* **121.** doi:10.1103/physrevlett.121.060502. https://doi.org/10.1103/physrevlett.121.060502 (Aug. 2018).

32. McKay, D. C. *et al.* Qiskit backend specifications for openqasm and openpulse experiments. *arXiv preprint arXiv:1809.03452* (2018).

33. Ferrini, G., Kockum, A. F., Garcia-Alvarez, L. & Vikstål, P. Advanced Quantum Algorithms (2019).

34. Heunen, C. & Vicary, J. *Categories for Quantum Theory* doi:10.1093/oso/9780198739623.001.0001. https://doi.org/10.1093/oso/9780198739623.001.0001 (Oxford University Press, Nov. 2019).

35. Krantz, P. *et al.* A quantum engineer's guide to superconducting qubits. *Applied Physics Reviews* **6,** 021318 (2019).

36. Milewski, B. *Category theory for programmers* (Bartosz Milewski, 2019).

37. Rol, M. *et al. PycQED_py3* version v0.2. Dec. 2019. doi:10.5281/zenodo.3574563. https://doi.org/10.5281/zenodo.3574563.

38. Alexander, T. *et al.* Qiskit pulse: programming quantum computers through the cloud with pulses. *Quantum Science and Technology* **5,** 044006 (2020).

39. Andréasson, N., Evgrafov, A. & Patriksson, M. *An introduction to continuous optimization: foundations and fundamental algorithms* (Courier Dover Publications, 2020).

40. Harris, C. R. *et al.* Array programming with NumPy. *Nature* **585,** 357–362. doi:10.1038/s41586-020-2649-2. https://doi.org/10.1038/s41586-020-2649-2 (Sept. 2020).

41. Murali, P., Mckay, D. C., Martonosi, M. & Javadi-Abhari, A. *Software Mitigation of Crosstalk on Noisy Intermediate-Scale Quantum Computers* in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (ACM, Mar. 2020). doi:10.1145/3373376.3378477. https://doi.org/10.1145/3373376.3378477.

42. Virtanen, P. *et al.* SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* **17,** 261–272. doi:10.1038/s41592-019-0686-2 (2020).

43. ANIS, M. S. *et al. Qiskit: An Open-source Framework for Quantum Computing* 2021. doi:10.5281/zenodo.2573505.

44. Collette, A. *et al. h5py/h5py: 3.2.1* version 3.2.1. Mar. 2021. doi:10.5281/zenodo.4584676. https://doi.org/10.5281/zenodo.4584676.

45. Cross, A. W. *et al.* OpenQASM 3: A broader and deeper quantum assembly language. *arXiv preprint arXiv:2104.14722* (2021).

46. Das, P., Tannu, S., Dangwal, S. & Qureshi, M. ADAPT: Mitigating Idling Errors in Qubits via Adaptive Dynamical Decoupling. doi:10.1145/3466752.3480059. eprint: arXiv:2109.05309 (2021).

47. Jurcevic, P. *et al.* Demonstration of quantum volume 64 on a superconducting quantum computing system. *Quantum Science and Technology* **6,** 025020. doi:10.1088/2058-9565/abe519. https://doi.org/10.1088/2058-9565/abe519 (Mar. 2021).

48. Nielsen, J. H. *et al. QCoDeS/Qcodes: 0.33.0 - March 2022 (2022-03-09)* version v0.33.0. Mar. 2022. doi:10.5281/zenodo.6341457. https://doi.org/10.5281/zenodo.6341457.

49. Branten, R. *Quantify* https://www.qblox.com/quantify.

Appendix

A. Required Imports

All examples in Appendix B. can be executed provided that the following Python modules are imported.

Listing 1: Sufficient imports to execute OpenPulse schedules on the Pingu backend.

```
from qiskit.pulse import ScheduleBlock, Schedule
from qiskit.pulse.instructions import Play, Delay, SetFrequency, SetPhase, ShiftFrequency,
    ShiftPhase, Acquire
from qiskit.pulse.library import Gaussian, Constant, GaussianSquare
from qiskit.circuit import Parameter
from qiskit.pulse.channels import MemorySlot
from qiskit.providers.tergite import Tergite
from qiskit.pulse.channels import DriveChannel, MeasureChannel, AcquireChannel
import numpy as np
```

A Qiskit backend object associated to the Pingu backend can be constructed with Tergite using the following code.

Listing 2: How to initialise a Pingu backend as a compilation target.

```
chalmers = Tergite.get_provider()
backend = chalmers.get_backend("Pingu")
backend.set_options(shots = 2000) # set number of measurement shots to 2000
```

This compilation target is assumed for all examples in Appendix B.. It is also convenient to define the following variables, to make the code more readable.

Listing 3: Miscellaneous helpful definitions.

```
μs, ns, MHz, GHz, mV  = 1e-6, 1e-9, 1e6, 1e9, 1e-3
meas_lo_freq = [6.214*GHz, 6.379*GHz]
qubit_lo_freq = [None, 4.775*GHz]
```

B. OpenPulse Examples

Listing 4: Two-dimensional resonator spectroscopy on left resonator.

```
qubit = 0
spec_pulse_width = 3.5*μs # readout pulse duration
freq = Parameter("freq") # sweep parameter: readout frequency
ampl = Parameter("ampl") # sweep parameter: amplitude

# frequency sweep
center = meas_lo_freq[qubit] # scan around last known location
span   = 30 * MHz
step   = .1 * MHz
start  = center - span/2
end    = center + span/2
frequencies = np.arange(start, end, step)
print(f"Sweeping frequency from {start/GHz}GHz to {end/GHz}GHz around {center/GHz}GHz with
    stepsize {step/MHz} MHz.")

# amplitude sweep
start  = 0*mV
end    = 14*mV
step   = 2*mV
amplitudes = np.arange(start, end, step)
print(f"Sweeping amplitude from {start/mV}mV to {end/mV}mV with stepsize {step/mV} mV.")

# create parametric schedule object
sched = ScheduleBlock(f"2DRS")
sched += SetFrequency(freq, backend.measure_channel(qubit))
sched += Play(Constant(int(spec_pulse_width/ns), amp = ampl, name = "Readout pulse"), backend.
    measure_channel(qubit))
sched += Delay(300, backend.acquire_channel(qubit), "TOF: 300 ns")
sched += Acquire(int(3*μs/ns), backend.acquire_channel(qubit), MemorySlot(qubit), name = "
    Integration window")

# generate sweep
sweep = [
    sched.assign_parameters({freq : f, ampl : a}, inplace = False)
    for f in frequencies
    for a in amplitudes
]
print("Schedule count:", len(sweep))

# transmit qobj
backend.run(sweep, meas_level = 1, meas_return = "avg", qobj_header={"frequencies" : frequencies,"
    amplitudes"  : amplitudes})
```

Listing 5: Multiplexed one dimensional resonator spectroscopy.

```
# parameterise readout pulse
spec_pulse_width = 3.5*μs
spec_pulse_amp = 14*mV /2

# create parametric schedule object
def append_spec(sched, qubit:int):
    freq_qubit = Parameter(f"freq{qubit}")
    center = meas_lo_freq[qubit]
    span   = 20 * MHz
    step   = .025 * MHz
    start  = center - span/2
    end    = center + span/2
    frequencies = np.arange(start, end, step)
    print(f"Sweeping from {start/GHz} GHz to {end/GHz} GHz around {center/GHz} GHz with stepsize {
    step/MHz} MHz.")

    sched += SetFrequency(freq_qubit, backend.measure_channel(qubit))
    sched += Play(Constant(int(spec_pulse_width/ns), amp = spec_pulse_amp, name = "Readout pulse")
    , backend.measure_channel(qubit))
    sched += Delay(300, backend.acquire_channel(qubit), f"TOF: {300} ns")
    sched += Acquire(int(3*μs/ns), backend.acquire_channel(qubit), MemorySlot(qubit))

    return frequencies, freq_qubit

sched = ScheduleBlock(f"MSRS")

frequencies0, freq_qubit0 = append_spec(sched, 0)
frequencies1, freq_qubit1 = append_spec(sched, 1)

# generate sweep
sweep = [ sched.assign_parameters({
    freq_qubit0 : f,
    freq_qubit1 : g,
}, inplace = False) for f,g in zip(frequencies0, frequencies1) ]
print("Schedule count:", len(sweep))

# send qobj
backend.run(
    sweep,
    meas_level=1,
    meas_return='avg',
    qobj_header={
        "frequencies0" : frequencies0,
        "frequencies1" : frequencies1,
    }
)
```

Listing 6: Two dimensional two-tone spectroscopy on right qubit.

```
# in subsequent listings we let the backend infer meas_lo_freq
# specify readout pulse and excitation pulse duration
qubit = 1
stim_pulse_width = 1.2*μs
spec_pulse_width = 3.5*μs
spec_pulse_amp   = 14*mV
freq = Parameter("freq")
ampl = Parameter("ampl")

# frequency sweep
center = qubit_lo_freq[qubit]
span   = 15 * MHz
step   = .14 * MHz
start  = center - span/2
end    = center + span/2
frequencies = np.arange(start, end, step)
print(f"Sweeping frequency from {start/GHz}GHz to {end/GHz}GHz around {center/GHz}GHz with
    stepsize {step/MHz} MHz.")

# amplitude sweep
start  = 0*mV
end    = 24*mV
step   = 2.3*mV
amplitudes = np.arange(start, end, step)
print(f"Sweeping amplitude from {start/mV}mV to {end/mV}mV with stepsize {step/mV} mV.")

# create parametric schedule object
sched = ScheduleBlock("2D-TTGS-AMP")
sched += SetFrequency(freq, backend.drive_channel(qubit))
sched += Play(GaussianSquare(
    duration = int(stim_pulse_width/ns),
    amp = ampl,
    sigma = 120,
    width = int(stim_pulse_width/ns)-500
),backend.drive_channel(qubit))
sched += Delay(int(stim_pulse_width/ns), backend.measure_channel(qubit), name = "Wait")
sched += Delay(int(stim_pulse_width/ns), backend.acquire_channel(qubit), name = "Wait")
sched += Play(Constant(int(spec_pulse_width/ns), amp = spec_pulse_amp, name = "Readout pulse"),
    backend.measure_channel(qubit))
sched += Delay(300, backend.acquire_channel(qubit), "TOF: 300 ns")
sched += Acquire(int(3*μs/ns), backend.acquire_channel(qubit), MemorySlot(qubit))

# generate sweep
sweep = [
    sched.assign_parameters({freq : f, ampl : a}, inplace = False)
    for f in frequencies
    for a in amplitudes
]
```

```
print("Schedule count:", len(sweep))

# send qobj
backend.run(
    sweep,
    meas_level = 1,
    meas_return = "avg",
    qobj_header={
        "frequencies" : frequencies,
        "amplitudes"  : amplitudes,
    }
)
```

Listing 7: One dimensional two-tone spectroscopy on right qubit.

```
# parameterise the readout pulse and the excitation pulse
qubit = 1
stim_pulse_width = 1.2*μs
stim_pulse_amp = 21*mV
spec_pulse_width = 3.5*μs
spec_pulse_amp = 14*mV
freq_qubit = Parameter("freq")

# frequency sweep
center = qubit_lo_freq[qubit]
span   = 30 * MHz
step   = .05 * MHz
start  = center - span/2
end    = center + span/2
frequencies = np.arange(start, end, step)
print(f"Sweeping from {start/GHz} GHz to {end/GHz} GHz around {center/GHz} GHz with stepsize {step
    /MHz} MHz.")

# create parametric schedule object
sched = ScheduleBlock("TTGS")
sched += SetFrequency(freq_qubit, backend.drive_channel(qubit))
sched += Play(GaussianSquare(
    duration = int(stim_pulse_width/ns),
    amp = stim_pulse_amp,
    sigma = 120,
    width = int(stim_pulse_width/ns)-500
),backend.drive_channel(qubit))
sched += Delay(int(stim_pulse_width/ns), backend.measure_channel(qubit), name = "Wait")
sched += Delay(int(stim_pulse_width/ns), backend.acquire_channel(qubit), name = "Wait")
sched += Play(Constant(int(spec_pulse_width/ns), amp = spec_pulse_amp, name = "Readout pulse"),
    backend.measure_channel(qubit))
sched += Delay(300, backend.acquire_channel(qubit), "TOF: 300 ns")
sched += Acquire(int(3*μs/ns), backend.acquire_channel(qubit), MemorySlot(qubit))

# generate sweep
```

```
sweep = [ sched.assign_parameters({
    freq_qubit : f
}, inplace = False) for f in frequencies ]

# send qobj
backend.run(
    sweep,
    meas_level=1,
    meas_return='avg',
    qobj_header={"frequencies" : frequencies}
)
```

Listing 8: Two dimensional Rabi oscillation experiment.

```
# parameterise
qubit = 1
spec_pulse_width = 3.5*μs
spec_pulse_amp = 14*mV
stim_dur = Parameter("stim_dur")
ampl = Parameter("ampl")

# amplitude sweep
start  = 0*mV
end    = 200*mV
step   = 5*mV
amplitudes = np.arange(start, end, step)
print(f"Sweeping from {start/mV} mV to {end/mV} mV with stepsize {step/mV} mV.")

# duration sweep
start  = 60
end    = 120
step   = 4
durations = np.arange(start, end, step)
print(f"Sweeping from {start} ns to {end} ns with stepsize {step} ns.")

# create parametric schedule object
sched = ScheduleBlock(f"2DGRabi")
sched += Play(
    Gaussian(
        duration = stim_dur,
        amp = ampl,
        sigma = stim_dur/5
    ),
    backend.drive_channel(qubit),
    name = "Rabi pulse"
)
sched += Delay(stim_dur, backend.measure_channel(qubit), name = "Wait")
sched += Delay(stim_dur, backend.acquire_channel(qubit), name = "Wait")
sched += Play(Constant(int(spec_pulse_width/ns), amp = spec_pulse_amp), backend.measure_channel(
    qubit))
```

```
sched += Delay(300, backend.acquire_channel(qubit), "TOF: 300 ns")
sched += Acquire(int(3*μs/ns), backend.acquire_channel(qubit), MemorySlot(qubit))

# generate sweep
sweep = [ sched.assign_parameters({
    ampl : a,
    stim_dur : d
}, inplace = False) for a in amplitudes for d in durations ]

# send qobj
backend.run(
    sweep,
    meas_level=1,
    meas_return='avg',
    qobj_header={
        "durations" : durations,
        "amplitudes"  : amplitudes
    }
)
```

Listing 9: One dimensional Rabi oscillation experiment.

```
# parameterise
qubit = 1
stim_pulse_width = 200*ns
stim_pulse_sigma = stim_pulse_width/(5*ns)
spec_pulse_width = 3.5*μs
spec_pulse_amp = 14*mV
ampl = Parameter("ampl")

# amplitude sweep
start  = 0
end    = 300*mV
step   = 0.45*mV
amplitudes = np.arange(start, end, step)
print(f"Sweeping from {start/mV} mV to {end/mV} mV with stepsize {step/mV} mV.")

# create parametric schedule object
sched = ScheduleBlock(f"GRabi")
sched += Play(
    Gaussian(
        int(stim_pulse_width/ns),
        amp = ampl,
        sigma = stim_pulse_sigma
    ),
    backend.drive_channel(qubit),
    name = "Rabi pulse"
)
sched += Delay(int(stim_pulse_width/ns), backend.measure_channel(qubit), name = "Wait")
sched += Delay(int(stim_pulse_width/ns), backend.acquire_channel(qubit), name = "Wait")
```

```
sched += Play(Constant(int(spec_pulse_width/ns), amp = spec_pulse_amp, name = "Readout pulse"),
    backend.measure_channel(qubit))
sched += Delay(300, backend.acquire_channel(qubit), "TOF: 300 ns")
sched += Acquire(int(3*μs/ns), backend.acquire_channel(qubit), MemorySlot(qubit))

# generate sweep
sweep = [ sched.assign_parameters({
    ampl : a
}, inplace = False) for a in amplitudes ]

# send qobj
backend.run(
    sweep,
    meas_level=1,
    meas_return='avg',
    qobj_header={
        "amplitudes"  : amplitudes
    }
)
```

Listing 10: Relaxation time experiment.

```
# parameterise
qubit = 1
spec_pulse_width = 3.5*μs
spec_pulse_amp = 14*mV
tau = Parameter("t1 decay time")
def myX():
    return Play(
        Gaussian(
            100,
            amp = 130*mV,
            sigma = 20,
            name = " "
        ),
        backend.drive_channel(qubit),
        name = " "
    )

# duration sweep
start  = 4
end    = 9000*3*3
step   = 51*2
durations = np.arange(start, end, step)
print(f"Sweeping from {start} ns to {end} ns with stepsize {step} ns. ({len(durations)} durations)
    ")

# create parametric schedule object
sched = ScheduleBlock(f"T1")
sched += myX()
```

```
sched += Delay(tau, backend.drive_channel(qubit), name = r"Wait: $\tau$")
sched += Delay(100 + tau, backend.measure_channel(qubit), name = "Wait")
sched += Delay(100 + tau, backend.acquire_channel(qubit), name = "Wait")
sched += Play(Constant(int(spec_pulse_width/ns), amp = spec_pulse_amp), backend.measure_channel(
    qubit))
sched += Delay(300, backend.acquire_channel(qubit), "TOF: 300 ns")
sched += Acquire(int(3*μs/ns), backend.acquire_channel(qubit), MemorySlot(qubit))

# generate sweep
sweep = [ sched.assign_parameters({
    tau : d
}, inplace = False) for d in durations ]

# send qobj
backend.run(
    sweep,
    meas_level=1,
    meas_return='avg',
    qobj_header={"durations" : durations}
)
```