CHALMERS
UNIVERSITY OF TECHNOLOGY

Developing Virtual Instruments for Control and Automation of a Quantum Processor via Specialised FPGA Hardware

Master's thesis in Computer Science - Algorithms, Languages and Logic

JOHAN BLOMBERG
GUSTAV GRÄNNSJÖ

Department of Microtechnology and Nanoscience
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2020

Master's thesis

Developing Virtual Instruments for Control and Automation of a Quantum Processor via Specialised FPGA Hardware

JOHAN BLOMBERG

GUSTAV GRÄNNSJÖ

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Microtechnology and Nanoscience
Quantum Technology Laboratory
Chalmers University of Technology
Gothenburg, Sweden 2020

Developing Virtual Instruments for Control and Automation of a Quantum Processor via Specialised FPGA Hardware
JOHAN BLOMBERG, GUSTAV GRÄNNSJÖ

Supervisor and examiner: Simone Gasparinetti, Department of Microtechnology and Nanoscience

Master's Thesis
Department of Microtechnology and Nanoscience
Quantum Technology Laboratory
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2020

Developing Virtual Instruments for Control and Automation of a Quantum Processor via Specialised FPGA Hardware

JOHAN BLOMBERG, GUSTAV GRÄNNSJÖ
Department of Microtechnology and Nanoscience
Chalmers University of Technology

Abstract

Superconducting quantum bits (qubits) are emerging as a leading platform within the field of Quantum Computing. However, current methods for control and readout of superconducting qubits are limited in how pulse sequences can be defined. One emerging solution for flexible pulse sequence generation is the Field-Programmable Gate Array-based Vivace platform. Vivace allows for control pulses to be defined not as a long sequence of points, but as reusable templates that are output at specific times within the sequence. In addition, these templates can be mixed with arbitrary carrier waves through a digital oscillator. This thesis describes the development of software known as *virtual instruments*, used to control Vivace.

These instruments provide a user interface for assembling complex sequences of waveforms and outputting them via Vivace, as well as reading qubit output. The instruments include features such as parameter sweeps of arbitrary numbers of pulses, interleaved averaging, copying of pulses onto multiple ports and quadrature amplitude demodulation. As the instruments are built on top of the Python-based *Labber* instrument control platform, they can be linked together and coordinated with all other Labber instruments. We use the instruments to successfully perform several types of qubit characterisation measurements such as Rabi measurements and Ramsey interferometry, among others. In addition, we write a script which automates the sequential execution of several such measurements, allowing for convenient qubit characterisation.

Keywords: Python, signal processing, quantum computing, superconducting qubit, Labber, instrument control

Acknowledgements

We would first and foremost like to thank our supervisor Simone along with our assistant supervisors Marina and Christian. You've helped out every step of the way, whether it has been through feature requests for our software, providing swift feedback on our work, or teaching us concepts from fields new to us.

We would also like to extend our gratitude to Mats, Riccardo and David from the Vivace team for all their work, without which this project would not even have existed. You have always taken the time for us and our questions, and have kept us in the loop in regards to the latest developments of your API.

Additionally, Philip was a great help with understanding how Labber works, and why (or why not).

Lastly, we would like to thank the head of division, Per Delsing, for initiating the collaboration between the Quantum Technology lab at Chalmers and the Vivace team at KTH, as well as his continuous support of the project.

Johan Blomberg and Gustav Gränsjö
Gothenburg, June 2020

Contents

Terms and Abbreviations

Term	Written out	Short explanation
API	Application Programmer's Interface	The parts of a program that are made accessible to third parties for the purposes of extending or otherwise utilising said program. Can be thought of as entry points into a program's black box.
	Classical computing	Computation done with traditional computer architecture, i.e. does not employ the use of quantum mechanical effects such as superpositions, entanglement etc.
FPGA	Field-Programmable Gate Array	A kind of processing unit that allows for its logic gates to be reprogrammed, which enables a variety of specialised use cases.
GUI	Graphical User Interface	Graphical representation of a program, which the user can interact with through buttons, input fields etc.
LUT	Look-Up Table	A data structure in which previously prepared values are stored for quick access.
LO	Local oscillator	Hardware capable of generating a high frequency signal. Used for mixing with signals of lower frequencies.
	Microwave	A wave with a frequency from one to a few hundred GHz. In this thesis, the frequency band of interest is 4-8 GHz.
QPU	Quantum Processing Unit	Processor using quantum mechanics for information manipulation (contains one or more qubits).
Qubit	Quantum bit	Representation of a quantum two-level system. Used in quantum computing similar to how bits are used in classical computing.

1

Introduction

This introductory chapter is meant to give an overview of the thesis project, by giving answers to the *why*, *what* and *how*. We begin with the background to the project, a high level view of the current state of quantum computing and *why* there is a desire for more efficient scheduling of sending electrical pulses to a QPU. We also present the project's goals and limitations to clarify exactly *what* will and will not be done. Lastly, we outline the development process of the project in order to also show *how* the project is planned to proceed.

1.1 Background

In recent decades, significant advances have been made within the field of quantum computing. Utilising quantum mechanics to surpass the computational efficiency of classical computers has gone from hypothetical concept [1–3] to reality [4, 5]. An ideal quantum computer is theoretically able to compute algorithms with as much as exponential increased efficiency compared to that of classical computers. Quantum computation has already had an impact within cryptography, where for example the protocol BB84 has been developed which is proven to be unconditionally secure [6]. There are currently commercially available security products based on that protocol, such as those offered by MagiQ [1], ID Quantique [2] or Quintessence Labs [3].

But while some impressive results have been achieved by quantum computers, there is still a long way to go before reaching the goal of large-scale quantum computation; today's quantum computers are not currently at a level where they can solve most or even many problems that we know of. In addition to requiring novel hardware and software solutions to construct and control a quantum computer, developing efficient quantum algorithms even for known problems is notoriously difficult [7].

1.1.1 Quantum computing and superconducting qubits

What sets a quantum computer apart from a classical one? The most fundamental part of a quantum computer is the qubit, analogous to the bit of classical computing. Qubits are two-level systems, which means that just like classical bits they have two distinct states of being, often denoted $|0\rangle$ and $|1\rangle$. But unlike bits, instead of only existing in one

[1] `https://www.magiqtech.com/`
[2] `https://www.idquantique.com/`
[3] `https://www.quintessencelabs.com/`

state or the other at any given time, they can exist in a superposition of their two states. This means that each state has a probability associated with it, which corresponds to the chance to receive that state as a result when measuring the qubit. One very powerful aspect of superpositions is the possibility for n qubits to provide parallel access to all 2^n combinations of individual states of the qubits. This, along with qubit interactions such as interference and entanglement provide the means for a quantum computer to outperform classical algorithms.[4]

In theory, there are a variety of ways in which a quantum computer can be constructed. The key difference between designs is how one implements the qubit. Several designs involve the trapping and manipulation of individual atoms or subatomic particles, due to quantum phenomena occurring at these scales [10, 11], [12, Sec. 3.1]. One wholly different approach is that of the *superconducting qubit*. These are special electric circuits that are built to make quantum phenomena manifest when in a state of superconductivity, achieved by cooling the circuit to millikelvin temperatures [13–15]. While these so-called "artificial atoms" fall short of their natural siblings in some areas such as noise sensitivity and decoherence time (the time it takes until a state superposition has decayed such that all known information about the state has been rendered useless) [11], superconducting qubits make up for this in other ways. The circuitry that forms a superconducting qubit is comparatively easy to couple to other such circuits, simplifying entanglement of qubit states and enabling multi-qubit superpositions [13, 16, 17]. Being circuits, they are also fairly easy to manufacture using established techniques [11]. In addition, the artificial nature of superconducting qubits gives quantum engineers greater freedom to construct qubits that work well with the rest of their systems, rather than having to build their systems around the quirks of natural qubits [10]. These are some of the qualities that have caused superconducting qubits to be regarded as one of the most promising candidates for qubit construction in quantum computers going forward [4, 18–20].

Since superconducting qubits are electric circuits, all interaction between them and any external hardware is done via electrical pulse signals [14, 16, 17]. Sending a pulse to the qubit moves it between different superpositions depending on the properties of the pulse, and reading the state of the qubit is similarly done by examining the properties of outgoing pulses. The shapes, lengths and timing of these pulses all affect how the state of the qubit is modified. This means that in order to properly control a QPU built with superconducting qubits, one would also need hardware capable of generating arbitrary and precisely configured electromagnetic pulses.

1.1.2 Current waveform generation solutions

There currently exists hardware solutions for generating highly customised waveforms, known as Arbitrary Waveform Generators (AWGs). With these it is possible to specify the exact shape of an outputted wave down to parts of a nanosecond in resolution. Commonly, this is done by uploading a list of points representing the pulse sequence to the AWG. These points specify the exact amplitude to be outputted for every time step, including zeroes for all times between pulses.

This method of working does not lend itself well to lengthy pulse sequences. Consider, for

[4] For the reader interested in a more thorough explanation of the concepts pertaining to quantum computing, see [8, 9]

instance, some experiment where one wants to output a set of pulses a large amount of times with a relatively long delay between them. To accomplish this one would need to either upload a very long list of points to the instrument's memory, or communicate back and forth every iteration, performing a new upload each time. Any issues arising from this are further magnified if one also wants to perform averaging over multiple runs of the same experiment.

Recently, the company Intermodulation Products has been developing an FPGA-based AWG known as *Vivace*, which offers a more flexible system for scheduling pulses and readout compared to earlier solutions [21]. Intermodulation Products consists of a team of researchers from KTH Royal Institute of Technology, and development of Vivace is done in collaboration with the Quantum Technology (QT) laboratory at Chalmers University of Technology. Vivace differs from the traditional AWG in how waveforms are defined. Instead of specifying pulse sequences as timelines of points, Vivace allows for pulses to be seen as configurable and reusable units, that can be scheduled for emission at specific points in time. Lists of points are only used to modify the overall shape of a pulse. As a pulse definition within Vivace includes both a starting time and a duration, the user does not have to explicitly define the space between pulses. As such, measurements can be set up in arbitrary lengths of time without affecting performance negatively.

However, Vivace on its own can only be interacted with via its Python API, so up until now researchers have had to write individual script files for every pulse sequence they want to run. This is problematic for two reasons. Firstly, it is inconvenient for researchers to write and modify Python code every time they want to define or update a measurement. For complex pulse sequences, the most efficient way to set it up through the API can be unintuitive, and not all researchers are proficient in writing code. The second issue is that only running Vivace through individual scripts makes it difficult to coordinate with other instruments, which are usually managed via some centralised instrument control software platform.

1.1.3 Quantum computer control structures

One might wonder how a user might eventually interact with a "finished" quantum computer. Will they be expected to set up individual pulse waveform parameters? The general answer is no; pulse sequence construction is expected to be but one abstraction layer among many within a larger *control stack* constituting the quantum computer. Just like how high-level code in a traditional computer goes through many conversions and compilations before becoming machine code affecting the hardware, quantum algorithms will require multiple software and hardware solutions to go from a high-level specification to pulses being sent through the QPU.

There is however no standardised vision of how such a stack should be constructed. Quantum computing is still in its infancy when compared to classical computing; the state of research is still too volatile for a consensus to be made across development teams [22]. However, a non-specific overview of a quantum computer control stack can be seen in figure 1.1. With this stack structure, a user is expected to define a quantum algorithm in some high-level software. Then, that algorithm will for each layer be transformed into an increasingly hardware-specific form, until specific pulses representing the quantum algorithm will be sent through the QPU, and sampled data returned to the user.

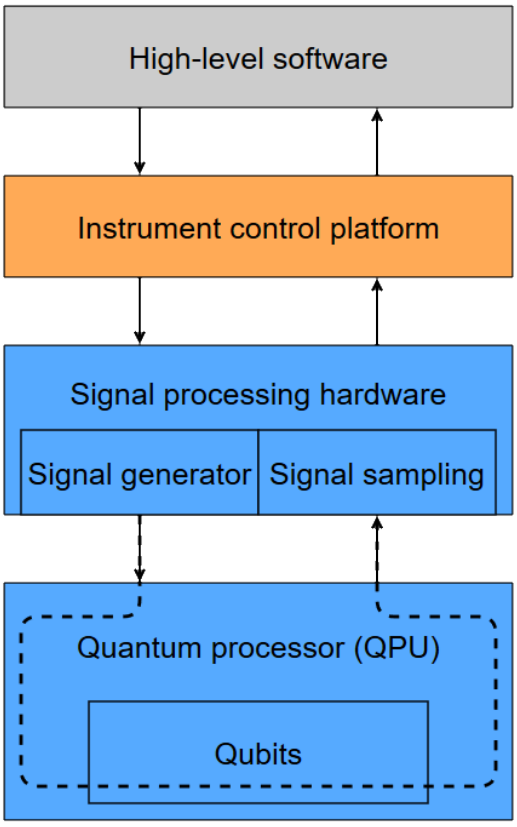

Figure 1.1: A general overview of a quantum computer control stack.

1.2 Problem description

Quantum processors based on superconducting qubits have been constructed at Chalmers' Quantum Technology laboratory. A few different pulse generators have been used to control these QPUs, with the most recent one being a set of Zurich Instruments HDAWGs (High Definition Arbitrary Waveform Generator). While functional, this solution suffers from the drawbacks of traditional point-by-point waveform specification. As such, QT wishes to move towards using Vivace for qubit control and readout. However, with the Vivace Python API being its only user interface, making new measurements and coordinating Vivace with other lab equipment is cumbersome.

These problems could be solved if it was possible to control Vivace with the instrument control platform in use at QT, *Labber* [23]. In addition to being a familiar working environment for the researchers at QT, Labber provides instrument control and readout via GUI and lets users set up experiments that use multiple instruments, through the use of *virtual instruments* that represent the hardware. A virtual instrument's behaviour is defined by software known as an *instrument driver*, which is responsible for communication between the virtual instrument GUI and the hardware instrument it represents.

The purpose of this project is to create a Labber virtual instrument that represents Vivace. This will let researchers work with Vivace's full feature set through the interface offered by Labber, both for issuing commands and reading result data. The increased abstraction

offered by the instrument would also allow for adding functionality beyond that which is offered directly by the Vivace API.

Beyond aiding in human-QPU interaction, the instrument would also allow for Vivace to be used by higher abstraction layers that extend above Labber. Examples of these include the Labber scripting API, which allows for efficient automation of experiments [24], and a web-based QPU control scheme that is currently being developed at QT. See figure 1.2 for a diagram of the abstraction stack, including layers that are planned to extend above Vivace's Labber driver.

A Labber driver for Vivace could potentially also benefit other institutions with QPUs based on superconducting qubits. Today, many such institutions use a variety of hardware and software solutions to control and read their QPUs, with many of them being developed in house and not made public [16, Sec. VI]. In contrast, the Vivace platform is intended to be publicly available for purchase, and the driver code developed during this project is planned to be made freely available through the official Labber driver repository [25].

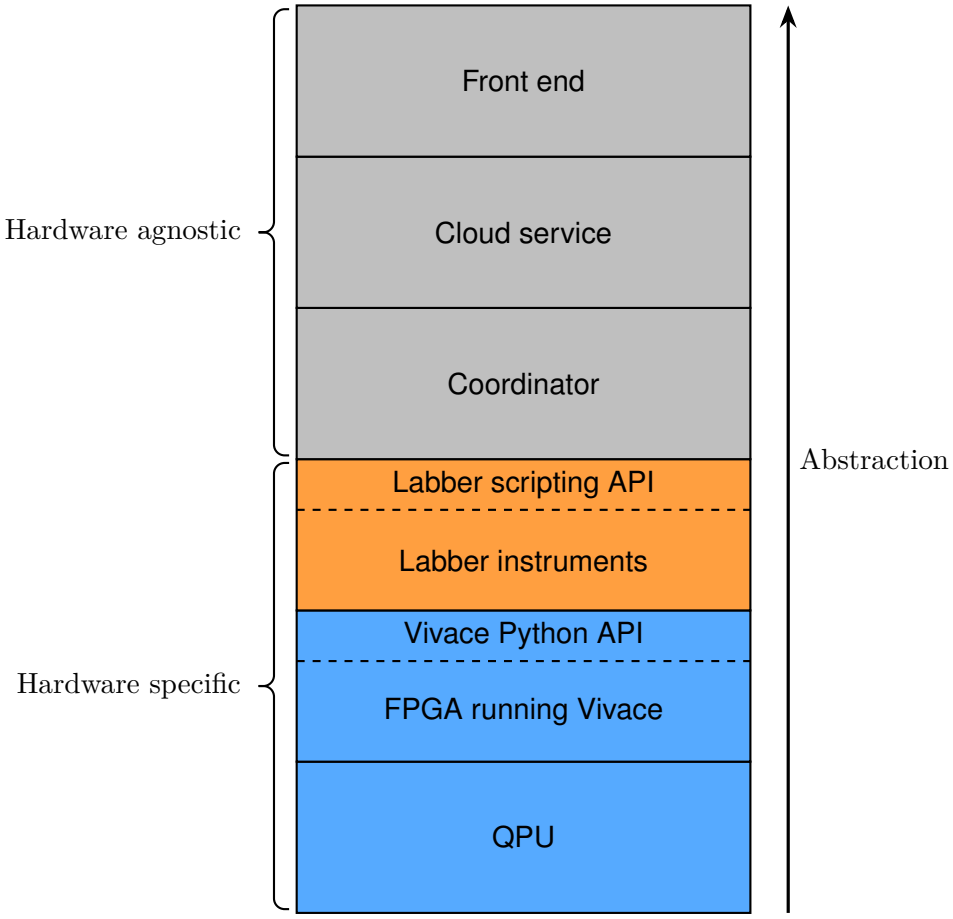

Figure 1.2: A generic visual representation of the quantum computer control stack planned at QT. The envisioned workflow is that a quantum algorithm is created by a user through a front end application and sent through the cloud as a list of instructions to a coordinating system with access to some number of QPU control stacks (possibly of different architectures). The coordinator then issues these commands from the user to an available control stack that can perform the requested tasks. Finally the results are propagated back up the stack to the user. Currently at QT only the blue-coloured parts are integrated fully, the orange are only partly implemented, and the grey are not yet implemented.

1.3 Aim

The main research question of this project is as follows: how should a system be designed that allows for convenient and flexible generation of intricate arbitrary pulse sequences for control and readout of a superconducting QPU setup, that is also easily automated and run alongside other laboratory equipment?

The goals that such a system should fulfil are divided into some core functionality requirements, as well as a few additional features that were requested to be implemented in the case time permitted. This core functionality, which we consider necessary to regard our solution a success, we dub our primary goal, and the additional features a secondary goal. It should be noted that most lab equipment at QT is currently run through Labber, and

as such the choice to develop our drivers within Labber was implicitly decided for us.

1.3.1 Primary goal

The primary goal of this project is to develop a virtual instrument for Labber which should communicate with the Vivace platform. The purpose of this instrument is to let a user schedule the sending of electrical pulse waveforms to the QPU as well as when to sample the QPU for data. More specifically, the following features should be supported by such an instrument:

- Pulses should be easy to schedule at an arbitrary time within a sequence. This makes it so that the user does not have to create the pulse sequence in a strict chronological order.

- Pulses should be defined by a base shape (envelope), and an optional carrier wave. These carrier waves should be sine waves for which the user can specify amplitude, frequency and phase.

- Pulse waveform shapes should be able to be defined as templates independently of carrier wave parameters, so that they can be easily reused for multiple pulses. The user should have the ability to pick from common presets (such as square, sine, triangle, etc.) in addition to having the option to define their own.

- It should be possible to create *port pairs*, where one port outputs the same pulses as the other, but with an optional phase shift and amplitude scale multiplier. This could be arranged manually on both ports in question as long as the above points are fulfilled, but an easier way to accomplish this would be in the user's best interest. Benefits of outputting pulses on paired ports are detailed in section 2.1.4.

- A way to create *sweeps*, repeats of a sequence of pulses but with some modification to the sequence between repetitions. E.g. one or more pulses increasing their frequency or duration every repetition. This should ideally be accomplished without sending data to and from the FPGA multiple times, due to such data transfers being a performance bottleneck.

- The instrument should support the usage of Vivace's interleaved averaging feature for more noise-resistant data. A more thorough explanation of interleaved averaging can be found in section 2.2.3.

- It should be possible to set up specific types of pulses used for quantum control, such as *Derivative Removal by Adiabatic Gate* (DRAG) [26] pulses, that make use of multiple output ports.

- A way to query the QPU at given times and present the resulting data in an easily readable manner. The user should be able to apply certain transformations to the data in order to better view certain aspects of it.

- The instrument should be easily controlled and coordinated along with other Labber instrument when constructing measurements.

1.3.2 Secondary goal

One additional goal for this project is the development of an automation script that can be used to run a series of qubit measurements. Such a script should be able to:

- Start several qubit measurements in a row without needing human intervention.
- Access and parse the results of a completed measurement in order to perform necessary calculations on the result data. This includes tasks such as fitting result values to some curve and judging the quality of the fit. If the error parameters of the curve fit are abnormally large, the script should be able to react accordingly (measure again or abort).
- Adjust some number of input parameters of measurements based on the results of one or more earlier measurements in the sequence.
- Provide continuous feedback as the measurements are run. While the script should not need human intervention, a human observer should still be aware of the script's progress and status at a glance.

Much of the functionality offered by such an automation script are tasks that would otherwise have to be done by a human, either manually or through the use of several different programs. The successful completion of this goal would serve to streamline this kind of procedure. If a number of similar scripts were to be written for some "universal" set of tasks and a variety of hardware solutions, this could lead to the formation of another abstraction layer. This layer would represent a move away from hardware-specific methods of qubit interaction, in accordance with the higher levels of QT's planned control stack, as seen in figure 1.2.

Since an automation script needs to not only handle communication with Vivace, but also run Labber measurements and control other instruments, it seems natural for it to be implemented using Labber's API for measurement automation. While the script could communicate with Vivace directly, it would need to re-implement most features from our Vivace instrument anyway to do so. Thus, we consider the completion of the primary goal a prerequisite for this secondary goal.

1.4 Limitations

The project is mainly focused on classical programming a few levels of abstraction above the quantum processor. So while the software developed as part of this project itself can and will be used for realising quantum algorithms, only classical programming is required for developing it. As such the project does not contain any novel ideas within quantum programming, and the measurements used for program verification are well established measurement protocols for characterisation of a qubit.

As seen in figure 1.2, the software developed for this project is planned to be incorporated into a full stack of abstraction layers from QPU to some high-level software. While our software interfaces with the layer below it (the Vivace platform), the planned higher layers have not had their implementation specifics fully established during the course of

this project. This means that we are not able to make any particular design choices to our software to facilitate the connection upwards in the abstraction stack. However, since most such connections are handled via Labber's API, the design of our driver should not have a considerable impact on the development of higher abstraction layers.

While our research question concerns waveform pulse generation in the general sense, this project is limited to working with the Vivace pulse generator and the Labber instrument control platform. The primary reason for this is availability. Vivace is the most advanced pulse generator at QT, and Labber is used for control of all other hardware instruments at the laboratory. It is out of scope for this project to rewrite drivers for local oscillators among other instruments for some alternate control platform.

1.5 Method description

In order to construct the virtual instrument specified in section 1.3.1, we need to develop two main components:

- A Labber instrument GUI which lets the user control the instrument.
- A driver which dictates the instrument's behaviour and issues commands to the Vivace platform based on the user's input.

The development of this instrument requires an understanding of both Labber's and Vivace's APIs, and as such a portion of the project is dedicated to the study of these. Work on the driver is carried out in an iterative loop of adding a single feature, then performing informal qualitative testing and verification. For the most part, the GUI is developed in tandem with the driver, by creating components related to the feature currently being worked on. All GUI development is done using Labber's provided framework. The development is done in close collaboration with our supervisors, who decide which features to add as well as their level of priority. In addition, they also perform full or partial testing and evaluation of completed features. Since they have more knowledge of realistic test scenarios and the expected behaviour of the system, they are able to provide valuable feedback.

Once the driver has the sufficient features needed to run some qubit measurement, we set up and run the measurement in question on actual qubit hardware. This serves multiple purposes:

- It lets us test if the relevant features can be used together without causing bugs or unexpected behaviour.
- It provides measurement results that can be compared against expected results to gauge whether the instrument generates the correct output in an authentic use case.

Once the instrument is feature-complete enough that all desired qubit measurements can be performed with it, the focus of the project is shifted to the development of a qubit tune-up script in order to fulfil the project's secondary goal, as outlined in section 1.3.2. The tune-up script makes use of Labber's scripting API to perform a sequence of measurements, where results of individual measurements are fed as input parameters into following ones.

In a similar fashion to the planned implementation strategy for the driver, the tune-up script is implemented and tested one measurement at a time. Evaluation consists of comparing the intermediate and final measurement results against recorded results obtained when running the measurement sequence manually.

2

Theory

This chapter provides an overview of various concepts relevant to the thesis. The reader is not expected to possess any particular prior knowledge of these topics, so depending on their academic background certain sections herein may not offer them anything new.

2.1 Signal processing for superconducting qubits

Interaction with a superconducting qubit is carried out through electrical microwave pulses. This section goes over the basics of how these pulses are formed, and the various transformations they undergo as part of our procedures used for qubit interaction; useful for understanding the underlying reasons for many features and requirements of our drivers.

2.1.1 Envelopes, carriers and pulses

The driver produced as part of this project is meant to be a tool for producing electromagnetic microwave pulses. As such, readers should be aware of what such a pulse is. First of all, a wave is simply an oscillation through some medium. It is described in terms of its *frequency* (how fast it oscillates) and *amplitude* (the height of the wave). Waves of the same frequency and amplitude can still differ in terms of their *phase*; a measure of where they are within their oscillation cycles in relation to each other. Figure 2.1 shows an example.

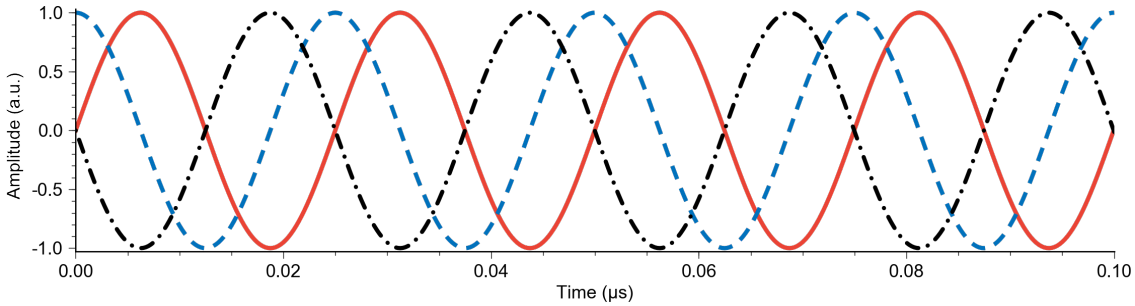

Figure 2.1: A sine wave (in red) at a frequency of 40 MHz. The blue dashed line is at a phase offset of $\frac{\pi}{2}$ radians, effectively acting as a cosine wave. The black dot-dashed line has a phase offset of π radians.

A *pulse* is a short period of activity, in this case a wave transmitted during a short period

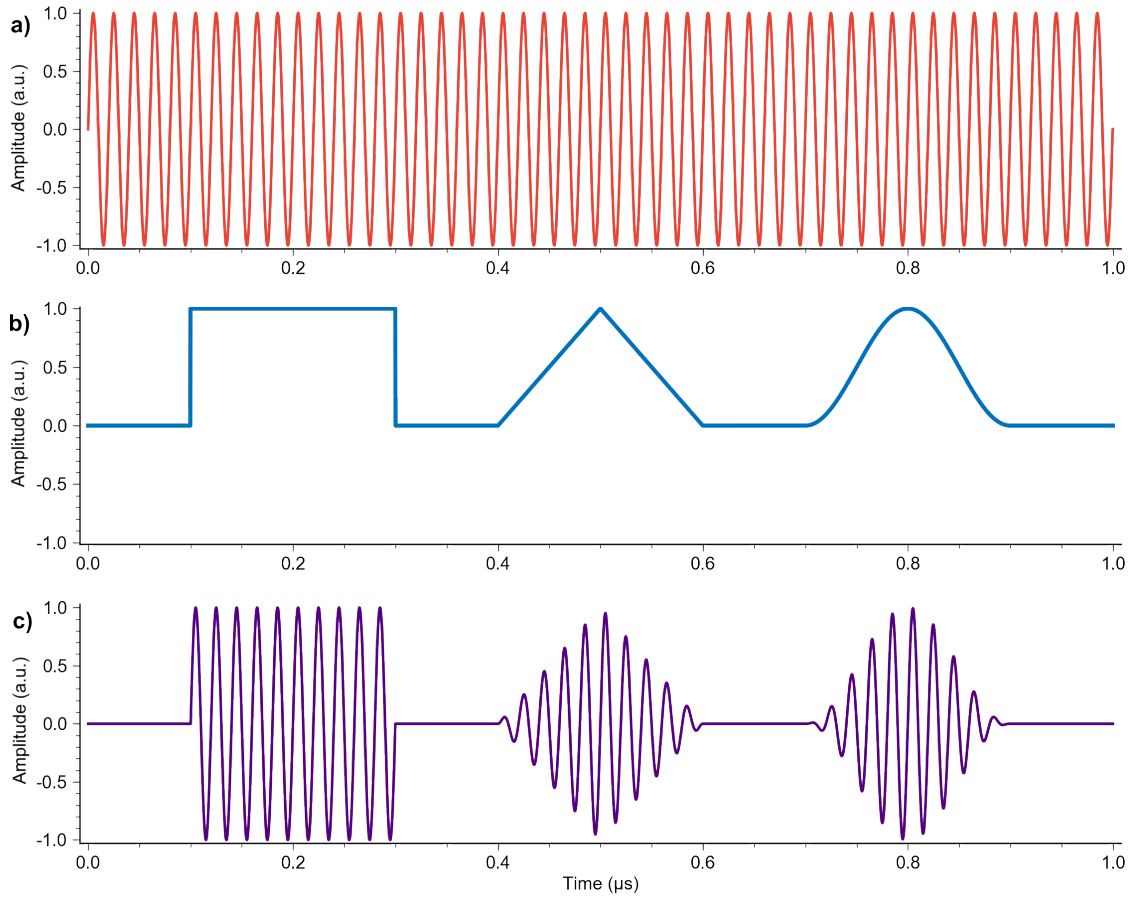

Figure 2.2: A carrier wave (top, in red) has its amplitude modulated by three different envelopes (middle, in blue) to produce three wave pulses (bottom, in purple) that use the carrier to encode the envelopes' shapes. The three envelope shapes are (from left to right) *square*, *triangle* and Sin^2.

of time. In the context of this thesis, pulses are formed by *modulating* a wave's amplitude with an *envelope*. An envelope is a function of time in the range $[0, 1]$, that starts and ends at 0. Amplitude modulation is performed by multiplying a wave's amplitude with one's chosen envelope, forming a pulse of the envelope's shape. Figure 2.2 shows a few examples of this process. It should be noted that when using envelopes in this way, the initial wave is referred to as a *carrier wave*, as it carries the information of the envelope's shape.

2.1.2 Analogue and digital signals

It is hard to work with continuous signals within software. Unless it can be completely defined through a known mathematical function, the signal needs to be digitised, i.e. stored as a list of discrete points representing the wave. So although the signal generators connected to Vivace output analogue waveforms, at the software level we are only able to represent signals as lists of height values, implicitly spaced in time by a pre-defined value. In the same way, when recording an analogue signal into a digital representation, it needs to be sampled at discrete points in time, storing each value separately. Vivace has a sampling rate of 4 GHz, and thus all signal values have a distance of 0.25 ns between each other.

2.1.3 Carrier modulation

For certain mediums, only signals within a specific frequency range can be transmitted. As an example, superconducting qubits have resonance frequencies in the GHz range, and thus pulses sent to such a QPU needs to be at similar frequencies. To transmit low-frequency information through a medium with a high-frequency bandwidth, techniques known as modulation and demodulation are employed. When modulating, the low-frequency (f_S) information that needs to be transferred is combined through a mixer with a carrier wave at a high frequency (f_{LO}) that the transferring medium accepts. This mixing is known as upconversion, and produces a high-frequency signal at f_{LO}, along with two other signals: the *lower sideband* at $f_{LO} - f_S$, and the *upper sideband* at $f_{LO} + f_S$. All the information of the original signal is contained in one of these sidebands, so generally some technique such as filtering is used to suppress the f_{LO} frequency along with the other sideband before transmission. On the receiving end, demodulation, or downconversion, is performed in order to retrieve the low-frequency signal from the received sideband. Demodulation is done in different ways depending on how the sideband was created, but one way to do it is by simply mixing the incoming signal with another carrier wave with frequency f_{LO}. This, just like the upconversion, creates signal content at f_{LO}, but since the incoming signal is at frequency $f_{LO} - f_S$, this time the sidebands instead show up at $f_{LO} - (f_{LO} - f_S) = f_S$ and $f_{LO} + (f_{LO} - f_S) \approx 2f_{LO}$. We once again have signal content down at our original frequency of f_S, and the rest of the generated signal content can be easily removed by a filter set to remove everything above some frequency between the possible values of f_S and f_{LO}.

2.1.4 IQ modulation

One issue with simply filtering out f_{LO} and one of the sideband frequencies after upconversion is that hardware filters are generally locked to specific frequencies. This makes such setups inflexible with regards to the frequencies at which information can be sent. A more flexible solution is *IQ modulation* (also known as quadrature amplitude modulation), which is the modulation scheme employed for sending signals to the QPUs at QT.

IQ modulation makes use of an IQ mixer, which has three inputs: signal I, signal Q and the LO signal. An IQ mixer consists of two separate mixers: one which mixes the I signal with the LO signal, and one that mixes Q with a version of the LO signal that is phase-shifted by $\frac{\pi}{2}$ radians. The outputs of these two mixers are then combined to form the IQ mixer's only output.

The purpose of an IQ mixer is to create a single sideband of the upconverted signal I. To accomplish this, the input signal Q must be a copy of I, phase-shifted by $\frac{\pi}{2}$ radians. The name IQ comes from this phase adjustment, with the two versions of the signal being referred to as the *In-phase* and *Quadrature* signals respectively. As described in section 2.1.3, the two mixers inside the IQ mixer each produce upper and lower sidebands at frequencies $f_{LO} - f_S$ and $f_{LO} + f_s$. Additionally, the phase φ of the two sidebands similarly becomes $\varphi_{LO} - \varphi_S$ and $\varphi_{LO} + \varphi_S$ respectively, which means $0 - 0$ and $0 + 0$ in the I case and $\frac{\pi}{2} - \frac{\pi}{2} = 0$ and $\frac{\pi}{2} + \frac{\pi}{2} = \pi$ in the Q case. Thus, when the results of the two mixers are summed, the lower sidebands – which are identical – add together to become stronger, while the upper sideband signals cancel each other out. This happens because their phase offsets are 0 and $\frac{\pi}{2}$, respectively, which means that they are mirror images of each other like the red and black signals in figure 2.1. Thus, the same IQ mixer

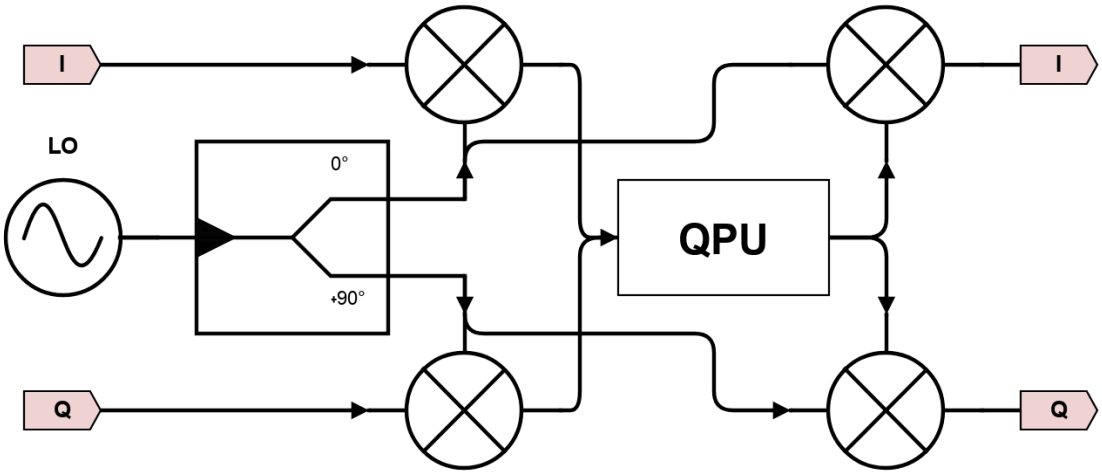

Figure 2.3: A schematic depicting a hardware setup performing IQ modulation and demodulation for sending signals through the readout line of a QPU.

can upconvert signals to a variety of frequencies, due to how the signal itself is used for sideband suppression.

For downconverting a signal that has been upconverted with IQ modulation, the natural choice of technique is IQ demodulation. As mentioned in section 2.1.3, downconversion in the general case is achieved by mixing the upconverted signal with a signal of the high frequency f_{LO}, producing the (approximately) original signal in the lower sideband. IQ demodulation differs slightly in that two versions of the f_{LO} signal is prepared; one without phase offset (I) and one at a $\frac{\pi}{2}$ offset (Q). The signal to be downconverted is mixed separately with each, producing two signals. The upper sidebands of these signals, as explained in section 2.1.3, end up at a frequency high enough to be easily filtered, while the lower sidebands return to the frequency f_S. The phase φ of these lower sideband signals becomes $0 - 0$ for I and $\frac{\pi}{2} - 0) = \frac{\pi}{2}$ for Q. The results of IQ demodulation is thus the signal pair that was originally used for the upconversion (along with any modifications to the signal by the QPU). A schematic of the signal connections and hardware components needed to perform IQ modulation and demodulation is shown in figure 2.3.

2.2 A brief introduction to quantum computing

This section explains certain concepts related to quantum computers, more precisely the ones relevant for grasping the purpose for the work carried out as part of this thesis project; understanding the effects and usages of different pulse sequences. The section starts by explaining the basics of quantum information and the qubit as well as the Bloch sphere, a useful tool for visualising qubit states. We then move on to how different gates can be used to transform quantum information, and lastly give an overview of the kind of qubit interactions that are used as part of our verification process.

2.2.1 Quantum information

Much like how a bit is the fundamental unit of information for a classical computer, a *qubit* plays the same role for a quantum computer [8]. In order to better explain the qubit, some notation needs to be introduced. When discussing quantum information, the *bra-ket* notation system is commonly used [27][28, Sec. 2.3]. Within this system, the state ψ of some qubit is denoted $|\psi\rangle$. The form $|\ \rangle$ is known as a *ket*, and simply signifies a column vector. The symbol written within the ket is nothing more than the vector's label, just as the symbols used to denote any other vector. The two observable states of a qubit are commonly referred to as $|0\rangle$ and $|1\rangle$, respectively.

At any point, a qubit is in some state $|\psi\rangle = a|0\rangle + b|1\rangle$, where $a, b \in \mathbb{C}$ are such that $|a|^2 + |b|^2 = 1$, and $|a|^2$ and $|b|^2$ are the probabilities that a measurement of $|\psi\rangle$ results in $|0\rangle$ or $|1\rangle$, respectively. As long as neither a nor b are equal to 0, the qubit in question is said to be in a *superposition* of states. One model for visualising the possible states a qubit can be in is the *Bloch sphere* [16]. The model takes advantage of the fact that $|a|^2 + |b|^2 = 1$, which means that state vectors can be visualised as points on the surface of a sphere with radius 1. Figure 2.4 shows a Bloch sphere. Note the poles denoted $|0\rangle$ and $|1\rangle$; these signify the states where $|a|^2 = 1$ and $|b|^2 = 1$, respectively. In the Bloch sphere, a state vector's polar angle (i.e. the angle between it and $|0\rangle$) represents the ratio between a and b. For instance, the state vector $|\psi_1\rangle$ in figure 2.5 has a higher probability of being measured as $|0\rangle$ than as $|1\rangle$, while $|\psi_2\rangle$ in the same figure has a 50/50 probability of either.

A qubit state can be expressed on the form

$$|\psi\rangle = e^{i\alpha}\cos\left(\tfrac{\theta}{2}\right)|0\rangle + e^{i\beta}\sin\left(\tfrac{\theta}{2}\right)|1\rangle \tag{2.1}$$

where α and β are the complex phases of a and b, and θ is the polar angle of the state vector. However, since a quantum state can be multiplied by any complex scalar of unit norm without changing [29, Ch. 3], it holds that $|\psi\rangle$ represents the same state as $e^{i\chi}|\psi\rangle$ for any angle χ and state $|\psi\rangle$. So for simplicity's sake we can by setting χ to $-\alpha$ write qubit states on the form

$$\begin{aligned} e^{-i\alpha}|\psi\rangle = \\ e^{i(\alpha-\alpha)}\cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i(\beta-\alpha)}\sin\left(\frac{\theta}{2}\right)|1\rangle = \qquad (\phi = \beta - \alpha) \\ \cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i(\phi)}\sin\left(\frac{\theta}{2}\right)|1\rangle \end{aligned} \tag{2.2}$$

The phase difference ϕ is represented in the Bloch sphere as the azimuthal angle of a state vector (the angle around the Z axis from the positive X axis). The vector ψ_3 in figure 2.5 shows how both θ and ϕ affects the position of the state vector.

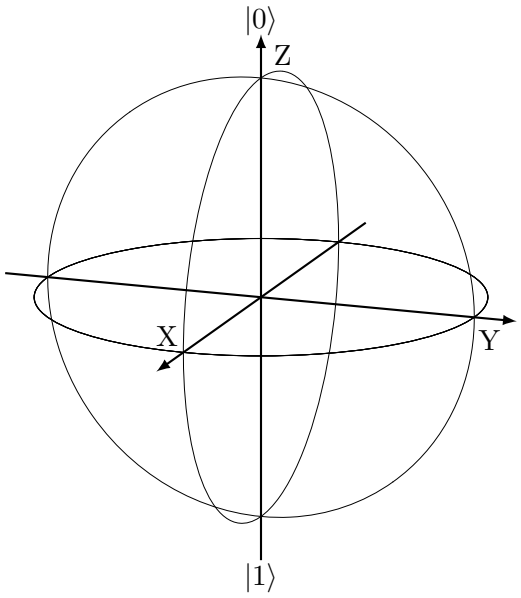

Figure 2.4: A diagram of a Bloch sphere.

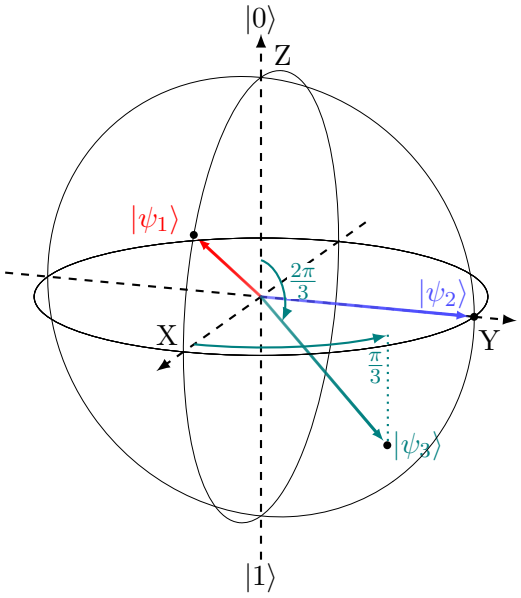

Figure 2.5: A Bloch sphere, with three state vectors:
$|\psi_1\rangle = \sqrt{\frac{3}{4}}\,|0\rangle + e^{i0}\sqrt{\frac{1}{4}}\,|1\rangle$, $(\theta = \frac{\pi}{3},\ \phi = 0)$
$|\psi_2\rangle = \sqrt{\frac{1}{2}}\,|0\rangle + e^{i\frac{\pi}{2}}\sqrt{\frac{1}{2}}\,|1\rangle$, $(\theta = \frac{\pi}{2},\ \phi = \frac{\pi}{2})$
$|\psi_3\rangle = \sqrt{\frac{1}{4}}\,|0\rangle + e^{i\frac{\pi}{3}}\sqrt{\frac{3}{4}}\,|1\rangle$, $(\theta = \frac{2\pi}{3},\ \phi = \frac{\pi}{3})$

2.2.2 Quantum programming

A classical computer performs calculations by manipulating its bits using *logic gates*. Just as classical bits have an analogue in the form of qubits, these gates also have an analogue in quantum gates. As the states of classical bits are discrete, any input and output of classical gates is always either 0 or 1, meaning that the complete behaviour of a classical

logic gate can be written out as a logic table. Quantum gates, on the other hand, operate on quantum superpositions, of which there is an infinite amount. This means that rather than tables, these gates are described in the form of linear transformations within the qubit state space. The Bloch sphere makes it possible to visualise the effects of such gates as rotations of state vectors within the sphere (in ideal, noiseless conditions). Any single-qubit gate can be broken down into rotations about the X, Y and Z axes of the Bloch sphere. Three elementary single-qubit gates are the X, Y and Z gates, so named because they represent a rotation of π radians around the sphere's X, Y and Z axes, respectively [16, Sec. IV.C]. Gates are, of course, a theoretical concept. For actual superconducting qubits, electromagnetic microwave pulses are used to achieve the effects of gates on their quantum state [16, Sec. IV.D] (see section 2.2.3 for simple examples). A quantum program consists of setting the necessary qubits to some initial value, applying a series of gates (e.g. a sequence of pulses) to them in order to manipulate their state, and finally measuring them to see which state they turn out to be in.

Naturally, there are also gates that take more than one qubit as input. A common two-qubit quantum gate is the CNOT (Conditional NOT) gate, which applies an X gate on its second input qubit if the first input qubit is in state $|1\rangle$. These kinds of conditional interactions are what give rise to *entangled* states, where the second qubit is put into a superposition of the two outcomes of the CNOT gate, based on the first qubit's superposition [16, Sec. IV.B][30]. Entanglement is one of the key factors that enable quantum computer programs to potentially surpass the limits of their classical counterparts, but multi-qubit interactions lie outside the scope of this thesis. Interested readers are recommended to see [31] for a simple demonstration of a quantum program that makes use of entanglement.

2.2.3 Qubit characterisation

A qubit possesses a variety of parameters used to describe its properties and behaviour. One would need to know the values of these to make proper use of the qubit in question. By performing interactions with the qubit in order to learn these parameter values, one is said to *characterise* the qubit. This section goes over a few common kinds of characterisations for two reasons: first, this serves to give examples of how pulses of different shapes can manipulate a superconducting qubit. Second, it familiarises the reader with measurements that are used to test the instrument developed as part of this project.

When working with our superconducting qubit, we use two distinct interaction channels, which we refer to as *control* and *readout*. On the control channel, electromagnetic pulses of varying forms act as single-qubit gates, modifying the state of the qubit. Connected to the qubit is a resonator, and it is to this resonator one can send pulses to determine the state of the qubit [14].

For many kinds of characterisation measurements, one needs to determine what specific superposition a qubit was in before it was measured. As we know, a qubit measurement will always yield a binary outcome: we either get $|0\rangle$ or $|1\rangle$. The relation between a qubit's superposition and measured value is analogous to a weighted coin; by flipping the coin once, very little information is gained as to what way the coin is weighted. Similarly, measuring a qubit once gives little information about the superposition it was in prior to the measurement. But by flipping/sampling many times, an increasingly accurate estimate of the ratio between the two states can be calculated via averaging. So in order to accurately gauge what superposition a qubit is in, one can repeatedly transform it into

the same state many times and average the amount of times it was measured as $|0\rangle$ and $|1\rangle$ respectively. The ratio of times the qubit was measured as $|1\rangle$ to the total number of measurements is known as *qubit population*.

In addition to averaging, a measurement might involve what is known as a *parameter sweep*. A sweep of a certain parameter means that the same sequence of pulses is sent to the qubit some number of times, but with the parameter using a different value for each iteration. Parameters can be things such as pulse amplitude, frequency, duration, start time, etc. Iterations are executed with enough delay to allow the qubit to return to $|0\rangle$ every time. While performing a sweep, it is still important to average the results for each value of the sweep. However, if the averaging is performed during the sweep, such that all averaging measurements are done for one parameter value before moving on to the next value, temporary noise or shifts over time of the qubit only affects certain parts of the sweep. It is better to use *interleaved averaging*; by performing one full sweep repeatedly and then averaging values of each sweep, these fluctuations instead affect all parts of the measurement somewhat equally, and have their impact lessened by the averaging.

2.2.3.1 Resonator spectroscopy

As mentioned above, the qubit circuit's readout channel can be used to measure the qubit's state. For some frequency, determined by the qubit, the resonator connected to the qubit circuit routes incoming electromagnetic pulse signals of that frequency to the circuit's ground. This frequency is known as the resonator frequency, denoted f_{res}. f_{res} differs slightly depending on if the qubit is in $|0\rangle$ or $|1\rangle$, denoted as f_{res_0} and f_{res_1} respectively. The difference between them is known as a dispersive shift. Finding f_{res_0} can be done by sweeping signals of gradually increasing frequency on the input line of the readout channel, while monitoring its output line. If the sent pulse's frequency is not routed to ground, the difference in signal power between input and output is negligible. If, on the other hand, the signal's frequency causes it to be absorbed by the resonator, the output is considerably weakened compared to the input. A successful resonator spectroscopy should look similar to the example in figure 2.6. The noticeable "dip" in the centre of the image marks the frequency value of f_{res_0}, where the difference in power between input and output signals is at its highest.

2.2.3.2 Two-tone spectroscopy

Once f_{res_0} is known, the next parameter to discover is the qubit frequency f_{01}. Control pulses sent at this frequency are said to be *resonant* with the qubit, and will have the greatest effect on its state. If a pulse is not quite resonant, the rotation of the qubit state will not be around the desired axis of the Bloch sphere, and will instead rotate around a tilted axis, not able to reach $|1\rangle$ from $|0\rangle$. f_{01} is identified by monitoring the readout channel at frequency f_{res_0} while also sweeping the frequency of a control pulse sent before the readout pulse. As the pulse frequency approaches f_{01}, the probability that the qubit will be measured as $|1\rangle$ increases. When the qubit is measured as $|1\rangle$, the dispersive shift will be in effect, moving the resonator frequency away from the monitored frequency, causing the readout channel's output signal to increase in strength. As this happens more frequently as we approach f_{01}, it causes the averaged readout value to spike, as seen in figure 2.7.

With f_{01} known, f_{res_1} can be found by fixing the control pulse at f_{01}, and once again

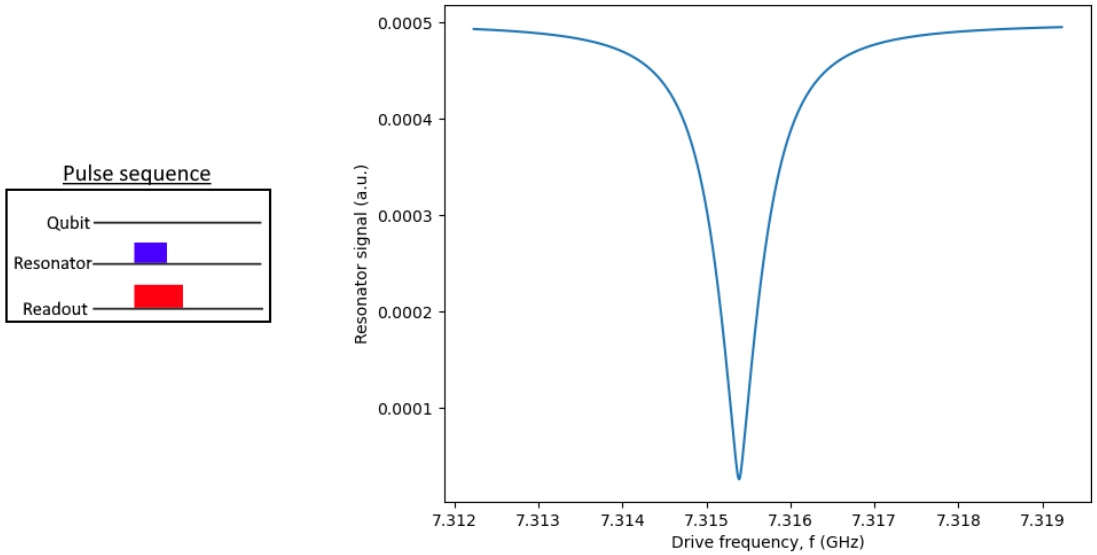

Figure 2.6: An artificial example of a resonator spectroscopy. The graph shows the resonator signal as a function of the readout frequency of the sweep, averaged over many measurements.

sweeping the readout frequency as in the resonator spectroscopy. This time, the dip in power is seen at f_{res_1}.

2.2.3.3 Rabi measurement

While it is true that a control pulse sent at frequency f_{01} can rotate the qubit from $|0\rangle$ towards $|1\rangle$, the degree of which the state vector is rotated is based on the control pulse's amplitude and duration. Higher amplitudes result in higher rates of rotation, while longer pulse durations will cause the rotation to be active for longer. For proper qubit control, it is important to know the pulse parameters that will cause a rotation of exactly π radians around the Bloch sphere (known as a π-pulse). One can find the proper amplitude for a given pulse duration by monitoring the readout channel at f_{res_0} while sweeping the amplitude of the control pulse, starting at 0. The resulting output should look similar to figure 2.8. The first peak of the curve corresponds to the qubit being in a pure $|1\rangle$ state, signifying that the π-pulse amplitude has been found. As the amplitude increases further, an even larger rotation is applied, rotating the qubit state past $|1\rangle$ back towards $|0\rangle$. This ever-increasing rotation causes the oscillating form of the graph.

2.2.3.4 Determining T_1

A qubit cannot maintain its state forever. Even though care is taken so that it is decoupled from the world around it, total isolation is not possible due to the simple fact that there needs to be some way to interact with it. As such, the qubit is susceptible to various kinds of noise from its environment, causing its state to shift over time through processes collectively known as *decoherence* [16, Sec. III]. It is considered a crucial requirement for quantum computing that one has time to manipulate and measure one's qubits before decoherence irreversibly damages their state [12]. It is thus necessary to characterise a

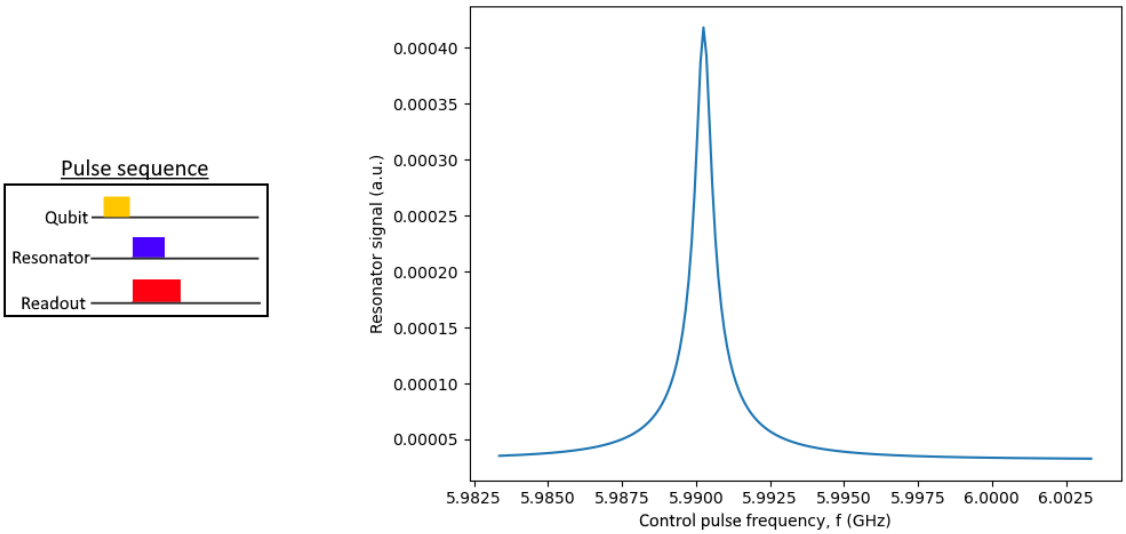

Figure 2.7: An artificial example of a two-tone spectroscopy. The graph shows the resonator signal as a function of the qubit control pulse frequency of the sweep, averaged over many measurements.

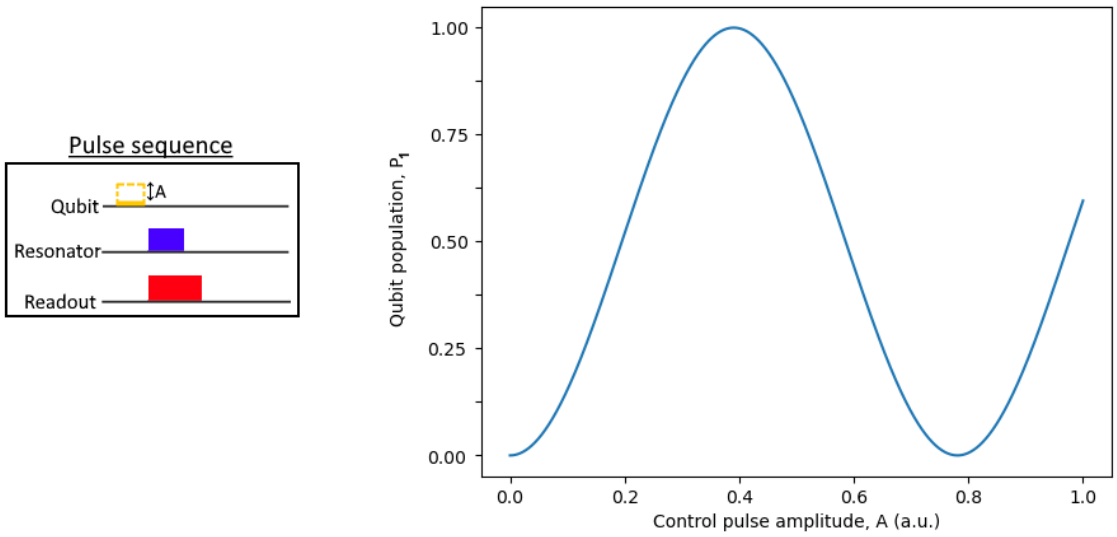

Figure 2.8: An artificial example of the results of a Rabi measurement. The graph shows the qubit population as a function of control pulse amplitude.

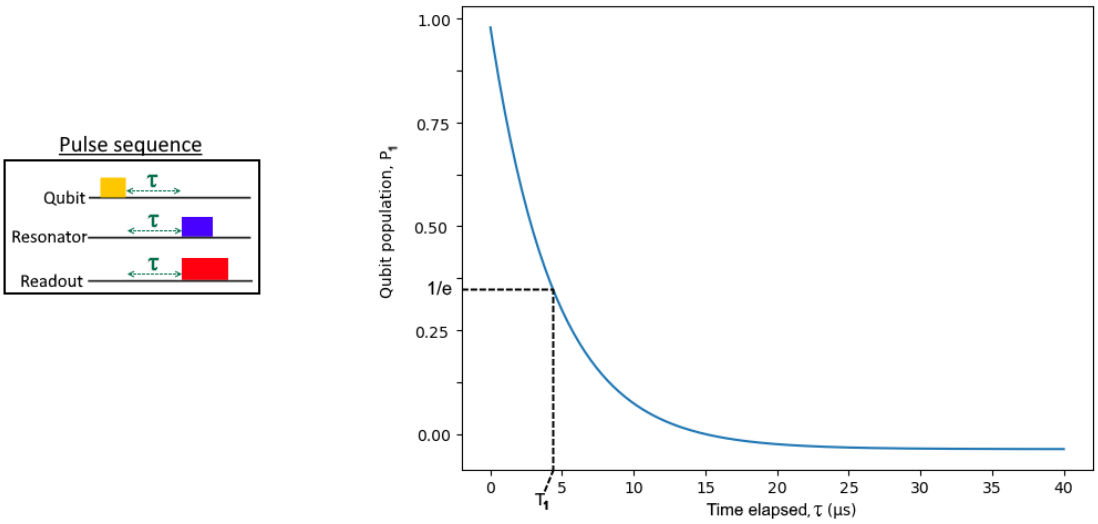

Figure 2.9: An artificial example of typical T_1 experiment results. The graph shows qubit population as a function of the time elapsed between transmitting the control pulse and the readout pulse. The value that T_1 would have been determined as in this hypothetical scenario is indicated by the dotted lines.

qubit's decoherence behaviour.

One common type of decoherence is known as *longitudinal relaxation* or *energy relaxation*, and can be seen as the qubit's state vector travelling up the Z axis of the Bloch sphere towards $|0\rangle$. The parameter used to describe a qubit's energy relaxation time is known as T_1. Determining it is done as follows: First, one sends a π-pulse (configured as per the results of a Rabi measurement) to the qubit in order to set it to the pure $|1\rangle$ state at the Bloch sphere's south pole. After some delay τ (that is swept over) the qubit's state is measured.

The results of the measurement can then be seen as the state vector's projected position on the Bloch sphere's Z axis as a function of τ. By fitting the normalised results to an exponential decay curve of the form $e^{-\tau/T_1}$, one can obtain the qubit's T_1 value. When $\tau = T_1$, its corresponding function value should be $e^{-T_1/T_1} = \frac{1}{e}$. T_1 can thus be found on the fitted data by looking at the τ value that corresponds to the function value of $\frac{1}{e}$. Figure 2.9 shows an example such fitted data.

2.2.3.5 Ramsey interferometry

In addition to energy relaxation, a qubit's state is also subject to transverse relaxation, i.e. loss of information about the qubit's phase. The qubit parameter that denotes transverse relaxation time is T_2, which is dependent on both the energy relaxation time T_1 (put simply: once the state has returned to $|0\rangle$ we have lost all phase information) and pure dephasing rate Γ_ϕ. Pure dephasing refers to a stochastic rotation around the Bloch sphere's Z axis (i.e. a rotation of the azimuthal angle ϕ) caused by noise. It holds that

$$\frac{1}{T_2} = \frac{1}{2T_1} + \Gamma_\phi$$

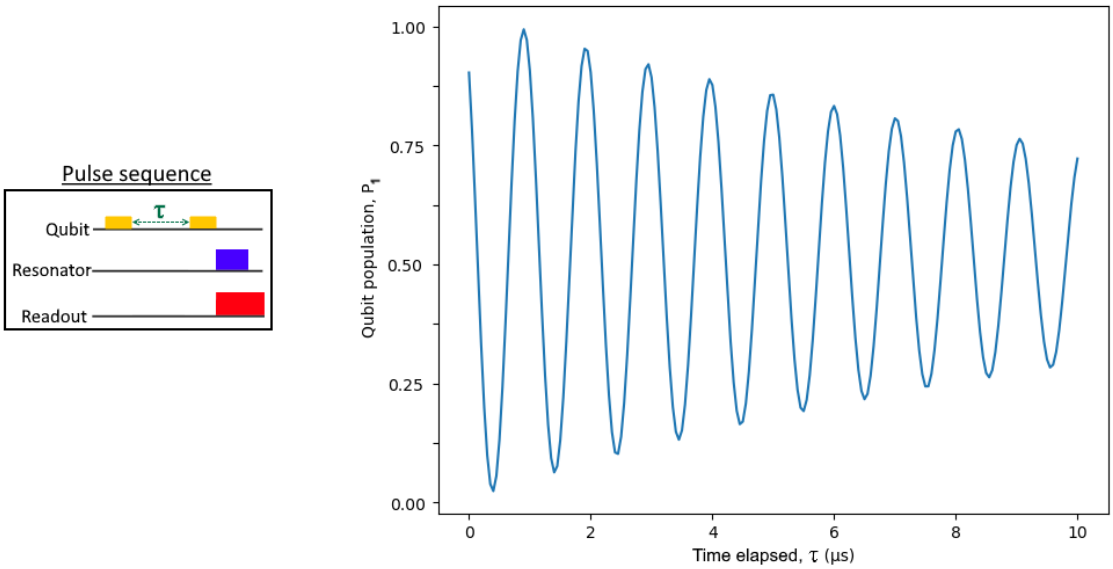

Figure 2.10: An artificial example of Ramsey interferometry results. The graph shows qubit population as a function of the time elapsed between transmitting the first and second $\frac{\pi}{2}$-pulses.

The transverse relaxation time T_2 of a qubit can be determined through a process known as Ramsey interferometry. To perform Ramsey interferometry, the qubit is initially rotated from $|0\rangle$ onto the equator of the Bloch sphere with a $\frac{\pi}{2}$-pulse (a π-pulse with halved amplitude to cause a rotation of $\frac{\pi}{2}$). Then, after a swept delay τ, another $\frac{\pi}{2}$-pulse is applied to the qubit, rotating around the same axis as the first pulse. The initial $\frac{\pi}{2}$-pulse is intentionally somewhat detuned from the qubit frequency in order to induce a rotation of the qubit's state over time around the Z axis of the Bloch sphere. As τ increases, this rotation causes the qubit state to have moved further along the equator before being rotated again, causing the results of the measurement to oscillate. These oscillations occur at a frequency equal to the difference between the qubit frequency f_{01} and the frequency of the control pulse. This means that by recording the frequency of the oscillations, it is possible to acquire a very exact measurement of f_{01} with a simple addition or subtraction.

The transverse relaxation causes the oscillations to converge over time; once the stochastic dephasing is applied for long enough, it is equally likely for the qubit's phase to be at any given value before applying the second $\frac{\pi}{2}$-pulse, causing the average qubit population to tend towards 0.5. The effects of the induced rotation along with the transverse relaxation is illustrated in figure 2.10.

The resulting data can be fitted to a function of the form

$$O + A \cdot \exp\left(\frac{-x}{T_2^*}\right) \cdot \cos(2\pi \cdot f_{\text{osc}} \cdot x + \phi_0)$$

where O is an offset in magnitude, A is the amplitude of the oscillations, x are the values of τ, T_2^* is the transverse relaxation time, f_{osc} is the frequency of the oscillations, and ϕ_0 is the initial phase of the oscillations. The $*$ in T_2^* signifies that Ramsey interferometry is subject to some additional noise (known as *inhomogenous broadening* [16, Sec. III]) that is separate from energy relaxation and pure dephasing.

2.3 Field-Programmable Gate Arrays

When designing a system built to handle very specific types of tasks (e.g. managing the hardware needed for sending and reading electrical pulses to a QPU), there are two main approaches: through software or hardware. The software approach, consisting of running compiled high-level code on a traditional CPU, offers great flexibility in what features and functionality can be implemented. However, a CPU still needs to sequentially execute instructions from its available set. As this instruction set is designed for general-purpose computation, it will not be optimised for any particular task. The hardware-based approach, on the other hand, would forgo the use of a general-purpose CPU and make use of specialised low-level circuitry. Such circuits are known as Application-Specific Integrated Circuits (ASICs). These work well, *if* they exist for the purpose at hand; integrated circuits are expensive to design and have a high initial cost before being able to mass produce. This means that specific integrated circuits are often only produced when large production volumes are foreseen. Additionally, settling on working with an ASIC necessitates that most required functionality is known beforehand; if new features are added for the system being designed, one would need to redesign and produce a new ASIC.

As a form of compromise between the efficiency of an ASIC and the flexibility of high-level software stands the Field-Programmable Gate Array (FPGA). The advantage an FPGA offers over an ASIC is spelled out in the first half of its name. Being *field-programmable* refers to how the FPGA's user, rather than its manufacturer, defines its behaviour. This behaviour is dictated by logic blocks, simple configurable units that handle specific types of calculations, which themselves can be configured to interact with each other in specific ways [32]. The code that determines an FPGA's behaviour is usually referred to as its gateware (as in [33]) or firmware (as in [34]). Not only do these configurable sections allow for an FPGA to be tailor-made to its user's needs, but the same hardware circuit can be repurposed to add new features, optimise existing functionality, and rectify errors from previous iterations. These properties lend themselves well to prototyping circuits for new purposes, and FPGA boards equipped with signal generating capabilities have seen widespread use with superconducting qubits[33–41].

2.4 The Vivace platform

The Vivace platform is actively being developed by Intermodulation Products [42], receiving direct feedback from the QT team at Chalmers. The Vivace hardware unit consists of an FPGA board[1] running Vivace's proprietary gateware at its core, capable of sending and receiving electromagnetic pulses at up to 4 GHz frequencies on 8 input ports and 8 output ports. Besides the hardware, the platform also comprises a Python API[2], which serves as the only way to control the hardware and receive digitised sampling data.

The setup of a measurement[3] through the Vivace API is done by defining a pulse sequence through a series of function calls and then ultimately calling the API function `perform_measurement()` to execute everything in the order that was specified. Pulses in Vivace consist of both an envelope template and an optional carrier (sine) wave. Envelopes are created by uploading a digitised list of points that form the envelope to a specific carrier generator; Vivace has two carrier generators per output port, and they can each hold up to eight envelope templates. Once an envelope is uploaded to Vivace, it can be scheduled for output with the function `output_pulse(envelope, start time)`. All outputs on a port are scaled according to the currently selected amplitude value, so it is important to make sure that the selected value is the desired one. The changing of this amplitude value also has to be scheduled explicitly by the user.

Unless otherwise specified, an emitted envelope modulates with the currently outputted wave of the carrier generator it was uploaded to. The starting and stopping of the carrier generators is scheduled independently from the emission of envelopes. When a carrier generator is started, it uses the currently selected frequency and phase values for that generator, similar to the amplitude value for the port.

Sampling is performed simultaneously for all ports on which sampling is actively enabled by the user. This port selection along with the duration of each sampling window should only be set once, and cannot change during the course of a measurement. The windows themselves can however be scheduled to begin at any number of points in time during the measurement.

[1] The specific model is a Xilinx Zynq UltraScale+ RFSoC ZCU111 board, with a Xilinx XCZU28DR-2FFVG1517E System-on-Chip FPGA. See [43] for specifications.

[2] The API documentation is publicly available at `https://intermodulation-products.com/manuals/vivace_manual/index.html`

[3] In Vivace terminology, a *measurement* is the act of sending a pulse sequence and recording the result.

2.5 The Labber instrument control platform

An instrument control platform offers a unified way to control a large number of physical instruments, potentially simultaneously. A program – such as Labber – acting as a frontend to physical laboratory hardware has a range of benefits. For one, it allows users to control the settings of many instruments from a single location. If each instrument were to be controlled through separate means, setting and updating their parameters would be cumbersome. Another benefit offered by such a platform is that it offers a way to coordinate instruments. For experiments involving more than one instrument, one would usually want some way to ensure that their timing is synchronised, that some of their parameters are updated simultaneously, and possibly even that some values of one instrument are based on the values of another. This is easier to accomplish if the instruments in question are managed by a single program instead of completely independent entities.

Managing the properties of hardware instruments through software also has the advantage of making it easy to record instrument settings. Many experiments or measurements require a very specific set of instrument parameters in order to obtain results. If these parameters are set via software, they can be automatically recorded as part of the measurement process, rather than having to save them externally and re-enter them manually if the measurement needs to be repeated later.

The instrument control platform used at QT is known as Labber [23]. It is for this software suite that the virtual instrument representing Vivace is developed. the Labber platform consists of three separate programs that are used to manage instruments, prepare and run measurements, and store results. These are detailed below.

2.5.1 Labber virtual instruments

For a physical instrument to be usable through Labber, it requires a driver definition file, which specifies which parameters it should possess and how the GUI for its virtual counterpart should look. A collection of these definition files are supplied by Labber for a set of common laboratory instruments, but there is also support for modification of existing ones or the creation of entirely new driver definitions. The code that manages the behaviour of a Labber instrument is known as its *driver*. Instrument drivers are written in Python, and make use of Labber's driver API. These drivers usually connect to some physical instrument and communicate with it directly, but they can also be made completely virtual, offering the capabilities of a Labber instrument without relying on actual hardware.

Instruments are managed through Labber's *Instrument Server*. It is through the Instrument Server program that one establishes connections to new instruments and initialises their driver definition files. It is also possible to directly control individual instruments in the Instrument Server. Double-clicking an instrument opens up its GUI, where instrument parameters can be set, the instrument can be turned on or off, and individual data traces can be fetched. Figure 2.11 shows the main window of the instrument server.

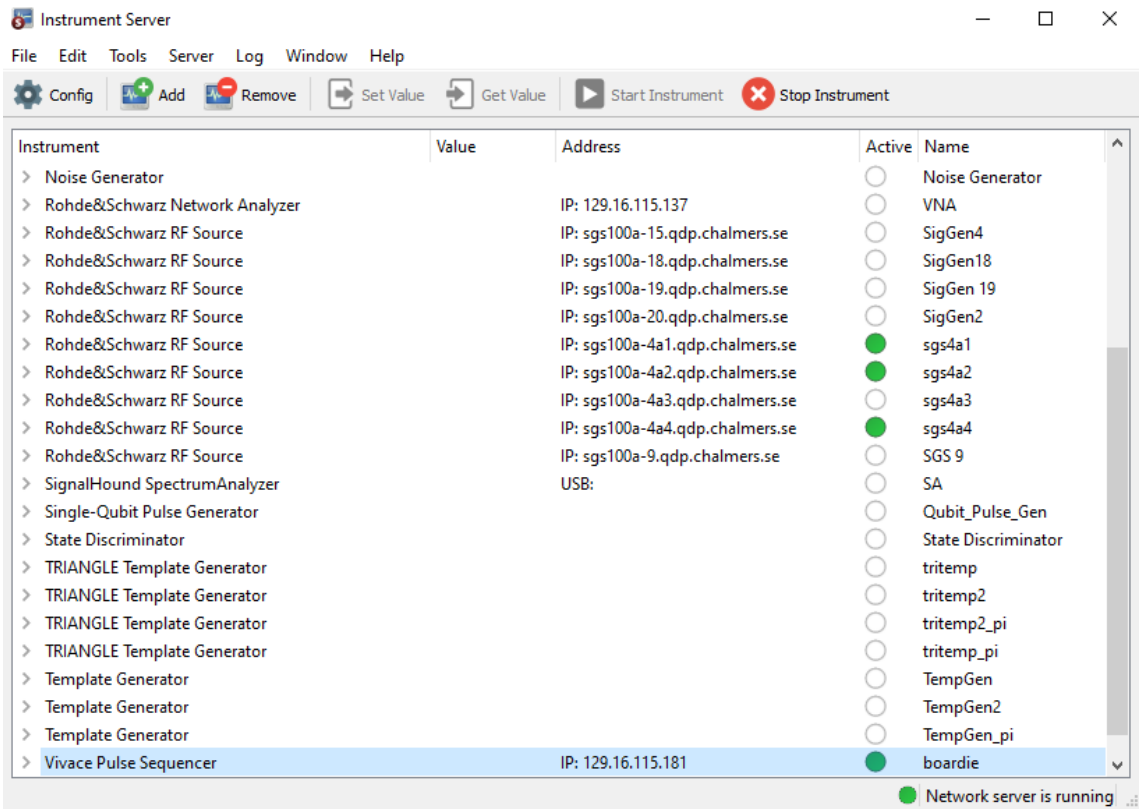

Figure 2.11: A screenshot of Labber's Instrument Server, showing the currently connected instruments.

2.5.2 Managing and running measurements

A *measurement* is the term used in the context of Labber to refer to the running of one or more instruments, which results in some data being produced. These measurements are configured in Labber's *Measurement Editor*. Figure 2.12 shows a typical measurement setup in this program. The editor allows one to define which instruments should be part of the measurement and set initial values of their parameters. In addition, one can enable *sweeping* of certain parameters, wherein they take on varying values over the course of a number of experiment steps. Connections can be set up between instruments, e.g. to let the output of one instrument be given as input to another. Finally, one needs to define which output channels of the involved instruments to sample for results. If the measurement contains any value sweeps, the output channels are sampled once for each value of the sweep, resulting in that many output traces.

2.5.3 Storage and viewing of measurement results

Whenever a measurement has been run by the Measurement Editor, a log file is automatically created. These log files not only contain all the instrument settings for that particular measurement, but also all the data collected during it. With the *Log Browser* and its subprogram *Log Viewer*, one can view graphical representations of that data, with a multitude of options for which sections of data to look at and how it should be presented (see figures 2.13 and 2.14). These settings include the number of plots to view simultaneously, or to view the data as a line graph or a heatmap. When viewing complex

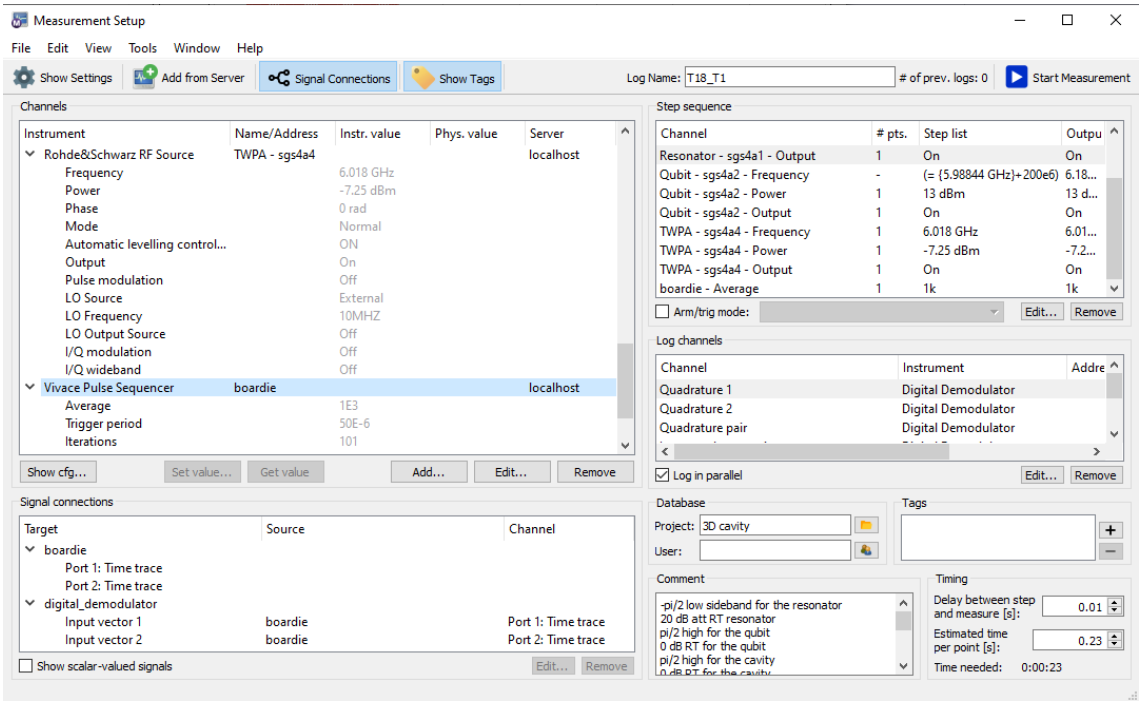

Figure 2.12: A screenshot of Labber's Measurement Editor.

data, Labber can display either the real part, the imaginary part, the angle in the complex plane or the length in the complex plane. There are also features to make adjustments of data values by applying mathematical functions on it like a Fourier transform. The Log Browser can also export graphical plots of selected data values, or into a variety of data formats more easily processed by different programs.

In addition, the Log Browser provides a form of database management of these log files. They can be marked as favourites, have log comments changed or be reassigned to particular users or projects for easier grouping. Since a log file contains instrument settings, it can be dragged from the Log Browser into the Measurement Editor to load all instruments and their settings used to produce that log file.

2.5.4 Measurement automation

Many of the tasks mentioned in the sections above can, to a certain extent, be automated. Labber offers an API for controlling the features of its programs through Python scripts [24]. For instance, one could tell the instrument server to connect to some particular instrument and set its parameters to desired values, forgoing the GUI entirely. One significant feature offered by the API is control over measurements. The script can set up a measurement – either from scratch, or by loading and modifying an existing configuration – and run it, saving the resulting log. Furthermore, measurement logs can be loaded in order to access the data they hold. This means that a chain of measurements can be run with a single script, wherein some result extracted from an earlier measurement's log can be used to modify the configuration of a later measurement.

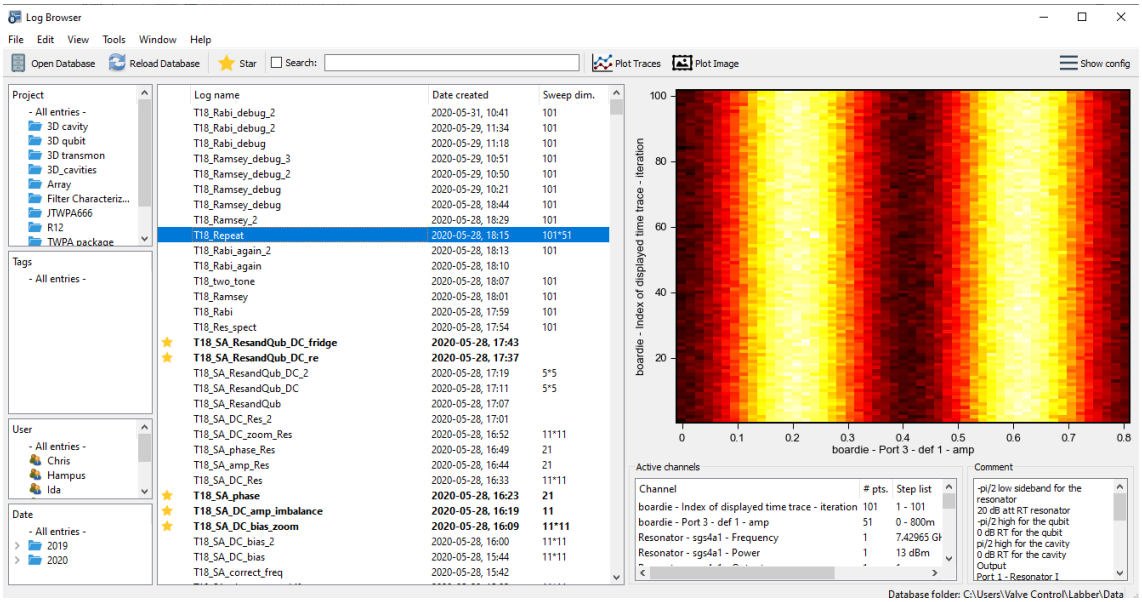

Figure 2.13: A screenshot of Labber's Log Browser. On the right is an image plot of the currently selected log.

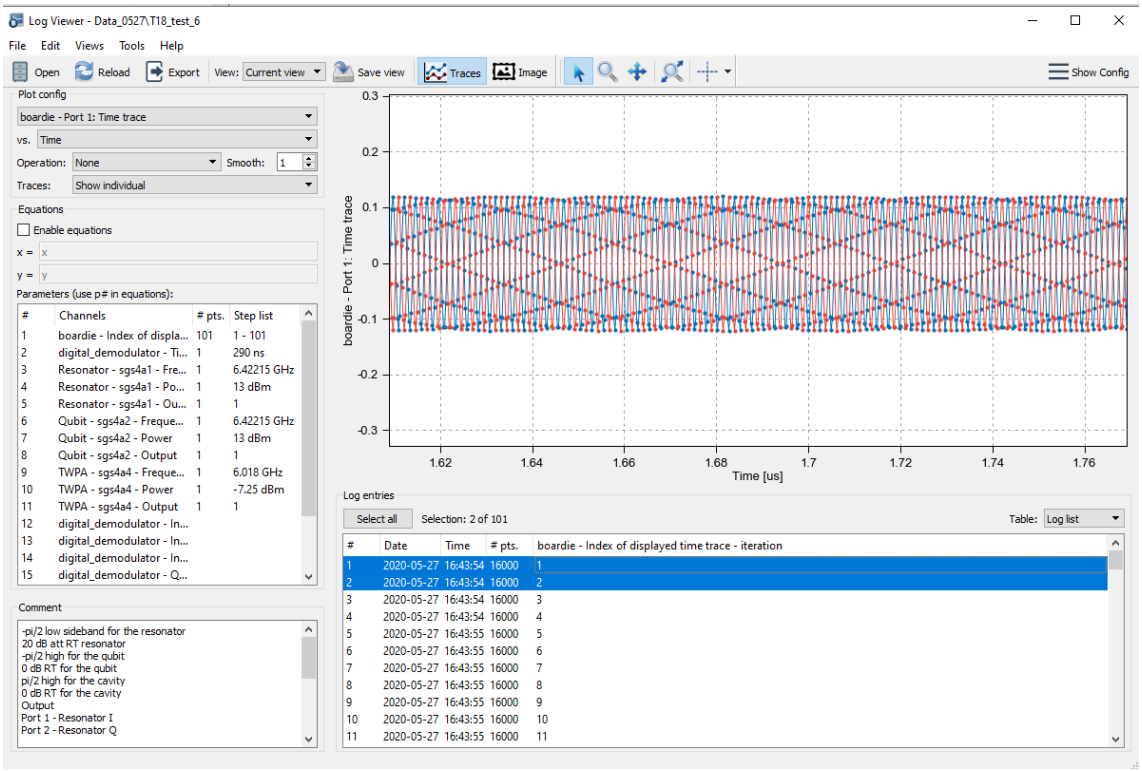

Figure 2.14: A screenshot of Labber's Log Viewer. The traces in the graph are parts of the data from the first two iterations of the measurement sweep, selected in blue at the bottom.

2.6 Time complexity of programs

Within the field of algorithms, one central concept is the notion of *complexity*. Complexity is a measure of how some aspect of an algorithm's computation grows with respect to the size of its input. In Computer science, the study of complexity typically concerns how complex an algorithm is in regards to memory consumption or computation time. Taken more generally, the time complexity of an entire program is a critical issue to consider in order to achieve efficient computation. It is important to note that an algorithm's complexity is constant regardless of the hardware it is run on – within reason. For instance, some algorithms can only feasibly be performed on certain hardware architectures.

At small input sizes, the difference in performance between algorithms of low and high complexity tends to be insignificant. However, as the size of the input grows larger, an algorithm that is too complex quickly becomes impossible to compute, no matter the hardware. Take as an example an algorithm that would need to process each point of input data 100 times versus an algorithm that would instead process the whole input for each data value in it (e.g. comparing each element in a list to all other elements). The first algorithm scales *linearly* with the input data, while the second does so *polynomially*. Already at an input size of 101, the first algorithm would compute faster than the second ($100 \cdot 101 < 101 \cdot 101$), and the difference between them would only grow as the size is increased further.

When it comes to complexity, it is true that "one bad apple spoils the bunch", i.e. the total complexity of a chain of algorithms is equal to its most complex part. This is due to complexity being measured in terms of polynomial degrees; less complex computations are considered negligible compared to the time incurred by the most complex computation. The driver developed for this project is supposed to be part of a stack of programs that can execute a quantum algorithm. The *raison d'être* of quantum algorithms is that they should be able to achieve a result faster than a traditional computer. It is thus important that no part of our driver – that must generate the quantum program – runs at a complexity so high that it invalidates the speed at which the algorithm itself computes the result.

3

Implementation

In this chapter, we present a more thorough explanation of the virtual instruments developed during this project. We begin with a walkthrough of the feature set and operating instructions for the instruments developed during this project, in order to give an understanding of their purpose and functionality. We then move on to present a more in-depth look at the code and data structures of the instruments.

3.1 The Vivace Pulse Sequencer instrument – Feature overview

The Vivace Pulse Sequencer (ViPS) is the main instrument used to communicate with Vivace. It is used to set up a sequence of pulses of different shapes, which are then outputted by the FPGA board at the specified times. Results can then be fetched from the instrument based on the sampling windows specified in the pulse sequence.

All of ViPS's settings can be configured through the use of its GUI; an example of this GUI can be seen in figure 3.1. The settings are divided into sections, seen in the leftmost list, and then into groups in the main body of the GUI. These individual settings are known as *quants*. Some quants are only visible depending on the state of other quants, such as the group for envelope number 2 only being visible if the quant *No. of envelope templates* is set to 2 or higher. Output from ViPS is given in the form of *traces* (lists of connected points in the XY plane), the type of which can be selected in the dropdown in the top bar. Only one trace can be viewed at a time through the GUI, and is displayed on the graph to the right if one presses the *Get new trace* button to the right of the dropdown. The data gathered during the sample windows configured on the board is accessed through the eight traces named *Port X - Time trace*, with $X = 1 \dots 8$, corresponding to the eight available input ports of the FPGA. In addition to traces containing measurement results, there are also traces created digitally for previewing ViPS's output. These can be used to more easily see the effect different settings have on the pulse sequence.

The following subsections go through the different sections of ViPS and what effect they have on the pulse sequence.

3.1.1 The *Envelopes* section

In this section the user can create templates for pulse envelopes, which can be re-used for multiple pulse definitions. Figure 3.1 shows the settings available in this section. The

user selects the active number of templates using the dropdown labelled *No. of envelope templates*, and is then able to configure that number of distinct templates by using the entry fields that appear. The *Envelope shape* quant lets the user choose from several common shapes. Some of these have extra parameters governing their form, the quants of which become visible when that particular shape is selected. The shapes available for selection are:

Square A normal square envelope that has the value 1 for its entire duration.

Long drive A type of square envelope handled in a special manner by Vivace, which can vary its duration during a single measurement. Unlike other types of envelopes, which take up multiple template memory slots in Vivace if they exceed the maximum duration of 1022 ns, the Long drive can be arbitrarily long without taking up more than a single template's worth of memory. Figure 3.1 shows a scenario in which template 2 is a Long drive configured with an initial duration of 500 ns that grows by 20 ns for each iteration of the pulse sequence.

If the user does not wish for the Long drive to start and end with the value 1 as a normal square pulse does, they can toggle *Use gaussian rise and fall* which will give the envelope gaussian-shaped (see below) rising and falling edges. The user specifies the duration of the leading and trailing edges.

Sin^2 The shape of a sine wave defined from 0 to π, squared.

Sin^P The shape of a sine wave defined from 0 to π, raised to the arbitrary non-negative power P, specified by the user.

Sinc The shape given by the sinc function, defined as $\mathrm{sinc}(x) = \frac{\sin(\pi x)}{\pi x}$ from $-x$ to x, where x is given by the user.

Triangle A triangle envelope, consisting of a linear increase from 0 to 1 for the first half of the envelope's duration, followed by a linear decrease from 1 to 0 for the second half.

Gaussian The shape given by the probability distribution function of a Gaussian (normal) distribution, cut off at $-x\sigma$ and $x\sigma$, where x is given by the user.

In addition to the above shapes, the user has the option of using *Custom* envelope templates. If so, they need to generate the desired shape with another Labber instrument, then pass the resulting waveform on to ViPS.

Besides shape, each unique template needs a duration. Since the envelope templates are designed to be used with multiple different pulses, their definitions do not include a starting time. However, because Vivace can only start pulses at even nanosecond values, ViPS allows users to "pad" templates by inserting up to 7 leading zeroes in their envelope data. Each of these zeroes correspond to a delay of the envelope's start by 0.25 ns. This effectively enables pulse start times to be specified with a *temporal resolution* of 250 picoseconds. However, since the padding becomes part of the template, the padding is present for all pulses that use that envelope. If the user only wants an envelope to be

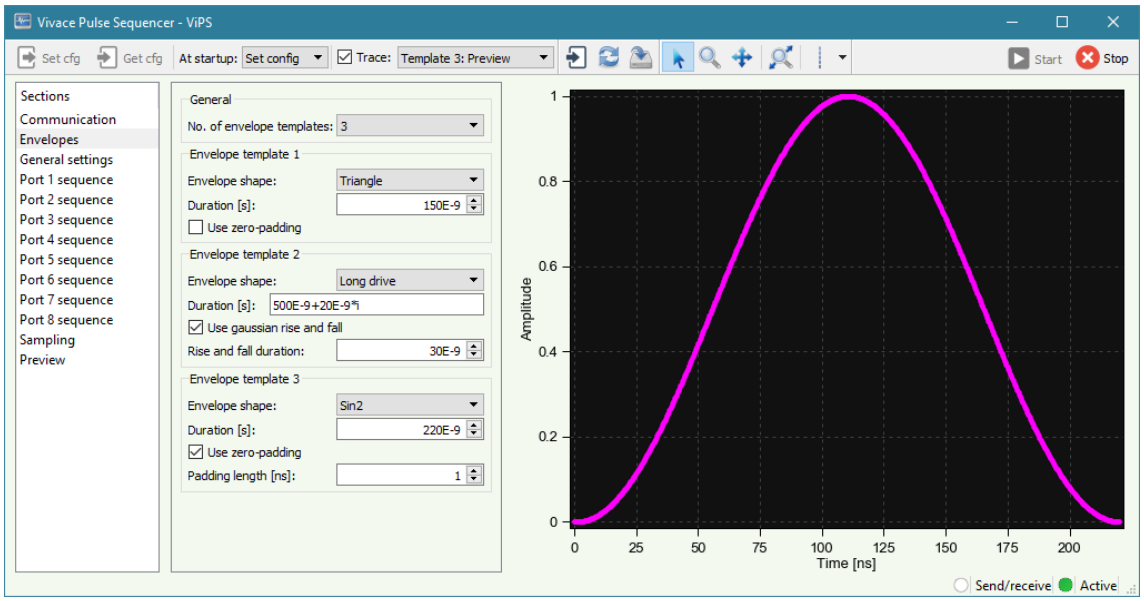

Figure 3.1: A screenshot taken of the *Envelopes* section of the ViPS instrument. On the right, the preview for template 3, a Sin^2 envelope, is displayed.

padded for some pulses, they would need to upload two separate templates to Vivace.

The number of each template is used as its identifier in the instrument, both when using its envelope on a pulse (see figure 3.3) and when previewing it (as in figure 3.1).

3.1.2 The *General settings* section

As the name implies, the General settings section lets the user perform general instrument configuration that affects the whole measurement.

When setting up a pulse sequence, all of the user's pulses are defined within a single *trigger period*, which is a user-set period of time. Along with defining the shapes, lengths, frequencies etc. of the pulses to be emitted, the user needs to specify when in time these pulses should be emitted within this trigger period. Copies of such a trigger period sequence are outputted one after another as many times as indicated by the value of the quant *Iterations*. In each of these iterations, the defined sequence is modified in the following ways:

- If the user has set up a pulse to *sweep* over amplitude, frequency or phase, it uses the sweep value corresponding to that particular iteration.

- The iteration index `i` can be used as a variable within certain fields used to set time and duration, which enables a flexible way of sweeping over these values. During the first iteration the parameter is equal to the base value of the setting, and for every iteration thereafter it is increased by the coefficient of the `i` variable (also known as the delta). For example, the start time string "`24e-9+6e-9*i`" has a base time of 24 ns and a delta of 6 ns, and it would thus be outputted at the time 24 ns in the first iteration, 30 ns in the second, 36 ns in the third, etc. See section 3.5.2 for details.

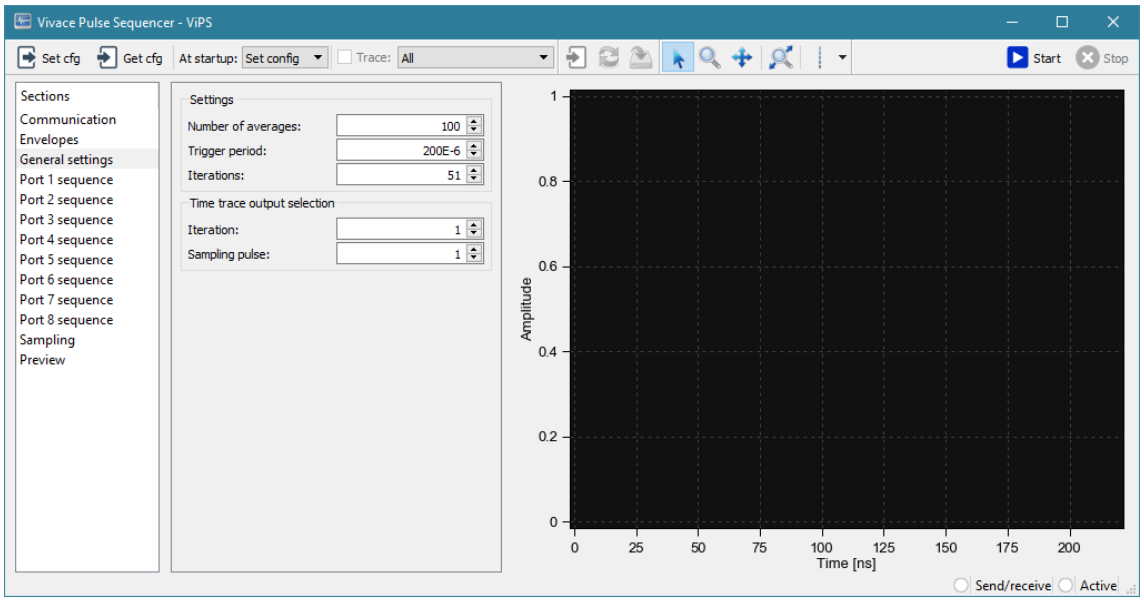

Figure 3.2: A screenshot taken of the *General settings* section of the ViPS instrument.

The sequence of these iterations makes up the complete pulse sequence. The user can then set a value in the *Number of averages* quant to repeat the whole pulse sequence a number of times, and average the results of these repetitions through interleaved averaging (as explained in section 2.2.3). For example, if the user defines 2 sample pulses per iteration on a single port and chooses 5 iterations and 15 averages, there are a total of 10 different averaged time traces available after running one measurement (2 windows $\times$ 5 iterations). Labber instruments are only capable of outputting a single trace at a time, so which one of these time traces the user receives when they request a trace from the corresponding port is dependent on the quants in the *Time trace output selection* group. If one wants to see the (averaged) sampling data of the second sampling window during the third iteration, then one would set the *Iteration* quant to 3, and *Sampling pulse* to 2. To obtain traces corresponding to several different iterations or sampling windows from a single measurement, the user needs to set up a Labber measurement that sweeps over these *Time trace output selection* quants.

3.1.3 Pulse sequence setup

The main setup of the individual pulses of the sequence is done through the *Port X sequence* sections corresponding to each output port. The standard way of defining pulses to a port is to set the port's *Mode* to *Define*. The user can then create up to 16 different pulse definitions, each with a number of quants which affect its behaviour:

Start times: The times at which the pulse in question should be outputted. These times can be given with both base and delta values. If multiple start times are given comma-separated, then a pulse is outputted for each time.

Envelope: The index of the envelope template to use for the pulse, defined in the *Envelopes* section (see section 3.1.1 for more information). This also determines the pulse's length.

Pulse repeat count: An option to output extra identical pulses directly following each pulse created by this definition. In effect, this multiplies the length of the total pulse emitted.

Sine generator (DRAG): There are two sine generators available per port to use for carrier waves. By setting up pulses on different generators simultaneously, one can add together pulses with potentially differing frequency/phase values. If this value is set to *None*, then only the envelope is outputted for this pulse, not modulated with the carrier wave. There is also an option to define the pulse as a DRAG pulse [26], which makes the pulse use both carrier generators of a pair of ports (see section 3.5.10 for implementation details).

Amplitude scale: Pulses have their amplitude multiplied by this value, in the range $[-1, 1]$.

Carrier frequency: What frequency the carrier wave should have during this pulse. If this value is set to zero, it has the same effect as setting *Sine generator* to *None*.

Phase: By setting a value in this quant, one can shift the phase of the carrier wave by up to $\pm 2\pi$ relative to the synchronised phase for this pulse's frequency. See figure 2.1 for a demonstration of the effects of phase shifting, and section 3.5.7 for details on phase synchronisation.

Sweep parameter: This dropdown has the options *None*, *Amplitude scale*, *Carrier frequency* and *Phase*. If a non-*None* value is chosen, the corresponding quant of the previous three in this list are hidden. That parameter instead has its value decided by a few new quants that appear at the bottom of the pulse definition. These quants define a range of values that the pulse uses throughout iterations.

Sweep formats: Sweeps can be defined in three different ways.
With *Linear: Start-End*, the value in the *Start* quant is used for the first iteration, the *End* quant for the last iteration, and a linear interpolation of those values are used for iterations inbetween.
With *Linear: Center-Span* the range is instead defined by the values in the *Center* and *Span* quants. The range starts at *Center* minus half *Span*, and ends at *Center* plus half *Span*.
With the *Custom* sweep format, the user can enter a list of arbitrary comma-separated values, one for each iteration.

If multiple pulses on the same port have the same carrier frequency, their phase is automatically adjusted so that they are synchronised with the phase of the first pulse (see section 3.5.7 for a more in-depth look at how this is implemented).

The *Mode* quant can also be set to *Copy*. This hides all quants related to defining pulses, and instead displays a dropdown where the user gets to choose which other port's pulse definitions to copy. The user can also apply a *Phase shift* and *Amplitude shift* to the port, which adjusts all copied pulses' phase and amplitude by the specified value. This is, for instance, useful for defining a Q signal to use in IQ modulation (as explained in section 2.1.4).

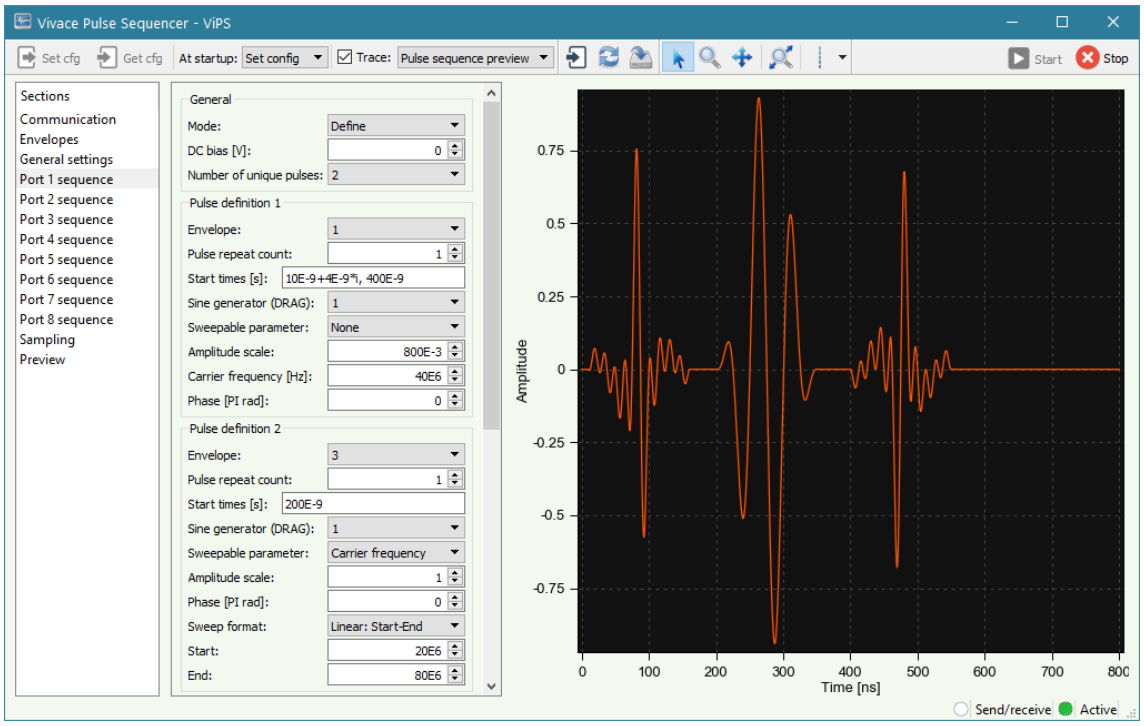

Figure 3.3: A screenshot taken of the *Port sequence* section of the ViPS instrument.

3.1.4 The *Sampling* section

Besides sending pulses, ViPS can also set up *sampling pulses*. These are pulses in name only, as a "sampling pulse" is simply a span of time in which Vivace records incoming pulse data. These time spans are also known as *sampling windows*. Compared to the outputting of pulses, there are significantly fewer parameters involved with sampling pulse setup (see figure 3.4). As with pulses in the pulse sequence setup, the user can give one or more starting times, which can include time deltas. However, since sampling is carried out simultaneously for all ports it is enabled for, there are no port-specific time settings. The user can also only give a single *Duration* value, which applies to all sampling pulses.

3.1.5 The *Preview* section

Besides individual envelope templates, it is possible to preview an entire sequence of pulses, as demonstrated in figure 3.3. All configuration of this *Pulse sequence preview* feature is handled in this section of the instrument. Since only one trace can be displayed at a time, the user has to select which port's sequence they wish to preview, as well as which iteration's pulses should be displayed. By default, the preview shows the pulses on a timeline from 0 to the end of the trigger period. Since the pulses might take up significantly less time than that, the user has the option to adjust the span of this timeline by toggling *Enable preview slicing*, which reveals two quants allowing them to set their own start and end times. The user also has the option to display any sampling windows they have set up in the preview. These are represented by square pulses with an amplitude of −0.1, and hide any outgoing pulses that they happen to overlap. We are forced to hide the overlapping pulses due to Labber only being able to display one (or every) trace at a time within the instrument GUI.

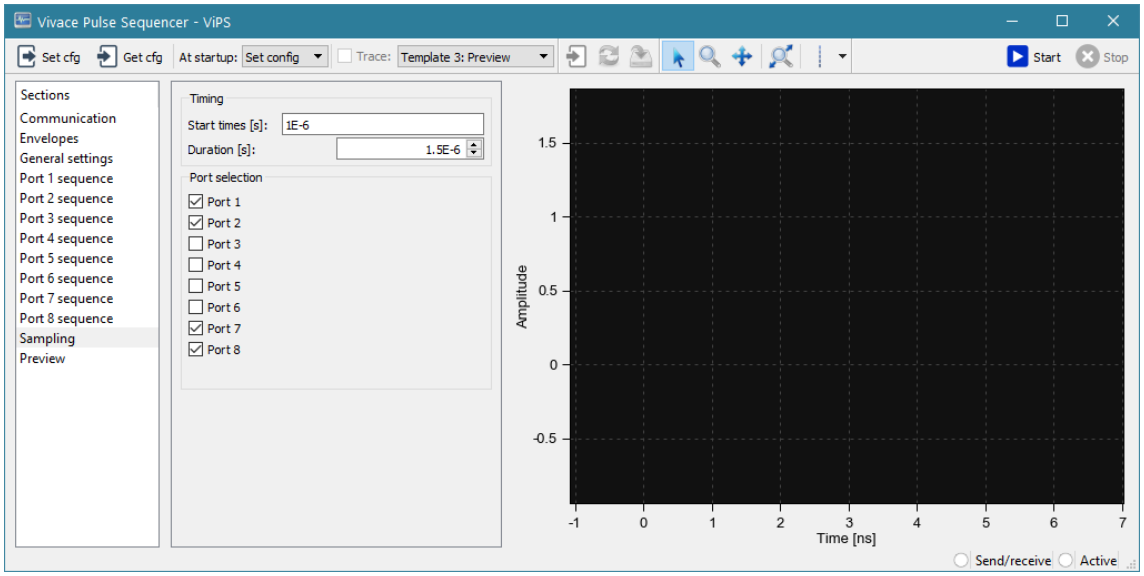

Figure 3.4: A screenshot of the *Sampling* section of the ViPS instrument.

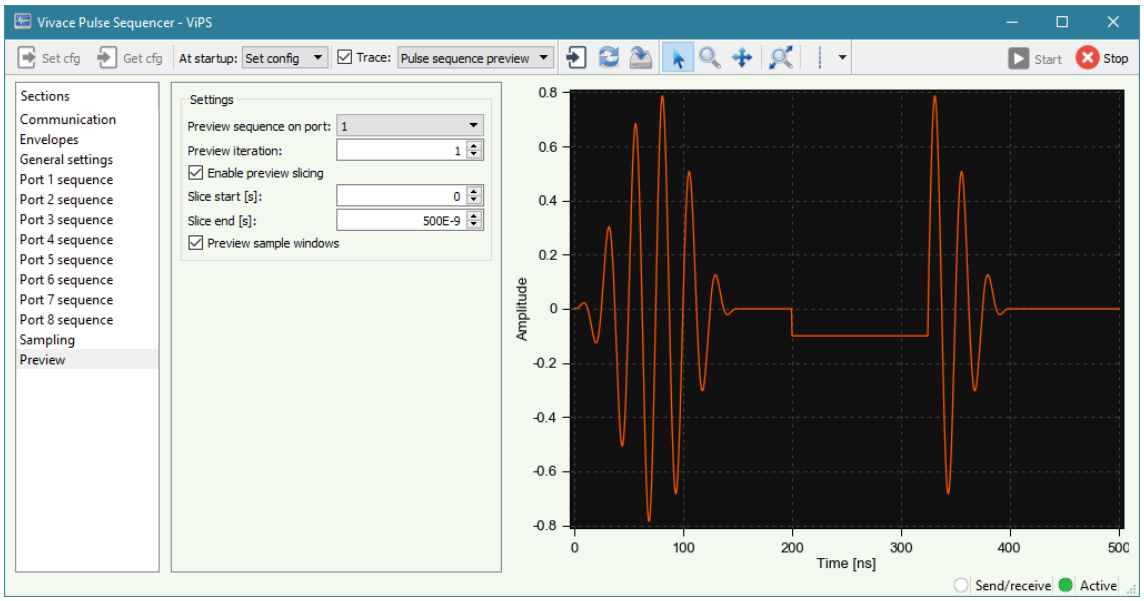

Figure 3.5: A screenshot of the *Preview* section of the ViPS instrument. The preview shows two pulses, both modulated with Sin^2 envelopes. The second pulse is partially obscured by a sampling window that starts at 200 ns and runs for 125 ns.

3.2 The Digital Demodulator instrument – Feature overview

The traces sent back to Vivace from the QPU (ViPS's output traces) have the same carrier frequency as the pulse sequences emitted by Vivace. However, the information that really matters in these signals is the pulses' envelopes. As part of the requirement that data returned from the board should be presented in a user-friendly way, we created the Digital Demodulator instrument for the task of extracting envelope information from ViPS's output. See section 2.1.4 for details on IQ (de)modulation. The decision to move this functionality to a separate instrument was made for two reasons. Firstly to not bloat the ViPS instrument with additional (optional) settings, and secondly because the functionality to demodulate qubit readout data is useful for other architectures besides Vivace. All Labber instruments use the same data structures for their data, and Labber supports the linking of one instrument's output to the input of another instrument. This way any Labber instrument can feed its readout data into the Digital Demodulator and access all of its functionality. Additionally, Labber supports the creation of an arbitrary number of instances of virtual instruments. This means that it is possible to create multiple Digital Demodulators with separate settings and run them in parallel within a single measurement.

The Digital Demodulator has two inputs for traces, and offers two kinds of output. The first output is *Quadrature*, which demodulates the input trace(s) and gives out an I and Q signal as real and imaginary components of a complex vector. The other output choice is *Integrated trace*, which demodulates the input as above to reduce the traces to pure envelopes before calculating the area under the envelopes using integration. The resulting values for the I and Q components are, again, given as real and imaginary parts of a complex number. The data is given on complex form partially because Labber cannot otherwise hold two traces in a single vector, but also because Labber offers several ways to display complex vectors and numbers (see section 2.5.3). All the available output trace types can be seen in figure 3.6.

There are some settings available to the user which affects how the output is calculated. They are the following:

Sampling frequency: What sampling rate the instrument expects the input data to be. This affects the filtering negatively if it does not match the actual sampling rate.

Digital downconversion frequency: The frequency to demodulate with. Should match the frequency of the input pulse's carrier wave.

Phase offset: An optional rotation of the IQ vector in the complex plane.

Filter type: The type of lowpass filter to use for removing unwanted frequencies from the input. Available choices are *Butterworth*, *Chebyshev type I*, *Chebyshev type II*, *Elliptic (Cauer)* and *Bessel/Thomson*. While they all work in a similar manner, these exhibit slight differences in behaviour and the additional parameters they take.[1]

[1]The Digital Demodulator uses the *Scipy* [44] Python library for its filter implementation. For a comparison of filter types as they behave in this library, see the reference pages for `butter`, `cheby1`, `cheby2`, `ellip` and `bessel` available at `https://docs.scipy.org/doc/scipy/reference/signal.html`.

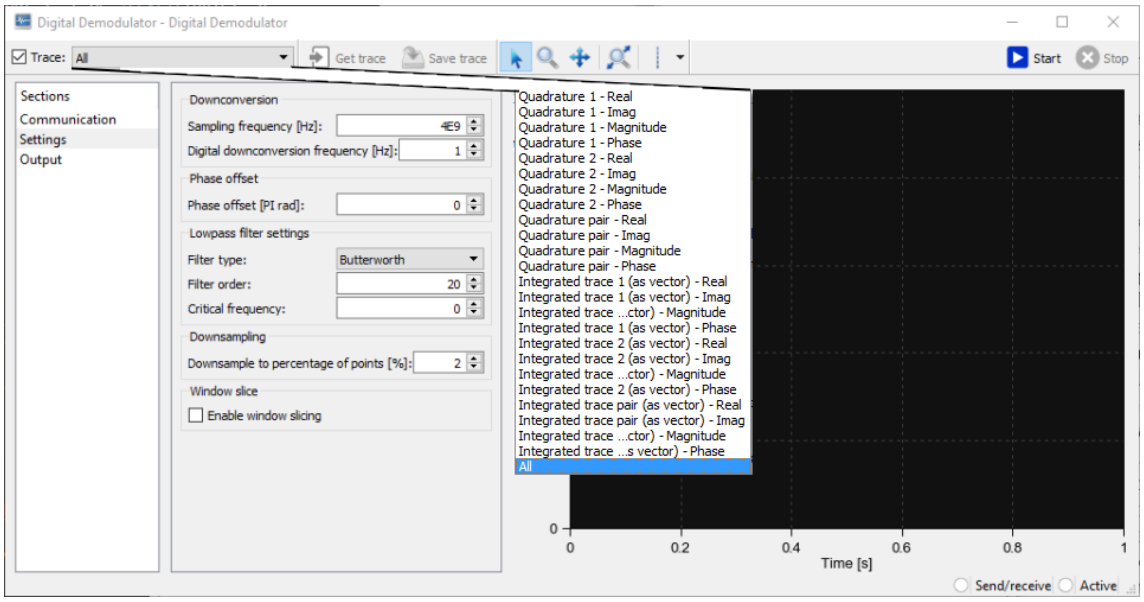

Figure 3.6: A screenshot of the *Digital Demodulator*, with the dropdown showing the available output types edited into the image so as not to block the view of the settings.

Filter order: This parameter is present for all filters. Higher order filters are more precise at the cost of higher computational complexity.

Critical frequency: This parameter is also present for all filters. It determines the highest frequency that should be allowed through the filter.

Downsampling: After demodulation, the shape of the trace can often be well represented even with a drastically reduced number of points, saving on further computation time and storage. The user can set a percentage of the input points to downsample the results to. Downsampling creates that amount of points linearly spaced within the original data's time bounds, with values interpolated from the nearest points in the original data.

Window slicing: With this option enabled, the user can specify a section in time of their input for which they want the integrated trace to be calculated. Has no effect on the Quadrature output.

3.3 The Pulsed Comb Generator instrument

There are certain kinds of pulses that cannot be generated using ViPS alone. This includes *combs*, combinations of sine waves at a variety of different frequencies. The two sine generators per port offered by Vivace are insufficient to produce combs, so we designed a separate instrument to handle these. The Pulsed Comb Generator (PCG) instrument, as illustrated in figure 3.7, lets its user combine as many as 20 sine waves of different frequencies, phases and amplitudes. If they wish, they have the option of modulating the combination with an envelope.

The PCG instrument is what is known as a *signal generator*; it is not connected to any

hardware and only exists to generate digital signal data. The trace it generates can be fed into any instrument that accepts such data as input. This means that it can be given to the *Custom envelope* input of ViPS. By doing so, and letting ViPS output a pulse using such an envelope with a carrier frequency of 0 Hz, one can simulate pulses created by many sine generators. However, this prevents the use of features such as phase synchronisation or sweeping carrier frequency during a measurement, since the Vivace sine generator is turned off. Once an envelope has been uploaded to Vivace, it cannot be modified for the duration of the measurement. It should be mentioned that Vivace and ViPS are planned to support more flexible usage of combs in the future.

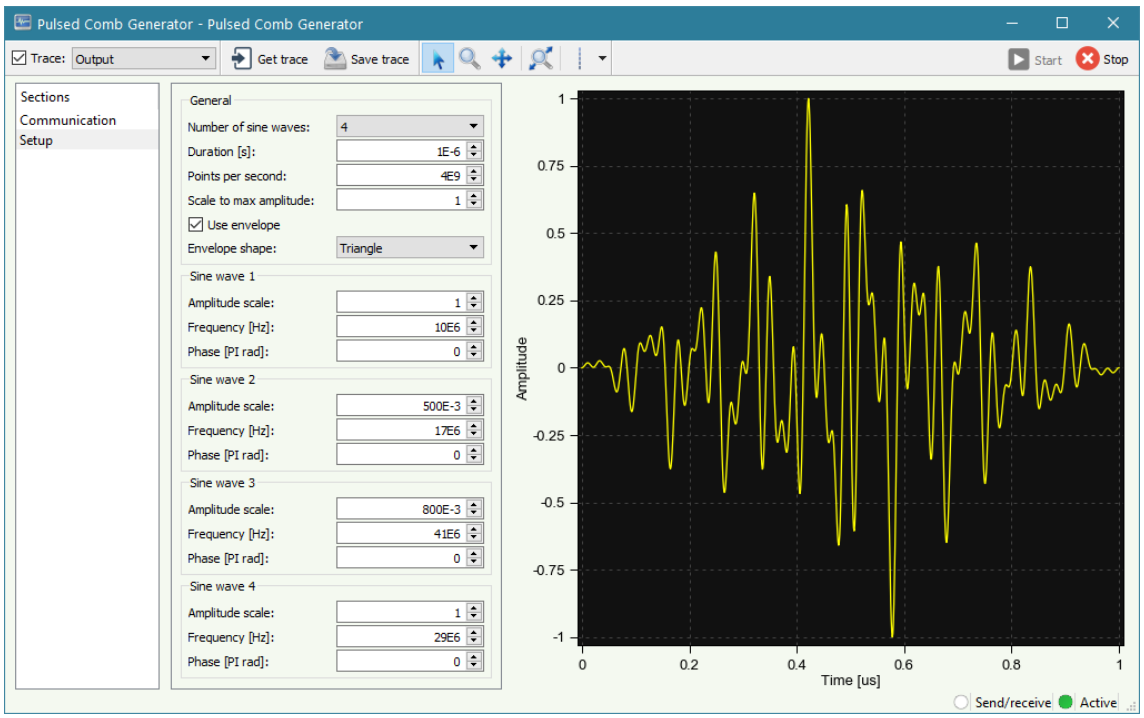

Figure 3.7: A screenshot of the PCG instrument. Four sine waves at different frequencies and amplitudes are combined and modulated by a triangle envelope.

3.4 Instrument GUI generation

All Labber instrument GUIs are created through so-called *definition files*, one per instrument [45, Sec. 12]. Labber processes these files from top to bottom, creating the GUI quant by quant. Whenever it reads a line of the form `[Name]`, a new quant is initialised with that name as its unique identifier. There are a number of keywords that when processed apply a setting to the most recently initialised quant, such as `label` that gives the quant a label in the GUI, or `datatype` which specifies its data type (and shape of its input field). An example of how definition code is converted into GUI elements can be seen in figure 3.8.

It quickly becomes obvious that writing these files by hand is not feasible once the instrument requires more than a handful of different settings. ViPS contains roughly 2500 quants, each requiring on average about 14 lines of code, resulting in a total of 35 000 lines. This is mostly due to the pulse definitions; quants cannot be reused, and so every

related quant has to be rewritten for each definition and every port. Our solution is to construct a Python program that can generate instrument definition files for us. The program has functions corresponding to the different keywords available in the Labber GUI API. These functions take keyword values as input, then prints the corresponding line(s) into a definition file. This allows the use of Python control structures and variables when creating the GUI, enabling easy creation and modification of several quants at the same time.

For example, changing the name of a section in the instrument can be done in a single line of code in the generator instead of a line in every quant contained within that section. Figure 3.9 shows a part of our generator program that constructs the same quant definitions as in figure 3.8, with much fewer lines of code. As a reference, the file that generates the 35 000 line long ViPS definition is 366 lines long.

The generator is generic, in the sense that it is in no way bound to ViPS or the Digital Demodulator. It can be used to create arbitrary Labber instrument definition code, as it gives access to all features of Labber's definition framework.

```
[Preview port]
label: Preview sequence on port
datatype: COMBO
group: Settings
section: Preview
combo_def_1: 1
combo_def_2: 2
combo_def_3: 3
combo_def_4: 4
combo_def_5: 5
combo_def_6: 6
combo_def_7: 7
combo_def_8: 8
[Preview iteration]
label: Preview iteration
datatype: DOUBLE
group: Settings
section: Preview
def_value: 1
low_lim: 1
set_cmd: int
[Enable preview slicing]
label: Enable preview slicing
datatype: BOOLEAN
group: Settings
section: Preview
tooltip: This will let you specify which
        segment of the pulse sequence to preview.
[Preview slice start]
label: Slice start
datatype: DOUBLE
group: Settings
section: Preview
unit: s
low_lim: 0
state_quant: Enable preview slicing
state_value_1: True
```

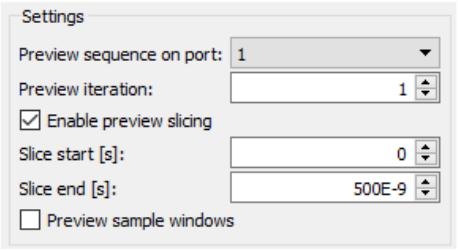

Figure 3.8: The instrument definition code on the left generates the first four quants of the GUI on the right once parsed by Labber.

```
def section_preview():
    section = 'Preview'
    group = 'Settings'

    gen.create_quant('Preview port', 'Preview sequence on port', 'COMBO', group, section)
    gen.combo_options(*[i for i in range(1, N_PORTS+1)])

    gen.create_quant('Preview iteration', 'Preview iteration', 'DOUBLE', group, section)
    gen.default(1)
    gen.limits(low=1)
    gen.set_cmd('int')

    gen.create_quant('Enable preview slicing', 'Enable preview slicing', 'BOOLEAN', group, section)
    gen.tooltip('This will let you specify which segment of the pulse sequence to preview.')

    gen.create_quant('Preview slice start', 'Slice start', 'DOUBLE', group, section)
    gen.unit('s')
    gen.limits(low=0)
    gen.visibility('Enable preview slicing', True)
```

Figure 3.9: A snippet of Python code that generates the quant definitions seen in figure 3.8 with the help of our generator program.

3.5 ViPS driver architecture

The reader should now be familiar with the feature set of ViPS. This section details the construction of the ViPS's driver, which controls its behaviour and handles communication with Vivace. Readers of this section are assumed to have a basic understanding of programming concepts.

3.5.1 The Labber instrument driver API

A Labber instrument can technically run without an accompanying Python driver. The instrument created from the definition file can connect directly to physical instruments that support communication through simple text commands. However, for more advanced functionality, a custom driver is needed as a middleman between the Labber instrument and the physical instrument. To avoid confusion, Labber virtual instruments are from here on referred to as "instrument", and hardware as "physical instrument". The driver is interacted with through four functions. `performOpen()` and `performClose()` are executed whenever the user starts or stops the instrument through the GUI. These methods are meant to perform connection to or disconnection from a physical instrument or other setup and cleanup code. The next function is `performSetValue()`, which is called whenever a value is entered into a quant in the instrument. That value is sent as a parameter to the driver, along with info about which quant is being updated. However, what the driver returns from this function is what is actually saved in the instrument (and shown in the quant field for the user). The last function is `performGetValue()`, and is called whenever the user requests a trace. The driver gets info about which quant is requested through the function, and should then procure and return it. The driver can also directly read or set the value of any quant in the instrument, no matter which quant the user currently updated or requested.

3.5.2 Custom datatypes

Some of ViPS's quants require input values of datatypes that are not natively supported by Labber. For instance, Labber only has two quant datatypes for numerical input: `REAL` and `COMPLEX`. However, there are quants in ViPS that should logically only accept integers (such as *No. of averages*) or those that should accept non-integer but discrete values (such as the length of zero-padding for envelope templates, which goes in increments of 0.25 ns). While cases like these are trivial to resolve using numerical Python operations in the driver, other quants require more work.

One such quant type is arbitrarily long lists of values, used to set up custom sweep sequences for pulse amplitude, frequency and phase. Although these lists are made up of numerical values which themselves are supported by Labber, lists of them are not. In order to enable list input, these quants are declared to be of the `STRING` datatype, which accepts sequences of arbitrary characters. Allowing any character means that users can use commas to separate multiple values, but also means that the driver has to check the string for invalid characters and perform manual parsing of numerical values before storing them internally.

Labber's `STRING` datatype is also utilised for the quants that are used to set the duration of Long drive envelope templates. Since these envelopes can have their duration shift per iteration, the quants should allow one to enter either a base time value, a delta value, or both. The specifics of how these values (dubbed "time strings") can be formatted are seen in figure 3.10. As is the case with the lists, these entries need to be parsed by the driver in order to ensure that they are properly formatted.

Lastly, the quants used to set the starting time for pulses should allow for an arbitrary amount of time strings, and thus also need to be declared as `STRING`. The driver parses these using both the techniques described above.

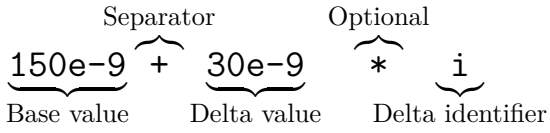

Figure 3.10: An example of a value string used to declare a time value that shifts with each iteration. Note that while either the Base or Delta values can be left out, at least one of them is required for the string to be valid.

3.5.3 Driver structure overview

Outside of `performGetValue()`, the ViPS driver does not do a lot. It does not establish a connection with Vivace through `performOpen()` and `performClose()`, since one is supposed to create a new connection to it for every measurement. As for `performSetValue()`, its primary purpose is to adjust certain quant values or ensure that they are correctly formatted, as described in section 3.5.2.

The bulk of the driver program's code is only executed with `performGetValue()` as an entry point. If the driver already has fetched results for the current settings of the instrument, then the requested trace is simply returned to the user. If not, then a pulse sequence

needs to be created and sent to the Vivace for measurement. As a high level overview, the process from receiving a trace query to returning measurement results consists of the following sequence of steps:

1. Connect to Vivace.
2. Fetch general settings from the instrument.
3. Construct envelope templates.
4. Collect pulse definitions and send envelopes to Vivace.
5. Perform port copying.
6. Construct amplitude, frequency and phase tables for all ports, and send these to Vivace.
7. Schedule changes of sine generator frequency/phase values.
8. Schedule envelope emissions and amplitude changes.
9. Tell Vivace to run the measurement and return results.

3.5.4 Envelope template construction

Reusable envelope templates exist in Vivace independent of individual pulse definitions, in a similar manner to how they are presented through ViPS. The first step of building these templates in Vivace is to read the definitions set up by the user in ViPS's *Envelopes* section (detailed in section 3.1.1). Based on the shape, duration, and other parameters that the user has given values, a list of height values that form the requested shape is constructed for each such definition. If the user has set up any custom templates, these are given in Labber's `TraceDict` format which already contain such shape lists. However, since other trace-generating instruments might not operate in the $[-1, 1]$ scale that Vivace uses, imported custom templates are rescaled to ensure compatibility.

While the shape lists are the foundation for constructing templates, they cannot be sent to Vivace by themselves. Unlike the templates of ViPS, Vivace's templates are defined per port and sine generator. Thus, shape lists are stored in the ViPS driver until it is time to process the pulse definitions.

3.5.5 Pulse definition construction

Once the shapes of all envelope templates have been generated, the driver iterates through all pulse definitions in the *Port X sequence* sections of the instrument and handles them one by one. For each definition and start time, we create a local representation of a pulse and store it in a large list containing all pulses to be emitted. These pulse definition objects contain values such as their start times; their port; their carrier generator index; amplitude, frequency and phase values for every iteration; which envelope index they use etc. It is also at this point we know that a certain envelope template is used on a specific port and sine generator, so it is uploaded to Vivace. If a later processed definition uses

the same template on the same port and carrier, we do not upload it to Vivace again since we can reuse the one from earlier.

Separately, sampling pulse definitions are also created and added to the list of pulses, based on the sample times defined in the *Sampling* section of the instrument. These definitions contain much less information than the output pulses: only their start time and a tag showing that they are sampling pulses. The duration of sampling pulses has to be a constant value in Vivace, so the instrument's sampling duration value is simply sent once instead of being stored in the sampling pulse definitions.

3.5.6 Copying of ports

Once all user-defined pulses have been collected and stored, the driver can start setting up pulse definitions on those ports that have been set to copy from another. The port that is copied and the port that copies are referred to as the *source* and *target* ports henceforth. For any port set to copy mode, the driver looks through the list of pulse definitions to find those that belong to the source port. Every such definition has a copy made and placed in the pulse definition list. The copy gets its port value updated to that of the target port, and its phase and amplitude values are shifted according to the shift values in the instrument. The pulse definition list is thus complete, and contains the correct information about every pulse that should be emitted.

3.5.7 Setting up look-up tables for amplitude, frequency and phase

Pulses in Vivace consist of two parts: carrier waves and envelopes. These can be outputted independently of each other. The amplitude, frequency and phase parameters of pulses are not given directly to Vivace as part of its output commands; instead, these parameters are stored in structures known as *look-up tables* (LUTs) that are prepared beforehand. Every port holds two LUTs: one for amplitude values and one that contains the frequency and phase values for each of the port's two sine generators. Both kinds have a pointer that indicates their currently active value (for the frequency/phase LUTs, a single "value" is made up of the two pairs of frequency and phase values). When a port's sine generators are started, their frequencies and phases are determined by the currently selected value pairs in their LUT. Similarly, when an envelope modulates an active carrier, the resulting pulse is scaled by the currently selected value of its port's amplitude LUT.

Each port's LUTs are set up in the ViPS driver by going through a chronological list of the pulse definitions on that port. To construct the amplitude LUT, the driver simply records each pulse definition's amplitude value (as long as that value has not been previously recorded), and finally sends the list over to Vivace. But when assembling the frequency/phase LUTs, special care needs to be taken due to the sine generators sharing a position in their LUT. We need to know at which points in time the carrier generators are to be stopped and started in order to step to a different pair of frequencies and phases for both generators. We also need to know what frequencies and phases are required during each of those periods. It is not as simple as taking the user-given phase value and entering that into the LUTs; pulses that share frequencies should be synchronised in phase to each other, and the user-given phase parameter should shift the pulse relative to that synchronised phase. This user-given phase shift value is referred to as relative phase. Take the following scenario as an example of how the frequency and phase LUTs are created, where the user has set up four pulses with the following parameters on one port:

Pulse no.	Start time	Frequency	Relative phase	Sine generator
1	10 ns	90 MHz	0	1
2	40 ns	90 MHz	0	1
3	70 ns	160 MHz	0	2
4	100 ns	160 MHz	π	2

Table 3.1: An example of a pulse sequence of length four. They all use a square envelope of length 20 ns.

In this scenario we start the generators two times, at 10 ns and at 100 ns. It is clear that we need to start at 10 ns (at the latest) in order to accommodate the first pulse. It requires the values $(90e6, 0)$ in the first generator's LUT: the pulse has a frequency of 90 MHz and since it is the first pulse of that frequency, it does not need any phase synchronisation. We do not need to change values for the second pulse, since it uses the same frequency that is already on generator 1, and has a relative phase of 0. By using the same carrier that was started at 10 ns, it naturally becomes synchronised to the first pulse's phase.

We do not need to change values for pulse 3 either, since it uses a different generator. As it is the first pulse of its frequency, its absolute phase should be equal to its relative phase 0. However, the generators were started at 10 ns, and have thus been running for 60 ns before this pulse is outputted, so the phase values that generator 2 is initialised with must be adjusted accordingly. In 60 ns at 160 MHz, the phase of the sine wave has thus progressed by 19.2π rad. To counteract this, the values that need to be saved to the generator 2 LUT become $(160e6, -19.2\pi)$. This way its phase is at 0 just as pulse 3 starts.

Pulse 4 should be synchronised to pulse 3 (due to their matching frequencies), but the user has entered a non-zero value for its relative phase. This means that we cannot keep using the same carrier that we started at 10 ns. Now, we have to shift to another LUT entry and restart the carrier to apply the values therein. The phase value that this generator must start with then becomes the phase difference between the two start points $(160e6 \cdot (100e\text{-}9 - 70e\text{-}9) \cdot 2\pi)$, plus the relative phase of π for a total of 10.6π. The final carrier change list can be seen in table 3.2, and a visualisation of how the LUT values and carriers interact with the outputted pulses can be seen in figure 3.11.

Start time	Frequency, generator 1	Phase, generator 1	Frequency, generator 2	Phase, generator 2
10 ns	90 MHz	0	160MHz	-19.2π
100 ns	Any	Any	160MHz	10.6π

Table 3.2: The start times and values needed for the sine generators in order to accommodate the pulse sequence given in table 3.1.

One might wonder what the point is of setting up the LUTs in such a complex manner. Why not just start and stop the sine generators at the beginning and end of each pulse, calculating phase values the same way every time? The issue lies in that the LUTs have a size limitation of 512 entries. If the user schedules a very long sequence of identical pulses, we need one LUT entry for each pulse in order to make sure their phases are synchronised to the first pulse. By constructing the LUT in this more complex manner, we only use

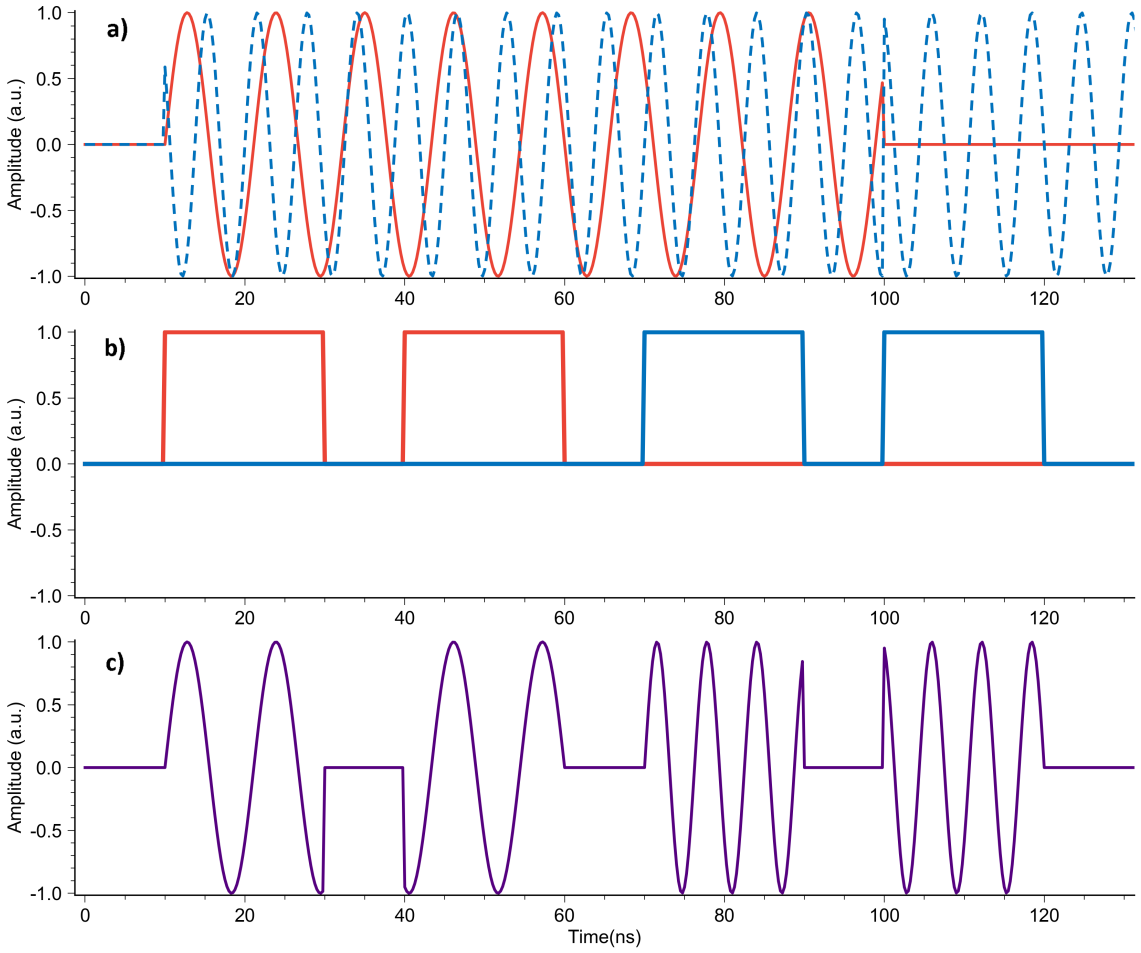

Figure 3.11: A visualisation of the pulse sequence defined by table 3.1. **a)** shows the output of the sine generators, the first one in red, the second one in blue. **b)** shows the envelopes of the four pulses, also colour-coded red and blue for generator one and two respectively. **c)** shows the resulting modulated pulses to be outputted by Vivace. Note the blue sine waves starting with phase -19.2π and 10.6π at 10 ns and 100 ns respectively to achieve the correct phases for pulses 3 and 4.

values that we need, and use them for as long as possible, minimising memory usage. To this end, we also remove duplicate entries in the LUTs before upload.

It is worth noting that although this algorithm might be less intuitive, its time complexity (see section 2.6) is still low. For every iteration of the measurement, we need only process each pulse definition once. This way, the computation time of the algorithm grows linearly with the amount of pulse definitions.

3.5.8 Sine generator scheduling

Once the carrier change list has been constructed, it can tell us what frequency and phase values are needed for the generators at all times of the measurement. In order to make use of this information, the driver steps through this list, and processes each entry in turn. First, the driver tells Vivace which LUT index contains the proper values for the carrier wave in question, and then tells it to output a new carrier starting at the indicated time and lasting until the next carrier change is scheduled. That way, outputted pulses (which are scheduled separately) always modulate with a carrier wave of the correct frequency and initial phase.

3.5.9 Pulse scheduling

While Vivace supports the addition of pulses that are active on both of a port's carrier generators at the same time, there are certain limitations to how this combination may be arranged in ViPS. The ViPS driver only allows users to declare pulses on both carriers if they start *and* end at the same time, or do not overlap at all. Initially this was done due to inconsistent behaviour in Vivace when stopping pulses while another was running, but even after this had been fixed, changing the behaviour was seen as a low-priority issue.

Once it has been verified that no pulses overlap, the driver can start setting up the full sequence of user-declared pulses. This is accomplished by iterating through the list of pulse definitions (the creation of which is detailed in section 3.5.5). For each iteration, the driver takes each definition and extracts every parameter Vivace needs from it. First, Vivace is told which index of its amplitude LUT it should navigate to, based on the pulse's amplitude value for the given iteration. Since there is a slight delay before newly-set LUT values are applied, Vivace is told to update its LUT as soon as possible after the end of the previous pulse. Following this, the pulse's starting time and duration are converted from iteration-based time to absolute time. At last, the output command is generated and sent to Vivace.

3.5.10 DRAG pulse construction

DRAG pulses [26] need special treatment when we create pulse definitions for them. A DRAG pulse is outputted on two ports simultaneously: the port the user defined them on (known as the base port) and some other port (known as the sibling port) specified through a quant. A single DRAG pulse generates four pulse definitions, each using their own sine generator on the two ports. These four pulse definitions do not use the defined envelope template as is; they instead use one of two new envelopes derived from the original template. Two special quant values are used to calculate these envelopes, *DRAG detuning frequency* and *DRAG scale*. The envelopes are constructed as follows:

1. Calculate a `beta` value, equal to `drag_scale * sampling_frequency`.

2. Construct a list of complex points, where the real parts are the points of the original envelope, and the imaginary parts are the gradient of the original points multiplied by `beta`.

3. Multiply the complex points elementwise with the set of points defined as `exp(i*2π*drag_detuning*t)` for all `t`, which are the values in time for each point of the original envelope (and `i` being the imaginary unit).

4. Let the real parts of the complex points be the first envelope, and the imaginary parts the second envelope.

But in addition to the envelopes, the phase values of the carrier waves also need adjusting. The first pulse on the base port should use a cosine wave instead of a sine wave, so it needs a $\frac{\pi}{2}$ phase shift to convert the default sine carrier wave into a cosine. The second should use a negative cosine wave with an additional $\frac{\pi}{2}$ phase shift. The first pulse on the sibling port should use a normal sine wave, and the second a negative sine wave with $\frac{\pi}{2}$ phase offset. The user can also set a *DRAG phase shift* (**DPS**) to further shift the phase values on the sibling port. An overview of all the parameters used by the four pulse definitions is given in table 3.3.

Pulse no.	Port	Sine generator	Envelope no.	Phase shift
1	Base	1	1	$\pi/2$
2	Base	2	2	$2\pi = 0$
3	Sibling	1	1	`DPS`
4	Sibling	2	2	`DPS`$+3\pi/2$

Table 3.3: Details of the four pulse definitions created by ViPS when processing a DRAG pulse definition in the instrument.

3.5.11 Logging of commands

Neither Labber nor Vivace natively supports any form of logging of interactions, which made it difficult to identify issues when developing ViPS. When running a measurement that does not cause any errors, but the resulting time trace looks wrong, it is hard to know which part of the code contains the issue. Even if the measurement does crash and gives some error message, it could have resulted from a propagating error in a different section of the code, still making it hard to identify the root cause. Thus, in order to assist in development of both currently implemented and future features, and to help troubleshoot both Vivace and the instrument, logging functionality has been developed for the ViPS driver. When toggled on, all calls to Vivace's API, along with their parameters and the time elapsed since measurement startup are written to a file, accessible even if the driver crashes partway through a measurement. This way it is possible to double-check that all calls to Vivace use expected parameters and are sequenced correctly.

3.5.12 Managing errors

ViPS is meant to be a user-friendly interface to Vivace. While offering users an easy way to access Vivace's full feature set is certainly an important part of fulfilling that goal, it is not the only one. Another crucial component of the ViPS instrument is how it lets its users know that something has gone wrong.

While the Vivace API offers a lot of freedom in setting up pulses, it (and its underlying hardware) has its limits. When the API attempts something that it cannot do, it is up to ViPS to communicate this to the user in the best manner possible. ViPS's error management mechanisms have been developed to adhere to the following principles:

- **Prevent errors from occurring in the first place**:
  Whenever possible, ViPS should not allow input values that always cause an error to occur. Such cases can usually be resolved with the help of built-in mechanisms within Labber. A commonly employed solution is to define numerical quants with upper and lower bounds, such as the amplitude scale of pulses which Vivace only allows to be within the range $[-1, 1]$. Another strategy used is to limit the user's input to a finite number of choices, such as only giving them the options to set up pulses on ports 1 through 8 (any other port number is rejected by Vivace).

- **Do not permit undefined or ambiguous behaviour**:
  Some errors cannot be "automatically" prevented as per the above principle. An example of this is the starting time values for pulses. Vivace only supports outputting pulses every other nanosecond, but one cannot prevent odd numerical values using Labber's quant definition tools. In this particular case, defining a pulse to be outputted at an odd number of nanoseconds is not considered an error by Vivace, but whether the resulting pulse would be outputted on the next or previous even nanosecond would be decided at random. Such behaviour could potentially produce unexpected results if part of an unknowing user's measurement. For quants like this, the ViPS driver parses their value once the instrument starts up, and if they contain values that lead to undefined behaviour, an error message appears that explains what the user has done wrong and how they should fix it. This also aborts the measurement that was about to be started.

- **Present as much pertinent information as possible**:
  When some forbidden action is performed, whether it be in the Vivace API or the ViPS driver, the error displayed to the user should make it easy for them to tell not only *what* went wrong, but also *where*. When dozens of pulses can be set up on each of eight different ports, the user does not particularly benefit from only being told that one pulse overlaps another somewhere, or that some pulse definition is set to use an envelope that has not been defined. For these kinds of errors, the error message presented to the user contains the port(s) in question, and typically some other relevant information (such as the offending pulse definition). Examples can be seen in figure 3.12.

- **Preserve the user's input**:
  Even if ViPS can discern exactly which quant caused an error, that quant keeps its user-given value until the user corrects it. This minimises work on the user's end in cases where e.g. the error is caused by a single digit in a longer value, or simply let

the user learn what it was that they did wrong. Furthermore, the user does not have to fear for some important value suddenly disappearing. When the quant is of the `STRING` type, ViPS can inject the word `INVALID:` at the start of the field to bring attention to it, while the original input is still preserved.

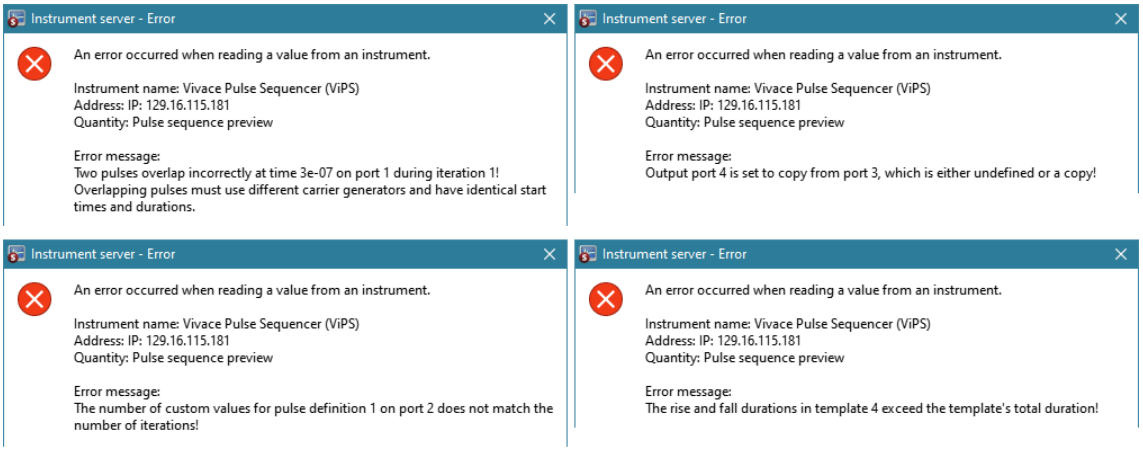

Figure 3.12: Examples of error messages generated by ViPS.

4

Results

This chapter begins with a presentation of the qubit characterisation measurement results that have been acquired using ViPS and the Digital Demodulator. These follow the same structure as the qubit characterisation walkthrough in section 2.2.3. At the end of the chapter comes a description of a script that uses these different measurements for automated qubit parameter tune-up.

4.1 Qubit characterisation measurements

Using ViPS and the Digital Demodulator, we have successfully performed all the measurements necessary for a complete qubit parameter characterisation on a live qubit in a lab environment. The results from these measurements are shown in the following subsections, along with a description of the relevant instrument settings needed to perform the measurement in question. In each measurement, both ViPS and the Digital Demodulator are used; the readout data from ViPS is fed into the Digital Demodulator for every trace collected by Labber, and it is the magnitude values of the integrated traces that are presented for all measurements in this section.

For these measurements, the hardware is set up as can be seen in figure 4.1. It should be noted that there are no special requirements as to which port numbers must be used; readout does not have to be done via the FPGA's first and second port, and the ingoing readout data does not have to be linked to the corresponding output ports.

The results of the following measurements have all been confirmed to be displaying the expected behaviour of the qubit by Marina Kudra, a researcher at QT.

4.1.1 Resonance spectroscopy

As explained in section 2.2.3.1, a resonance spectroscopy consists of a single readout pulse whose frequency is being linearly swept across a given range. In ViPS this is accomplished by first creating a square envelope of a suitable length (in this case, 100 ns); then defining a pulse on port 1 using that envelope. Port 2 is defined as a copy of port 1, with an added phase shift of $\frac{\pi}{2}$ radians so that it becomes the Q pulse counterpart to the I pulse on port 1. Finally, sampling is enabled on ports 1 and 2, set to start after the readout pulse is finished. As part of the same measurement, a local oscillator is configured to sweep over a range of frequencies within which the qubit frequency is expected to be. The Digital Demodulator's downconversion frequency is set to match the ViPS pulse's frequency through Labber's measurement setup. The measurement uses interleaved averaging over 1000 sweeps.

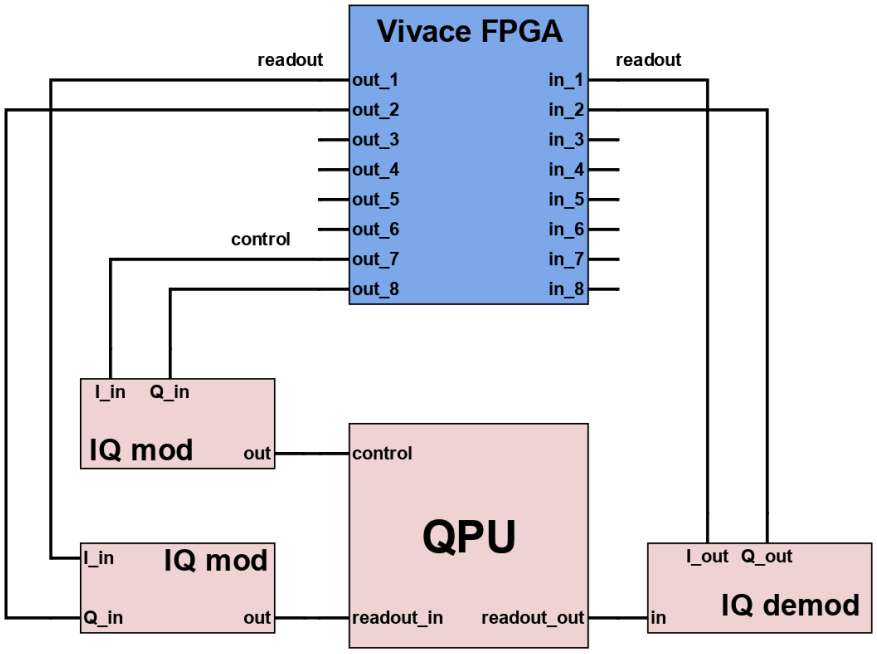

Figure 4.1: An example signal connection diagram between the QPU and Vivace. The QPU's control line input is an IQ modulated signal of port 7 and 8 of the FPGA. The readout input is similarly an IQ modulated signal of ports 1 and 2 of the FPGA. The demodulated readout output is sent to the FPGA's input ports 1 and 2. For a more detailed explanation of the *QPU*, *IQ mod* and *IQ demod* boxes, see section 2.1.

Figure 4.2 shows the results of one of our resonance spectroscopy measurements. As can be seen from the dip in magnitude in the plot, the resonator frequency f_{res} of our qubit turned out to be approximately 7315.4 MHz. For a more precise measurement of f_{res}, one could perform another resonance spectroscopy, swept over a narrower frequency range closer to our first estimate of 7315.4 MHz.

4.1.2 Two-tone spectroscopy

A two-tone spectroscopy is performed by sweeping the frequency of a control pulse sent to a qubit, and using a readout pulse at frequency f_{res} to determine its resulting state. Setting up a two-tone spectroscopy measurement can thus be done by modifying the measurement setup used for a resonance spectroscopy. The LO used for the resonator line has its frequency fixed to the the qubit's resonator frequency (in this case, 7315.4 MHz), while a second LO is introduced and set to sweep over a frequency range suspected to contain the qubit excitation frequency f_{01}. In ViPS, a control pulse is defined on port 7, set to use the same envelope as the readout pulses. Port 8 is configured to be the Q pulse to port 7's I pulse in the same way as port 2 was. The measurement uses interleaved averaging over 200 sweeps.

Figure 4.3 shows the results of one of our two-tone spectroscopy measurements. Just as explained in section 2.2.3.2, approaching f_{01} will cause the dispersive shift to occur more often, so the readout signal at frequency f_{res} will no longer be absorbed by the resonator, causing the output trace to spike. The frequency corresponding to the peak of the spike is ~5990.25 MHz, so this is our estimate of f_{01}. The smaller spikes to the sides can be

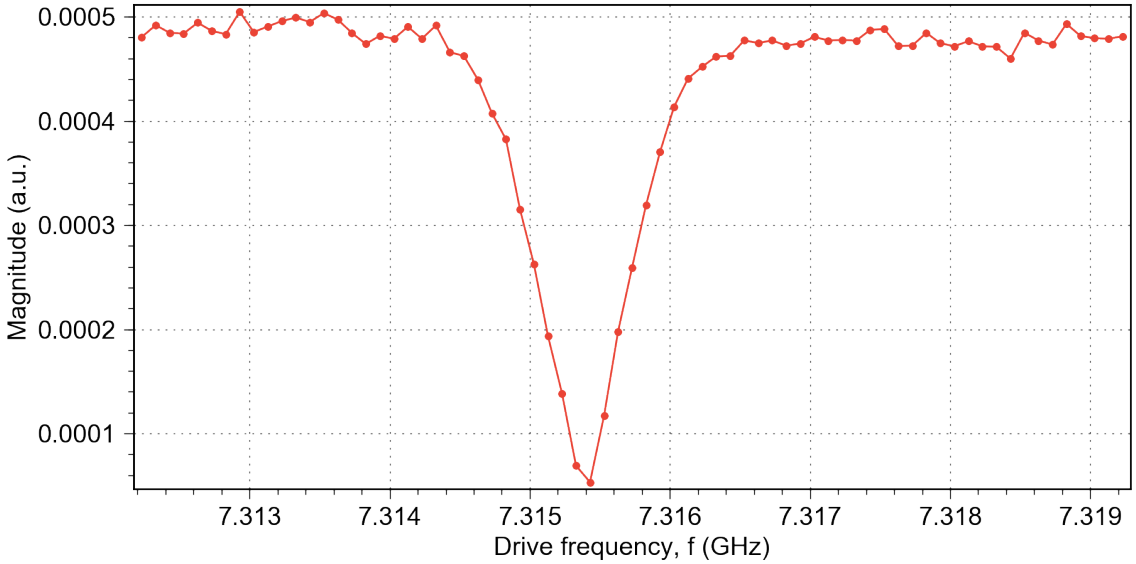

Figure 4.2: Results of our resonance spectroscopy. The plot shows the magnitude of the integrated resonator signal as a function of the frequency of the readout pulse.

seen as a result of the tilted axis we rotate around when the control pulse is not resonant. As the axis we rotate around is tilted more and more, the state vector needs to "travel" a shorter distance along the Bloch sphere's surface in order to rotate back to $|0\rangle$. As the control pulse's amplitude and duration are kept constant, the state vector will move the same distance every time. And so as the sweep's distance to f_{01} increases and decreases, this causes the oscillations seen in the plot.

4.1.3 Rabi measurement

The purpose of a Rabi measurement is to determine the parameters necessary to construct a π-pulse. This is done by sending a control pulse at frequency f_{01} to the qubit, with a fixed duration and a pulse amplitude that is swept over a linear range (or vice versa). By measuring the qubit's state immediately after this control pulse, one can obtain an approximate value for the qubit's population as a function of the swept parameter, with the ideal parameters being found when the qubit's population is 1. See section 2.2.3.3 for details. The ViPS setup of our Rabi measurement is very similar to the setup for the two-tone spectroscopy. The main point of difference is that the amplitude of the control pulse is swept from 0 to 1 in ViPS while the qubit control LO frequency is kept at f_{01}. Setting up the sweep in ViPS consists of setting the pulse's sweep parameter to *Amplitude*, and specifying the end points 0 and 1. The number of sweep steps is set to 101 with the *Iterations* quant and the measurement uses interleaved averaging over 200 sweeps.

Figure 4.4 shows the results of one of our Rabi measurements. As the control pulse's amplitude increases, the qubit's population oscillates due to the state vector rotating further along the Bloch sphere's surface. The first peak of this plot is at $i = 32$, which gives a π-pulse amplitude scale value of 0.31.

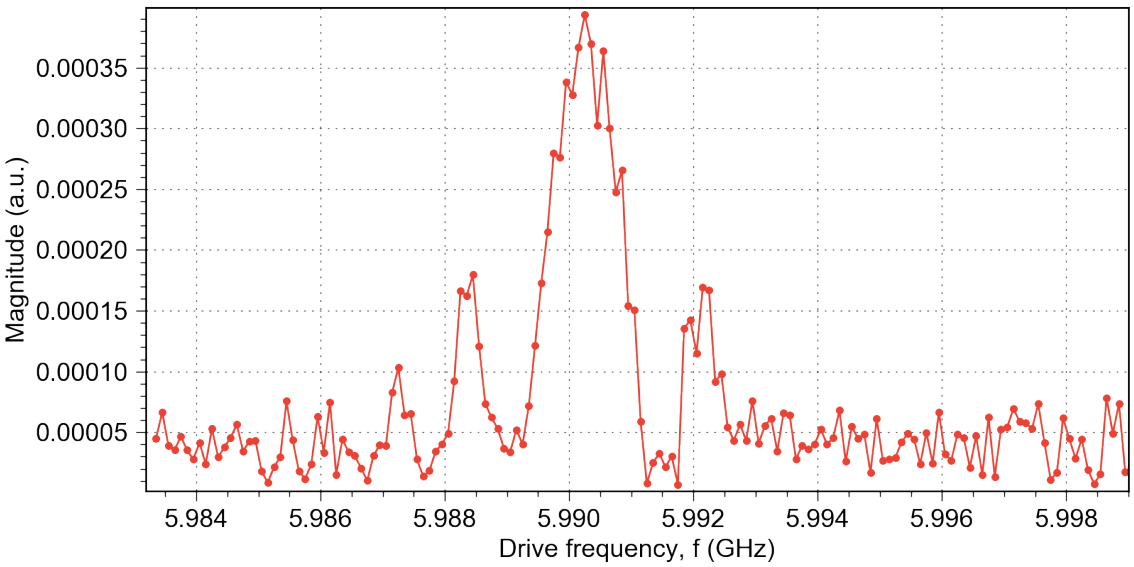

Figure 4.3: Results of our two-tone spectroscopy. The plot shows the magnitude of the integrated resonator signal as a function of the frequency of the control pulse.

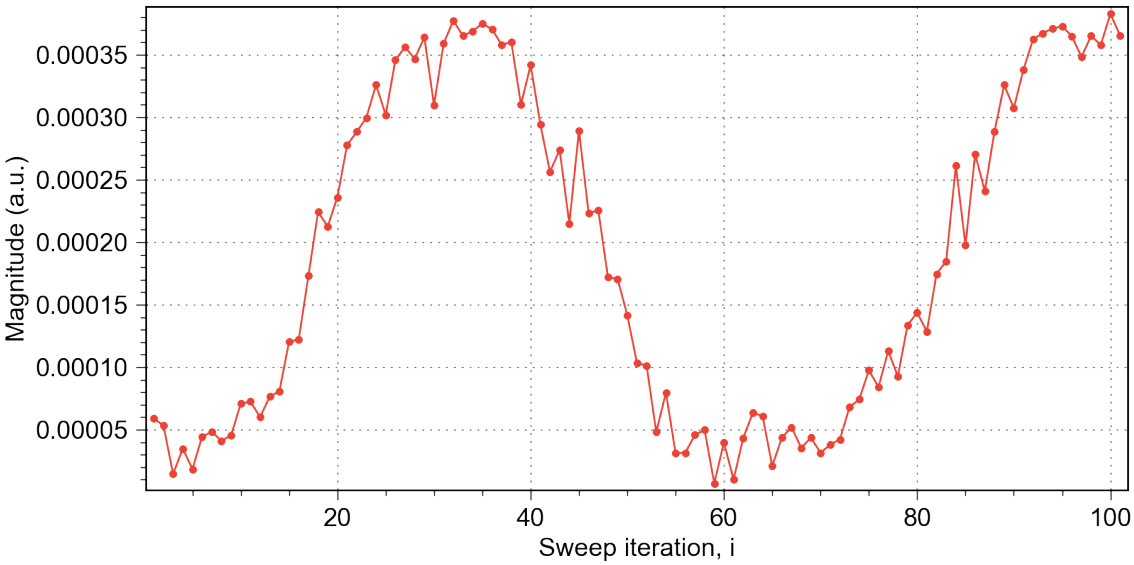

Figure 4.4: Results of our Rabi measurement. The plot shows the magnitude of the integrated resonator signal as a function of the sweep iteration. The amplitude of the control pulse is swept from 0 to 1 over 101 iterations.

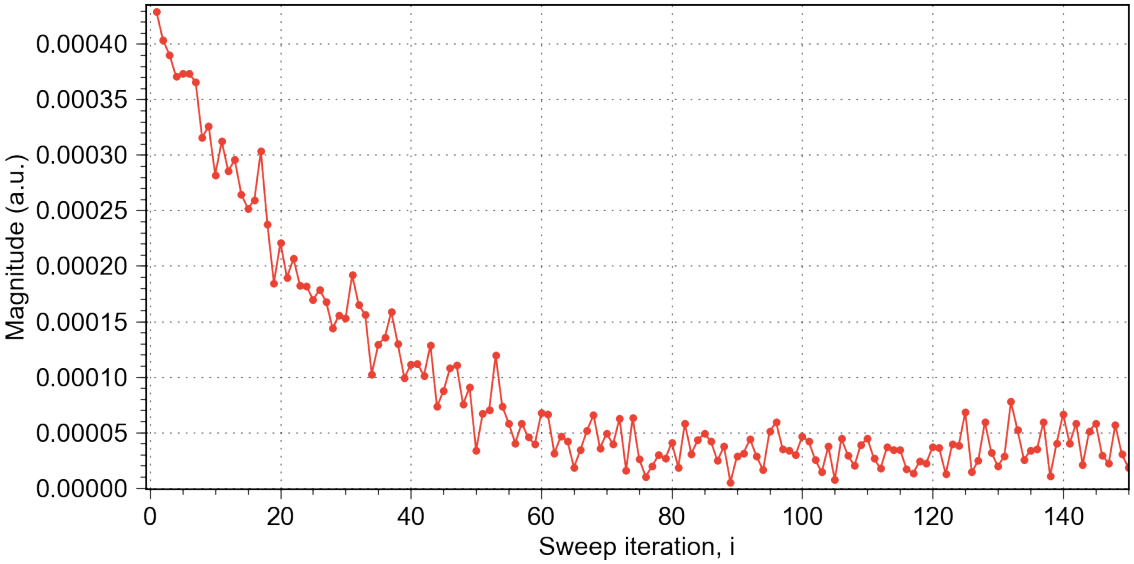

Figure 4.5: Results of our T_1 measurement. The plot shows the magnitude of the integrated resonator signal as a function of the sweep iteration. The time elapsed between control and readout pulse, τ is swept from 0 µs to 30 µs over 150 iterations.

4.1.4 T_1 measurement

This measurement is performed to learn a qubit's T_1 value, which is used as a measure of its energy relaxation time. It consists of sending a π-pulse to the qubit, then measuring its state after some linearly swept delay τ. This allows one to visualise the qubit's population as a function of the time elapsed between π-pulse and measurement. As explained in section 2.2.3.4, this decaying curve is used to find T_1. Setting up a T_1 measurement thus requires configuring pulse start times. While the control pulse – with its amplitude scale now set to 0.31 – remains constant in its start time of 0 ns, the readout pulse needs to start slightly later with each iteration. This is accomplished by defining its starting time as `100E-9+200E-9*i`, meaning that its initial start time is at 100 ns (immediately following the end of the control pulse), and that the time delay τ grows by 200 ns each iteration. The starting time of the sampling window is also set to `100E-9+200E-9*i`, in order to "follow" the readout pulse. The number of iterations is set to 150 and the measurement uses interleaved averaging over 200 sweeps.

Figure 4.5 shows the results of one of our T_1 measurements. The qubit population's decay over time is clearly visible. Using the technique laid out in section 2.2.3.4, our value for T_1 is estimated to be 3.76 µs.

4.1.5 Ramsey interferometry

Ramsey interferometry is performed to measure a qubit's dephasing rate. It is performed by sending an initial $\frac{\pi}{2}$-pulse to the qubit, and then sending another after a swept delay τ. The qubit's state is measured immediately after the second $\frac{\pi}{2}$-pulse, which makes it possible to visualise qubit population as a function of τ. Measurement setup is similar to that of the T_1 measurement. Instead of using a single π-pulse on the control channel, it is split into two $\frac{\pi}{2}$-pulses, scheduled to emit one after the other. The readout pulse is scheduled after the second control pulse, and the sampling window with the readout pulse

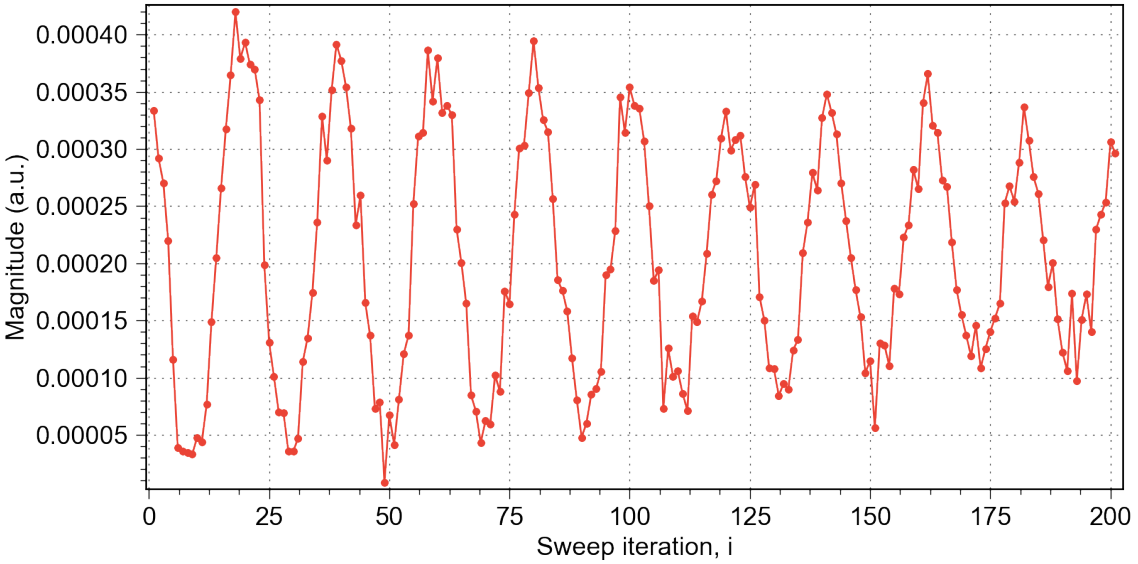

Figure 4.6: Results of our Ramsey interferometry. The graph shows the magnitude of the integrated resonator signal as a function of the sweep iteration. The time elapsed between control pulses, τ is swept from 0 µs to 10 µs over 201 iterations.

as usual. The starting time of the first $\frac{\pi}{2}$-pulse is kept constant through iterations, but the start times of the second control pulse, the readout pulse and the sampling window all have a time delta of 50 ns added to their start times. By doing this, the distance between the two $\frac{\pi}{2}$-pulses grows every iteration, while the distance between the second control pulse and readout stays the same. The number of iterations is set to 201 and the measurement uses interleaved averaging over 200 sweeps.

Figure 4.6 shows the results of one of our Ramsey interferometry measurements. As explained in section 2.2.3.5, it is possible to fit a curve of the form

$$O + A \cdot \exp\left(\frac{-x}{T_2^*}\right) \cdot \cos(2\pi \cdot f_{\text{osc}} \cdot x + \phi_0)$$

to the resulting plot. In figure 4.11, such a curve fit is shown, and the parameters of the fitted curve are as follows:

$$\begin{aligned} O &\approx 0.0002 \\ A &\approx -0.0002 \\ T_2^* &\approx 12.3 \text{ µs} \\ f_{\text{osc}} &\approx 0.981 \text{ MHz} \\ \phi_0 &\approx 3.86 \end{aligned}$$

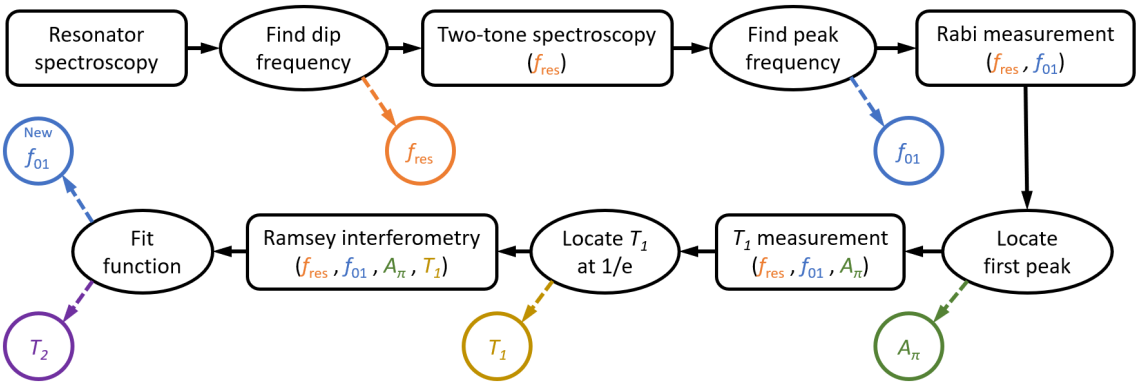

Figure 4.7: An overview of the program flow of the automated qubit tune-up script. The rounded rectangles signify Labber measurements, while the ovals signify processing of measurement data by the Python script. The coloured circles represent the storing of parameter values, and the similarly-coloured parameter lists show when those parameters are used in later Labber measurements.

4.2 Qubit tune-up automation

In order to fulfil the secondary project goal (as detailed in section 1.3.2), we have implemented a script, referred to as the *Qubit tune-up script*, that is capable of successfully performing the above measurements in sequence without human intervention. The name can be somewhat misleading, as it is not the qubit itself that is tuned up. Instead, the parameter values used to communicate with it are fine-tuned, by running the sequence of characterisation measurements described in section 2.2.3. The script is written in Python, and uses Labber's scripting API to load and modify measurement configuration files. The configurations used to achieve the results described in section 4.1 are used as bases for the different characterisations that the script performs, and an overview of the flow of the program can be seen in figure 4.7.

4.2.1 The tune-up process

The script begins by loading the configuration of a resonance spectroscopy measurement and executing it through Labber. A Lorentzian function is fitted onto the result data obtained from the measurement, as can be seen in figure 4.8. The fitting is done with the SciPy method `curve_fit`[1], which requires a definition of the function to fit, as well as some initial guesses for the parameters of that function. The method then tries to find the parameter values that minimise the sum of the square differences between the data and the function values. For the Lorentzian function, all initial guesses are stored as pre-defined values within the script. Once a good fit has been found, the frequency value found at the curve's minimum is stored as f_{res}.

Following that, the script loads a two-tone spectroscopy measurement configuration. Since f_{res} is now known, the frequency of the readout LO is overwritten with this new value. The measurement is then performed, and f_{01} is located by finding the highest point of the result data, without using fitting.

[1] An in-depth description can be found at `https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html`

Parameter	Guess	Final value	Fit error %
f_{res}	7.31543 GHz	7.31539 GHz	~0
f_{01}	–	5.99 GHz	–
A_π	0.3	0.322	0.42
T_1	5.00 µs	4.50 µs	3.27
T_2^*	$2T_1 = 9.00$ µs	12.34 µs	7.5
f_{osc}	1 MHz	0.98 MHz	0.1

Table 4.1: A table showing the initial guesses, final recorded values and fit error percentages for some of the parameters used in of the automatic qubit tune-up script.

Both f_{res} and f_{01} are then used as the readout and control frequencies of a Rabi measurement. Onto this data we want to fit a sine wave to find the exact half-period of the oscillations, which would give us the correct π-pulse amplitude A_π. However, unlike the previous fitting, the initial guesses to the fitting parameters are not all pre-defined in the script. Instead, the highest point of the first peak in the measurement data is given as the initial guess for the half-period. The fitted curve will then adjust it to give a fine-tuned central point of the peak, and that amplitude value we store as A_π. A plot showing the data and the fitted function can be seen in figure 4.9.

Next, the script performs a T_1 measurement, using A_π to perform properly calibrated π-pulses. An exponential decay curve is fitted to the result data of this measurement, and is used to locate and store T_1 in accordance with the procedure described in section 2.2.3.4. A plot showing the data and the fitted function can be seen in figure 4.10.

Lastly, the script performs two Ramsey interferometry measurements, with the amplitude of the $\frac{\pi}{2}$-pulses set to $A_\pi/2$. As mentioned in section 2.2.3.5, Ramsey interferometry measurements are typically performed with a slightly detuned control frequency in order to more easily analyse the effects of dephasing. Our two measurements are performed with control frequencies of $f_{01} \pm f_\Delta$, with f_Δ set to 1 MHz. The function given in section 2.2.3.5 is fitted to the results in order to obtain a value for T_2^*. With the frequency of the oscillations f_{low} and f_{high} from the two measurements, we also calculate a new updated version of f_{01}:

$$f_{01_{\text{new}}} = \frac{(f_{01_{\text{old}}} - f_\Delta + f_{\text{low}}) + (f_{01_{\text{old}}} + f_\Delta - f_{\text{high}})}{2} = \frac{2f_{01_{\text{old}}} + f_{\text{low}} - f_{\text{high}}}{2}$$

The `curve_fit` method also returns a matrix containing the estimated covariance of the function parameters. From this matrix we compute one standard deviation errors for some parameters of importance for the tune-up script. A table showing our initial guesses, final results and fit error percentages for a selection of qubit parameters can be seen in table 4.1.

The parameter values found by the tune-up script have been confirmed by Marina Kudra to be identical (within measurement errors) to ones achieved by performing the measurements manually.

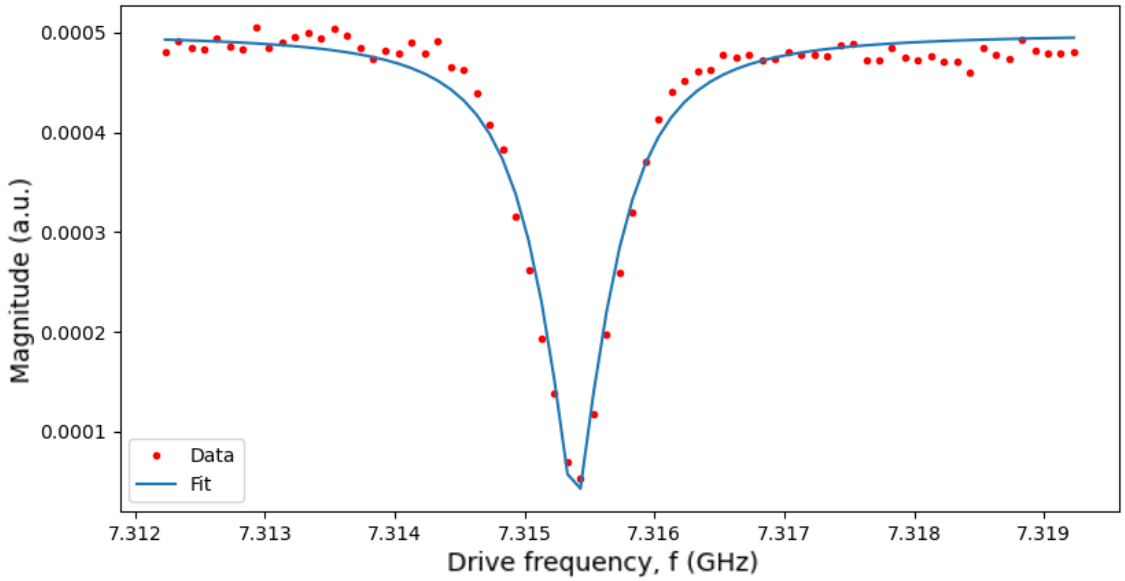

Figure 4.8: A plot showing the magnitude data from our resonance spectroscopy in red, along with a Lorentzian curve fitted onto that data in blue. Both are given as a function of the drive frequency.

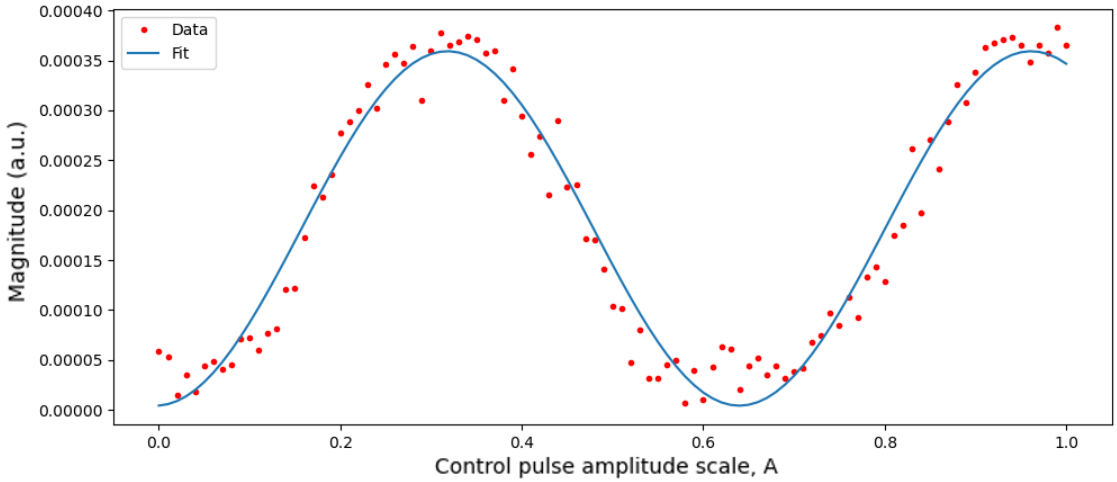

Figure 4.9: A plot showing the magnitude data from our Rabi measurement in red, along with a sine wave fitted onto that data in blue. The magnitude is here given as a function of the control pulse amplitude.

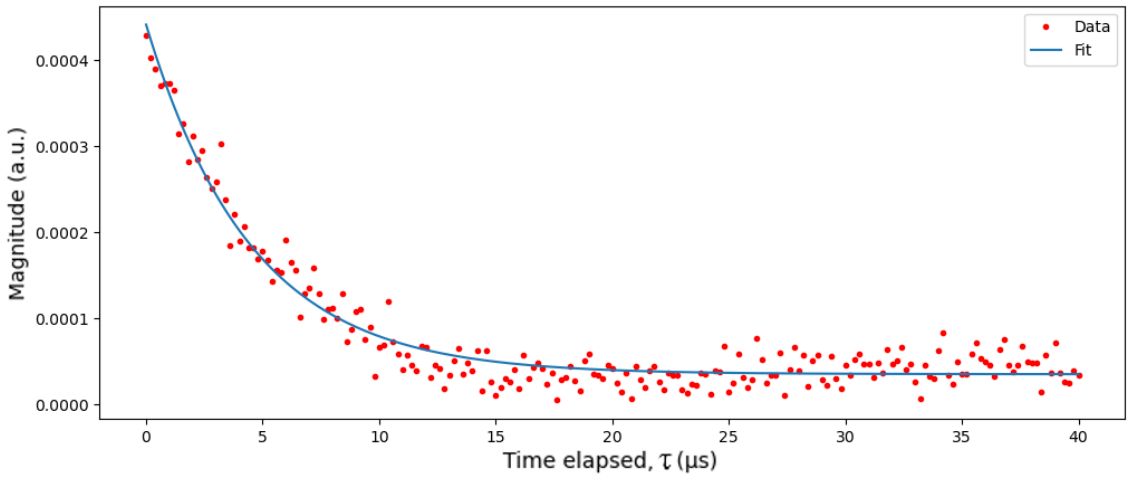

Figure 4.10: A plot showing the magnitude data from our T_1 measurement in red, along with an exponential decay curve fitted onto that data in blue. The magnitude is given as a function of τ, the duration between the control and sample pulses.

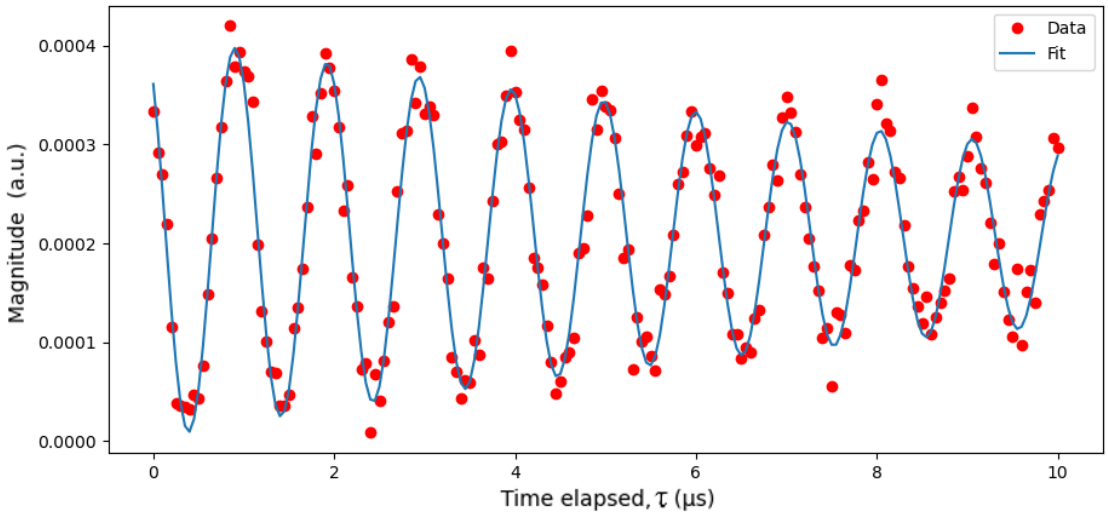

Figure 4.11: A plot showing the magnitude data from our Ramsey interferometry in red, along with a curve fitted onto that data in blue. The magnitude is given as a function of τ, the duration between π-half pulses.

5

Discussion & Conclusion

In this concluding chapter, we share our thoughts on a few topics related to this project that warrant discussion. Further, we present ideas for possible avenues of expanding upon the project. Finally, we summarise the work and evaluate how well it fulfils the overall aims and goals of the project.

5.1 Future proofing of software

Software has a short shelf life, especially when it is as closely connected to another program as our instrument drivers are. Whenever the structure of the Vivace API is updated, our drivers require maintenance depending on the severity of the changes. Within a software stack, structural changes in the lower levels inevitably propagate upwards to some degree, although the effects can be lessened by efficiently constructed abstraction layers. As an example, there was a point during this project when the Vivace API changed from having a single wave generator per port to two, requiring the LUTs to be defined in pairs. The part of the ViPS driver code that constructs LUTs to send to Vivace unavoidably had to change, but the user interface did not other than to allow the user to now select a generator. The user does not need to know what LUTs are or how they are constructed, and so by abstracting that part away from the GUI level, the changes needed were mostly limited to our driver. In the same way, no hypothetical abstraction layers built on top of our drivers should need to be modified to accommodate smaller changes in implementation at the hardware level.

5.2 The costs and benefits of an instrument control platform

As Vivace provides an API for accessing its features via regular Python code, one may wonder what point there is in making it controllable via an instrument driver for a platform such as Labber, instead of building a distinct Vivace control program. It is our opinion that there are two main sets of benefits brought about by implementing ViPS as a virtual instrument for Labber.

The first set of benefits are those related to ease of development. As described in section 3.8, Labber constructs instrument GUIs automatically based on the instrument's definition file. As such, the only issue of GUI construction was fitting our instrument design into the rigid Labber GUI mould of quants in groups and sections. We have thus not needed to pay any mind to other aspects of GUI development such as connecting fields properly to the program's internal logic, polling for user input and designing an

efficient system for handling values. Of course, writing a GUI from scratch using some third-party library would have given us greater control over how to structure GUI components, allowing us to improve usability beyond what is permitted by Labber. However, such improvements would only be useful for direct instrument control, and not when, for instance, the instruments are controlled by an automation script.

The second set of benefits of implementing ViPS as a Labber instrument is the connectivity and coordination that comes with any instrument control platform, as outlined in section 2.5. If ViPS did not support these features, using Vivace together with other instruments would require setting up a Python script that should somehow manage to run a pre-defined Labber measurement while simultaneously running the ViPS program such that the two remain synchronised. Running all instruments through the same measurement program also makes it easier to consolidate measurement results and settings, as well as passing data from one instrument to another (exemplified in how the Digital Demodulator can work with any Labber instrument that outputs IQ-modulated data).

5.3 Future work

Although ViPS represents a shift in working methodology compared to older waveform generation solutions that have seen use at QT, it would be prudent to perform a more thorough comparison of usability and usefulness between these ways of working. The results of such an evaluation would give a more concrete indicator as to the benefits and disadvantages of this new architecture. For instance, a quantitative evaluation could be made by setting up specific pulse sequences on ViPS and some other instrument, then comparing the time taken to set up and generate certain pulse sequences, as well as ensuring that the results they produce are similar enough. In the same vein, a qualitative evaluation of e.g. usability could be made through interviews with users.

At QT, there are currently plans for an advanced abstraction stack that would allow users to control the laboratory's qubits through a remote and hardware-agnostic frontend (an outline of the currently planned architecture for this system can be seen in the grey portions of figure 1.2). Given the plans for a somewhat universal feature set to be employed by the coordinator layer, it is important that ViPS has all the functionality required by that feature set. As such, establishing a link between this layer and ViPS's driver is necessary for the construction of the full software stack.

As is touched upon in section 5.1, ViPS should ideally be kept updated to stay apace with any changes made to Vivace and its API. Furthermore, as Vivace is being actively developed, new features are occasionally added. These features would need to be implemented in ViPS in order to be easily accessible.

As more kinds of measurements are successfully performed, automation scripts that can perform these with limited to no human input should be created. For instance, calibration of signal mixers could potentially be handled in a similar manner to the qubit tune-up script developed as part of this project.

5.4 Conclusion

During this project, we have developed the instrument *Vivace Pulse Sequencer* (ViPS) for the Labber instrument control platform. ViPS acts as a frontend for Vivace, an FPGA-based pulse sequencer capable of generating electrical pulses described in a format that separates envelopes from carrier waves. This contrasts how pulse generators typically model such sequences as a single linear sequence of values.

Part of our research question concerns how intricate pulse sequences should be generated. Through ViPS, users can define pulse sequences in a very similar manner to Vivace's internal representation. This allows for pulse sequences to be built using reusable components, such as envelopes of specific shapes being used for multiple carriers. The way pulses are defined within a single trigger period has been shown to be a flexible way to construct sequence structures that change over time, such as sweeps over multiple parameter values, start times or durations.

Performing qubit measurements requires control of more than just waveform generator hardware; for example local oscillators need to be synchronised with pulse emission during the measurement. ViPS has successfully been used for a variety of qubit characterisation measurements on a superconducting qubit at Chalmers' Quantum Technology laboratory, showing that ViPS is capable of performing measurements in conjunction with other instruments. These measurements have required the use of advanced pulse sequences structures that involve sweeping pulse parameters, changing start times over a series of sequence iterations, and interleaved averaging.

To obtain legible information from a QPU, it must be possible to receive demodulated versions of the signals sent to it. The Digital Demodulator instrument was developed for this purpose, and can accurately extract such data from arbitrary signal sources and sample rates. The demodulated data can also be converted into arbitrary sampling rates, further extending the flexibility of the instrument.

In addition to using ViPS to perform qubit characterisation measurements individually, we have made use of Labber's measurement automation framework to create a qubit tune-up script. The procedure described in this script involves running a sequence of measurements and adjusting parameters of ViPS based on earlier measurements' results. This not only demonstrates the benefits of controlling Vivace through an instrument control platform, but serves as an example of how further abstraction levels for qubit control could be constructed on top of ViPS and Labber in general.

Based on the above results, we conclude that the implementation of the Vivace Pulse Sequencer and Digital Demodulator instruments are together a successful example of a system that allows for flexible generation of intricate arbitrary pulse sequences for control and readout of a superconducting QPU setup, that is also easily automated and run alongside other laboratory equipment. Thus, we deem ViPS and the Digital Demodulator answers to our research question.

Bibliography

[1] P. Benioff, "The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines", *Journal of Statistical Physics*, vol. 22, pp. 563–591, May 1980, ISSN: 1572-9613. [Online]. Available: https://doi.org/10.1007/BF01011339.

[2] P. W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer", *SIAM Journal on Computing*, vol. 26, pp. 1484–1509, 1997. [Online]. Available: https://doi.org/10.1137/S0097539795293172.

[3] R. P. Feynman, "Simulating physics with computers", *International Journal of Theoretical Physics*, vol. 21, pp. 467–488, Jun. 1982, ISSN: 1572-9575. [Online]. Available: https://doi.org/10.1007/BF02650179.

[4] F. Arute, K. Arya, R. Babbush, *et al.*, "Quantum supremacy using a programmable superconducting processor", *Nature*, vol. 574, pp. 505–510, 2019, ISSN: 1476-4687. [Online]. Available: https://doi.org/10.1038/s41586-019-1666-5.

[5] C. Fisher and E. Abenojar. (Apr. 2009). IBM quantum experience: Overview, [Online]. Available: https://www.ibm.com/quantum-computing/technology/experience/ (visited on May 11, 2020).

[6] C. H. Bennett and G. Brassard, "Quantum cryptography: Public key distribution and coin tossing", *Theoretical Computer Science*, vol. 560, pp. 7–11, Dec. 2014, ISSN: 0304-3975. [Online]. Available: http://doi.org/10.1016/j.tcs.2014.05.025.

[7] P. W. Shor, "Progress in Quantum Algorithms", in *Experimental Aspects of Quantum Computing*, H. O. Everitt, Ed. Boston, MA: Springer US, 2005, pp. 5–13, ISBN: 978-0-387-27732-5. [Online]. Available: https://doi.org/10.1007/0-387-27732-3_2.

[8] J. Preskill, "Lecture Notes for Physics 229: Quantum Computation", 1997. [Online]. Available: www.theory.caltech.edu/people/preskill/ph229/.

[9] M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information*. Cambridge University Press, 2010.

[10] T. D. Ladd, F. Jelezko, R. Laflamme, *et al.*, "Quantum computers", *Nature*, vol. 464, pp. 45–53, Mar. 2010, ISSN: 1476-4687. [Online]. Available: https://doi.org/10.1038/nature08812.

[11] R. J. Schoelkopf and S. M. Girvin, "Wiring up quantum systems", *Nature*, vol. 451, pp. 664–669, Feb. 2008, ISSN: 1476-4687. [Online]. Available: https://doi.org/10.1038/451664a.

[12] D. P. DiVincenzo, "The Physical Implementation of Quantum Computation", *Fortschritte der Physik*, vol. 48, pp. 771–783, 2000. [Online]. Available: https://doi.org/10.1002/1521-3978(200009)48:9/11%3C771::AID-PROP771%3E3.0.CO;2-E.

[13] M. H. Devoret, A. Wallraff, and J. M. Martinis, "Superconducting Qubits: A Short Review", 2004. [Online]. Available: https://arxiv.org/abs/cond-mat/0411174.

[14] C. Križan, "Instrument and measurement automation for classical control of a multi-qubit quantum processor", Master's thesis, Chalmers University of Technology, Department of Computer Science and Engineering, Computer Engineering Division, Gothenburg, Sweden, 2019. [Online]. Available: https://hdl.handle.net/20.500.12380/300065.

[15] G. Andersson, "Circuit quantum electrodynamics with a transmon qubit in a 3D cavity", Master's thesis, KTH, Applied Physics, 2015. [Online]. Available: http://kth.diva-portal.org/smash/record.jsf?pid=diva2%3A813846.

[16] P. Krantz, M. Kjaergaard, F. Yan, *et al.*, "A quantum engineer's guide to superconducting qubits", *Applied Physics Reviews*, vol. 6, p. 021 318, 2019. [Online]. Available: https://doi.org/10.1063/1.5089550.

[17] J. Clarke and F. K. Wilhelm, "Superconducting quantum bits", *Nature*, vol. 453, pp. 1031–1042, Jun. 2008, ISSN: 1476-4687. [Online]. Available: https://doi.org/10.1038/nature07128.

[18] D. Dong, C. Chen, B. Qi, *et al.*, "Robust manipulation of superconducting qubits in the presence of fluctuations", *Scientific Reports*, vol. 5, p. 7873, Jan. 2015, ISSN: 2045-2322. [Online]. Available: https://doi.org/10.1038/srep07873.

[19] M. Kjaergaard, M. E. Schwartz, J. Braumüller, *et al.*, "Superconducting Qubits: Current State of Play", *Annual Review of Condensed Matter Physics*, vol. 11, pp. 369–395, 2020. [Online]. Available: https://doi.org/10.1146/annurev-conmatphys-031119-050605.

[20] G. Wendin, "Quantum information processing with superconducting circuits: a review", *Reports on Progress in Physics*, vol. 80, p. 106 001, Sep. 2017. [Online]. Available: https://doi.org/10.1088%2F1361-6633%2Faa7e1a.

[21] (2020). Intermodulation Products – Vivace, [Online]. Available: https://intermodulation-products.com/products/vivace (visited on May 20, 2020).

[22] F. T. Chong, D. Franklin, and M. Martonosi, "Programming languages and compiler design for realistic quantum hardware", *Nature*, vol. 549, pp. 180–187, 2017, ISSN: 1476-4687. [Online]. Available: https://doi.org/10.1038/nature23459.

[23] (2020). Labber – Software for Instrument Control and Lab Automation, [Online]. Available: https://labber.org/ (visited on May 20, 2020).

[24] (2020). Labber Python API Documentation, [Online]. Available: https://labber.org/online-doc/api/index.html (visited on May 22, 2020).

[25] (2020). Instrument drivers for Labber, [Online]. Available: https://github.com/Labber-software/Drivers (visited on May 21, 2020).

[26] F. Motzoi, J. M. Gambetta, P. Rebentrost, *et al.*, "Simple Pulses for Elimination of Leakage in Weakly Nonlinear Qubits", *Phys. Rev. Lett.*, vol. 103, p. 110 501, 11 Sep. 2009. [Online]. Available: https://doi.org/10.1103/PhysRevLett.103.110501.

[27] P. A. M. Dirac, "A new notation for quantum mechanics", *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 35, pp. 416–418, 1939. [Online]. Available: https://doi.org/10.1017/S0305004100021162.

[28] M. Dobšíček, "Quantum computing, phase estimation and applications", Jan. 2008. [Online]. Available: https://arxiv.org/abs/0803.0909.

[29] S. Weinberg, *Lectures on quantum mechanics*. Cambridge University Press, 2015.

[30] A. C. Lierta, T. Demarie, and E. Munro. (May 22, 2018). Quantum Computation: a journey on the Bloch sphere., Quantum World Association, [Online]. Available: https://medium.com/@quantum_wa/quantum-computation-a-journey-on-the-bloch-sphere-50cc9d73530 (visited on Jun. 4, 2020).

[31] I. L. Chuang and Y. Yamamoto, "Simple quantum computer", *Phys. Rev. A*, vol. 52, pp. 3489–3496, 5 Nov. 1995. [Online]. Available: https://doi.org/10.1103/PhysRevA.52.3489.

[32] R. Keim. (Aug. 13, 2018). What Is an FPGA? An Introduction to Programmable Logic, All About Circuits, [Online]. Available: https://www.allaboutcircuits.com/technical-articles/what-is-an-fpga-introduction-to-programmable-logic-fpga-vs-microcontroller/ (visited on Jun. 3, 2020).

[33] C. A. Ryan, B. R. Johnson, D. Ristè, *et al.*, "Hardware for dynamic quantum computing", *Review of Scientific Instruments*, vol. 88, p. 104 703, 2017. DOI: 10.1063/1.5006525. [Online]. Available: https://doi.org/10.1063/1.5006525.

[34] L. Steffen, Y. Salathe, M. Oppliger, *et al.*, "Deterministic quantum teleportation with feed-forward in a solid state system", *Nature*, vol. 500, pp. 319–322, Aug. 2013, ISSN: 1476-4687. DOI: 10.1038/nature12422. [Online]. Available: https://doi.org/10.1038/nature12422.

[35] Y. Salathé, P. Kurpiers, T. Karg, *et al.*, "Low-Latency Digital Signal Processing for Feedback and Feedforward in Quantum Computing and Communication", *Phys. Rev. Applied*, vol. 9, p. 034 011, 3 Mar. 2018. [Online]. Available: https://doi.org/10.1103/PhysRevApplied.9.034011.

[36] D. Ristè, M. Dukalski, C. A. Watson, *et al.*, "Deterministic entanglement of superconducting qubits by parity measurement and feedback", *Nature*, vol. 502, pp. 350–354, Oct. 2013, ISSN: 1476-4687. [Online]. Available: https://doi.org/10.1038/nature12513.

[37] P. Campagne-Ibarcq, E. Flurin, N. Roch, *et al.*, "Persistent Control of a Superconducting Qubit by Stroboscopic Measurement Feedback", *Phys. Rev. X*, vol. 3, p. 021 008, 2 May 2013. [Online]. Available: https://doi.org/10.1103/PhysRevX.3.021008.

[38] Y. Salathé, P. Kurpiers, T. Karg, *et al.*, "Low-Latency Digital Signal Processing for Feedback and Feedforward in Quantum Computing and Communication", *Phys. Rev. Applied*, vol. 9, p. 034 011, 3 Mar. 2018. [Online]. Available: https://doi.org/10.1103/PhysRevApplied.9.034011.

[39] P. Magnard, P. Kurpiers, B. Royer, *et al.*, "Fast and Unconditional All-Microwave Reset of a Superconducting Qubit", *Phys. Rev. Lett.*, vol. 121, p. 060 502, 6 Aug. 2018. [Online]. Available: https://doi.org/10.1103/PhysRevLett.121.060502.

[40] R. Gebauer, N. Karcher, D. Gusenkova, *et al.*, "State preparation of a fluxonium qubit with feedback from a custom FPGA-based platform", 2019. [Online]. Available: https://arxiv.org/abs/1912.06814.

[41] B. Kannan, D. Campbell, F. Vasconcelos, *et al.*, "Generating Spatially Entangled Itinerant Photons with Waveguide Quantum Electrodynamics", 2020. [Online]. Available: https://arxiv.org/abs/2003.07300.

[42] (2020). Intermodulation products – About, [Online]. Available: https://intermodulation-products.com/about (visited on Jun. 2, 2020).

[43] (2020). Xilinx – Zynq UltraScale+ RFSoC ZCU111 Evaluation Kit, [Online]. Available: https://www.xilinx.com/products/boards-and-kits/zcu111.html (visited on Jun. 3, 2020).

[44] P. Virtanen, R. Gommers, T. E. Oliphant, *et al.*, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python", *Nature Methods*, vol. 17, pp. 261–272, 2020. [Online]. Available: https://doi.org/10.1038/s41592-019-0686-2.

[45] (2020). Labber Documentation, [Online]. Available: https://labber.org/online-doc/html/index.html (visited on May 27, 2020).