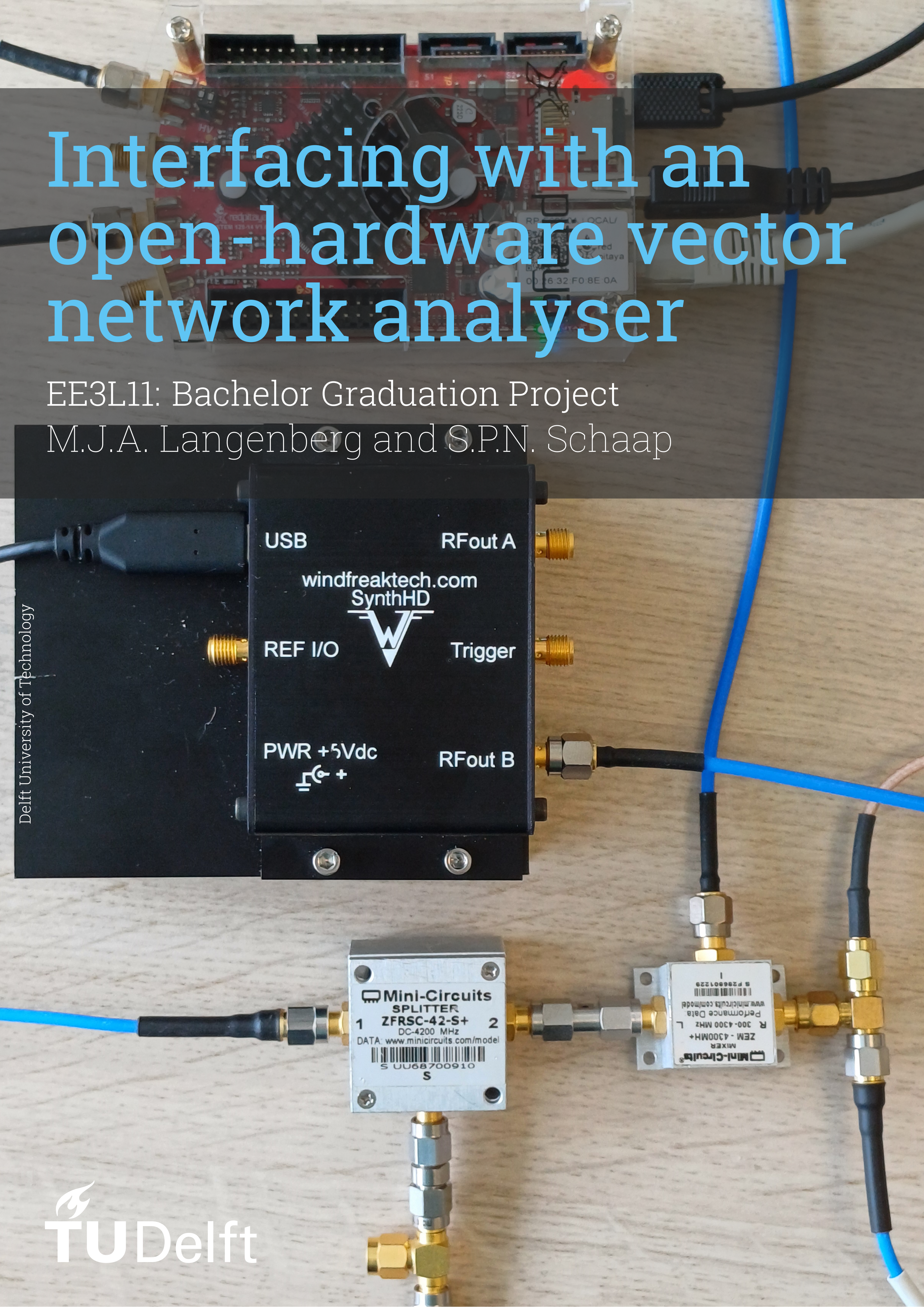

Interfacing with an open-hardware vector network analyser

EE3L11: Bachelor Graduation Project

M.J.A. Langenberg and S.P.N. Schaap

Delft University of Technology

TU Delft

Interfacing with an open-hardware vector network analyser

by

M.J.A. Langenberg and S.P.N. Schaap

in partial fulfillment of the requirements for the degree of

Bachelor of Science

in Electrical Engineering

defended on Friday June 21, 2024 at 09:30 a.m.

Students:	M.J.A. Langenberg	5557313
	S.P.N. Schaap	5597781
Project supervisor:	Prof.dr. G. Steele	TU Delft
EEMCS supervisor:	Dr.ir. N. Haider	TU Delft
Thesis committee:	Prof.dr. G. Steele	TU Delft
	Dr.ir. N. Haider	TU Delft
	Dr.ir. J.S.S.M. Wong	TU Delft

Cover:	Red Pitaya STEMlab 125-14, SynthHD V2 by Windfreak Technologies LLC, Mini-Circuits ZFRSC-42-S+ splitter and Mini-Circuits ZEM-4300MH+ Mixer connected by coaxial cables
Style:	TU Delft Report Style, with modifications by D. Zwaneveld and M.J.A. Langenberg

TU Delft

Preface

The last two months have been dedicated to working on the bachelor thesis you have in front of you. This project, conceived by Gary Steele, has involved tremendous effort to present a working prototype and to prove the concept.

We would like to extend our heartfelt thanks to Gary Steele, Nadia Haider, and Stephan Wong for their invaluable support and guidance, which were instrumental in making this project a success in our eyes. We are proud of how far we have come and how much we have learned over this period.

Additionally, we want to thank Ruben Dirkzwager, Anne Hinrichs, Maarten Oudijk and Samet Öztürk for their excellent teamwork and collaboration throughout this project.

M.J.A. Langenberg and S.P.N. Schaap
Delft, June 2024

Abstract

For microwave qubit readout in research applications, a Vector Network Analyser has been designed. The objective of this project was to design and build a modular, extensible VNA, containing open hardware and implemented in open-source software.

This thesis discusses the Python implementation of an interface between the digital signal processing step, taking place inside an FPGA, and the output of data to the user, being in graphical form and as systematical data structure to be stored on a PC. The interface is split up into a server, responsible for communicating with the FPGA on the same chip, and a client, which receives the measurement data from the server via the Transmission Control Protocol and controls the radio frequency signal generators that serve as stimulus for the device under test and as local oscillator for downconversion.

An overview of VNAs and their application in this project is given in the first chapter. The programme of requirements and implementation overview are discussed next, followed by detailed explanations of the Python implementation of the server and client software. The achieved results satisfy the requirements for throughput, extensibility and data transfer overhead time. The thesis concludes with recommendations for future developments and extensions to this project.

Contents

[2] This chapter is shared between the three theses written by the three subteams of the project.

Nomenclature

Abbreviations

abbreviation	definition / description
ADC	Analog-to-Digital Converter
API	Application Programming Interface
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory, stores data used by SoC
DSP	Digital Signal Processing
DuT	Device under Test
EMW	Electromagnetic Wave
FPGA	Field Programmable Gate Array
GPIO	General Purpose Input/Output
GUI	Graphical User Interface
hwh	hardware handoff, file extension
IF	Intermediate Frequency (EMW)
ipynb	interactive Python (Jupyter) notebook, file extension
IQ	In-phase and Quadrature-phase components of a sinusoid
IVI	Interchangeable Virtual Instruments (foundation)
LAN	Local Area Network
LO	Local Oscillator
MMIO	Memory-Mapped Input/Output
NI-VISA	National Instruments VISA
PL	Programmable Logic, using FPGA technology
PS	Processing System
-Py-	Python prefix or suffix
REF	Reference signal, not going through the DuT
RF	Radio Frequency (EMW)
SCPI	Standard Commands for Programmable Instruments
SDR	Software-Defined Radio
SMA	SubMiniature version A, type of connector for coaxial cables
SoC	System on Chip, combination of PL and PS
SQUID	Superconducting Quantum Interference Device
TCP	Transmission Control Protocol
ui	user interface (also used as file extension)
USB	Universal Serial Bus
VISA	Virtual Instrument Software Architecture
VNA	Vector Network Analyser
VSA	Vector Signal Analyser

Units

unit symbol	meaning
b	bit
B	byte
°C	degrees Celsius
dB	decibel
dBc	decibel, used for power ratio to carrier signal
dBm	decibel, compared to 1 milliwatt
Hz	hertz (s^{-1})
s	second
V	volt

1

Introduction[1]

1.1. VNA, a general overview

A Vector Network Analyser (VNA) is a device that sends an electromagnetic wave (EMW) at a known frequency and amplitude through a Device under Test (DuT) or network, and records the reflected and transmitted waves [1]. The recorded waves are compared to the transmitted wave to derive a vector output, giving the change in amplitude and phase caused by the DuT.

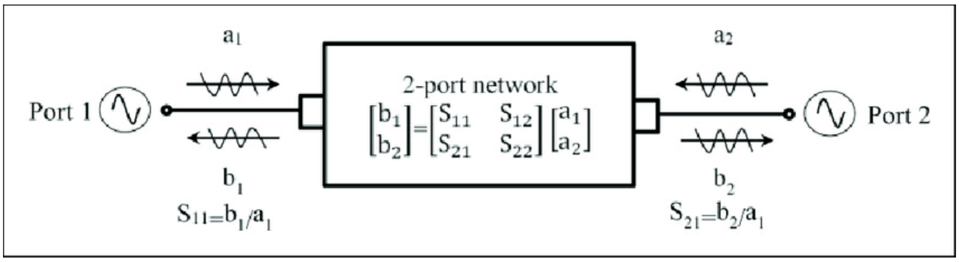

Figure 1.1: S-parameters [2]

The reflected EMW, transmitted EMW and the EMW that is sent by the VNA, which from now on can be referred to as the reference signal, can be represented using two sinusoidal waves: an in-phase cosine (I) and a sine, shifted by 90 degrees compared to I, referred to as the quadrature wave (Q). These waves are combined to form a complex mathematical IQ representation: $I + \mathrm{j}Q$.

The change caused by a DuT in its reflection and transmission of the reference signal are quantified by scattering parameters, or S-parameters, which are a form of network parameters. For a two-port DuT, the S-parameters can be put inside a 2×2-matrix [3], shown in figure 1.1. These parameters contain information about both the phase and amplitude change caused by the DuT, in a complex form. They are obtained by complex division of the reflected or transmitted signal by the reference signal, such as in equation (1.1).

$$S_{21} = \frac{b_2}{a_1} = \frac{I_{\mathrm{trans}} + \mathrm{j}Q_{\mathrm{trans}}}{I_{\mathrm{ref}} + \mathrm{j}Q_{\mathrm{ref}}} \tag{1.1}$$

For this project, this S_{21} transmission parameter is of interest, which relates the transmitted signal (b_2 in figure 1.1) to the reference signal (a_1).

VNAs have two main procedures to test a DuT. The first procedure is called frequency sweep, where an EMW is sent with a constant power and a frequency changing over a short time span in predefined

[1]This chapter is shared between the three theses written by the three subteams of the project.

steps. This procedure is used to determine the frequency dependence of the reflection and transmission parameters of the DuT. The second procedure is a power sweep, where an EMW is sent with constant frequency and a power changing over a short time span. This procedure is used to determine the power transfer of the DuT at different input powers. For this project, only the frequency sweep is of interest, and implementation of power sweeping is left to future projects.

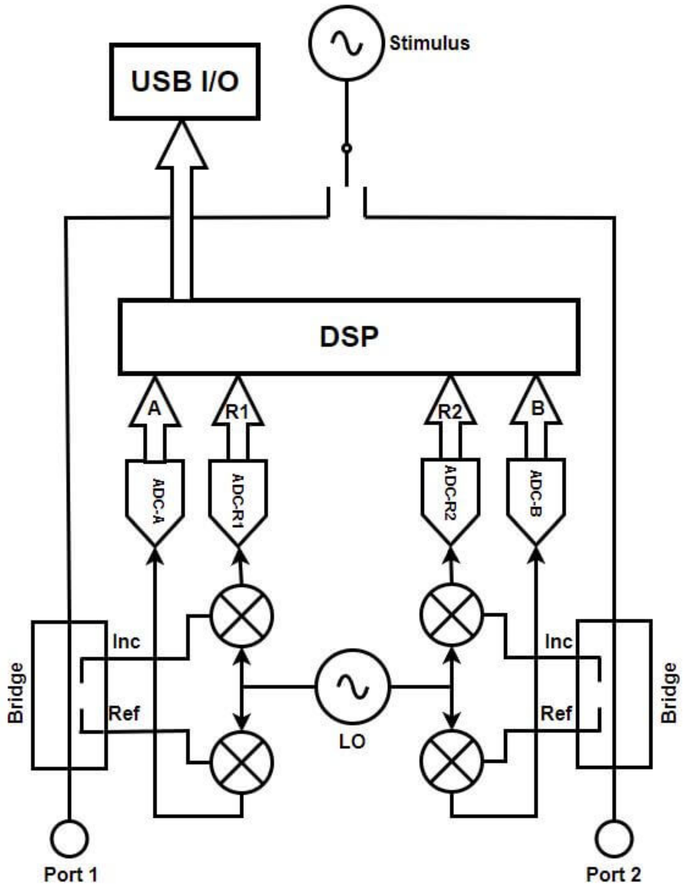

Figure 1.2: Block diagram of a simple VNA, [4]

The internal working of a general simple VNA is shown in figure 1.2. An RF stimulus coming from port 1 is provided to a DuT, which is connected between port 1 and port 2 (not shown in the figure). The stimulus is passed through a bridge (directional coupler), which splits the EMW in forward- and backward-going waves, which takes this signal as reference (Ref). This reference signal is demodulated into a lower frequency IF signal (intermediate frequency) using a mixer and a local oscillator (LO). The intermediate frequency is determined by the difference in frequency of the LO and the incoming signal. The reflected EMW coming from the DuT will be split off as the “Inc”-signal by the bridge at port 1, and the transmitted wave as the “Inc”-signal at port 2. They then go through the same process as the reference signal, to obtain two more IF signals. The same process can also be done with a RF stimulus coming from port 2, producing another reference, transmission and reflection IF signal, to study the effects of the DuT in two directions by finding other S-parameters. It must be noted that some of the functionality of the general VNA of figure 1.2 are omitted in the VNA of this project, as will become clear from the functional requirements in section 1.4.

All IF signals are then digitised in analog-to-digital converters (ADCs) and processed in the Digital Signal Processing unit (DSP). In the DSP unit, the four S-parameters are calculated by doing complex divisions such as the one in equation (1.1). After that, the data can be retrieved via a data bus such as USB, or be immediately shown on a screen.

1.2. Application in quantum research

A Transmon qubit is a type of superconducting charge qubit. It consists of a superconducting quantum interference device (SQUID), a non-linear inductive element made of two superconductors separated by a thin insulating barrier, and a shunting capacitor C_{t}. The SQUID consists of two Josephson junctions in a loop. The Josephson junctions provide the non-linear inductance necessary to create quantised energy levels with nonuniform spacing (also known as anharmonicity). Anharmonicity is the key to confining the dynamics of multi-level quantum system (such as a Transmon) to within a two-level subspace when it is driven.

Being able to confine the dynamics within a two-level subspace is important, because it simplifies the system to a manageable quantum bit, or qubit, which is the fundamental unit of information in quantum computing. This confinement allows for clear distinction between the two states, $|0\rangle$ and $|1\rangle$, necessary for reliable quantum operations and algorithms. It also reduces the likelihood of leakage into higher energy states, which can lead to errors and decoherence, thus improving the overall stability and performance of quantum circuits. The primary role of the shunting capacitor is to increase the charging energy relative to the Josephson energy, which mitigates the effects of charge noise and enhances the robustness of the qubit.

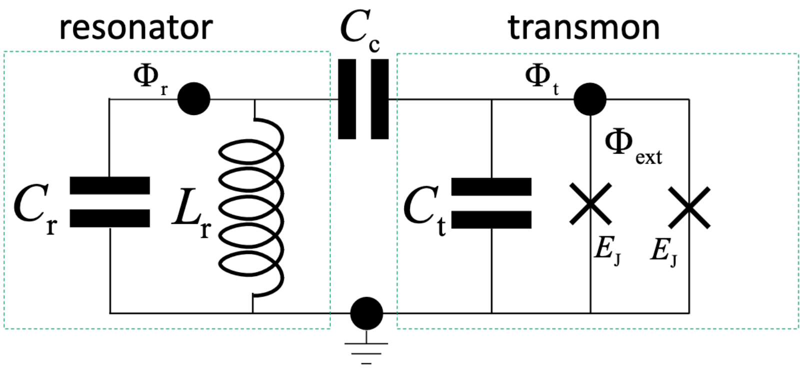

Figure 1.3: Transmon qubit coupled to a resonator [5]

Figure 1.3 shows the lumped element model of the Transmon qubit coupled to a resonator. The resonator is implemented as a waveguide (here modelled as a single inductance L_{r} and capacitance C_{r}). The resonator is the mechanism by which the qubit is read out, so it is also called the readout resonator.

The key to the microwave readout is sending a calibrated microwave pulse towards the resonator. This pulse is typically set at or near the resonator's base frequency ω_{r}, but the qubit-state-dependent frequency shift (either to $\omega_{\mathrm{r}}-\chi$ or to $\omega_{\mathrm{r}}+\chi$) affects how this pulse interacts with the resonator. Reading out a qubit in practice is done by the use of a Vector Network Analyser. Qubit measurement can be performed by taking the superconducting qubit circuit as the device under test (DuT) and measuring its S_{21} parameter. This parameter helps to determine changes in the microwave signal due to the qubit-state-dependent frequency shift, thereby enabling the measurement of the qubit state.

In figure 1.4, an actual picture of the Transmon qubit can be seen, together with the readout resonator and what a successful readout looks like. In figure 1.5, a more schematic representation of the readout procedure is shown.

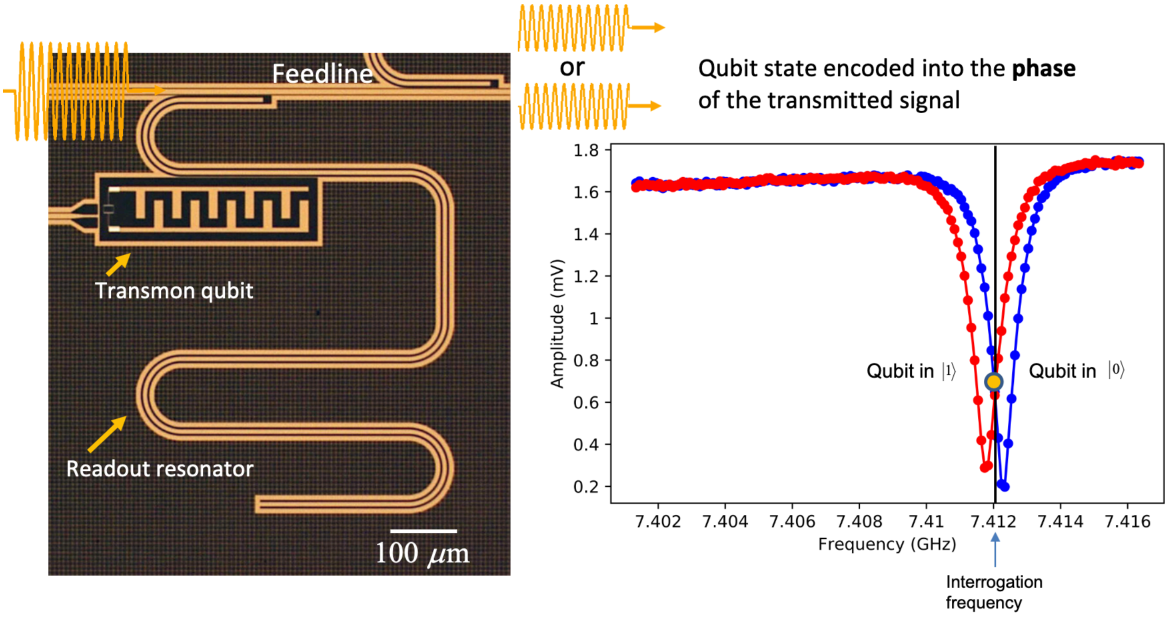

Figure 1.4: Left: image of a real Transmon qubit and the attached readout resonator; right: amplitude of transmitted signal through the qubit as a function of applied frequency [5]

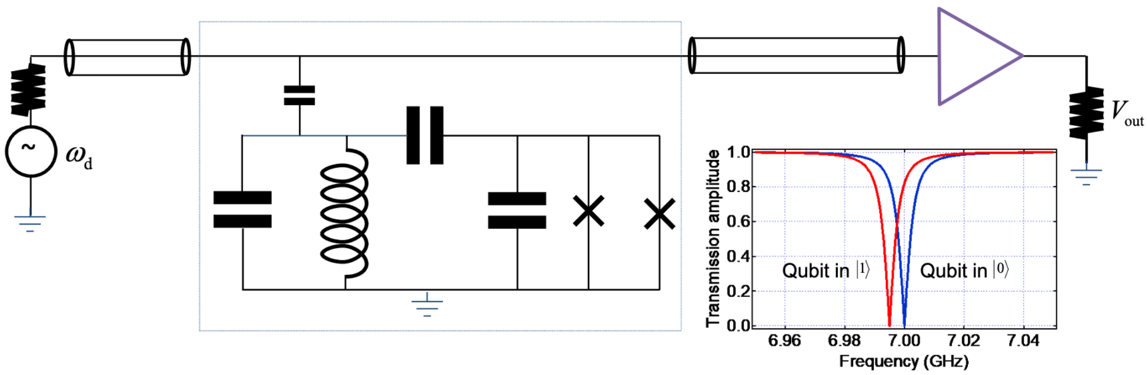

Figure 1.5: Readout of a Transmon qubit [5]

1.3. Existing solutions

Commercial VNAs from companies like Keysight and Tektronix are often quite expensive, having price tags of several tens of thousands of euros [6]. This is in large part due to their accuracy combined with a large frequency range which extends into multiple gigahertz, which requires expensive components. Extensibility is provided with equally expensive options, but the devices offer limited flexibility since users are limited to the offerings of the company for that specific model.

Cheaper options are available too, in the price range of a hundred to several hundreds of euros, but these options provide a narrower frequency range and lower accuracy [7]. Being sold in a single package, these options also do not offer much extensibility without having to study the (often open-source) documentation thoroughly.

In the field of quantum computing, VNAs are sold as quantum controllers [8][9]. These systems offer most of the flexibility that is required for qubit research, but have prices in the range of hundred thousands of euros. This is the case because of their very high accuracy and very large frequency range.

To offer much higher flexibility than the mentioned VNAs, and low to moderate prices, there have been projects on VNAs using SDR (Software-Defined Radio) technology, which recreates (expensive) analogue EMW components in software [10]. This can be done using for example a field programmable gate array (FPGA) to obtain even higher flexibility and processing speed. A hobbyist's attempt to create a VNA using SDR technology on an FPGA is well-documented on the internet [11]. There has also been a paper on an FPGA-based alternative for a VNA used for imaging in industry in the range of 200 GHz [12]. Recently, there has also been an effort to create a VNA or quantum controller using SDR technology on an FPGA [13].

1.4. Functional requirements

The requirements for the VNA of this project select the basic VNA functionality which is most useful for the application of interfacing with qubits. Omitting other functions of a commercial VNA is what makes it possible to offer a cheaper and more modular system. The system can be made using off-the-shelf RF components, an FPGA and a RF signal generator. Both the hardware design for the FPGA and the interfacing software are made open-source, to make the VNA available and customisable in the research sector. To make the interaction with the VNA understandable for the researchers, Python code is used for the user interface and API. The qualitative requirements of the entire VNA are shown below:

1. The system must have the ability to measure the S_{21} (transmission) parameter.
2. The system must be modular, so the system should work with most RF generators without any adjustments.
3. The system must be designed in such a way that it is usable by students and researchers without experience in electrical engineering.

The absolute calibration of the device is not important. It will only be used for relative measurements, because the S_{21} parameter is just a ratio between input RF signals (through-DuT or reference) and the output RF signal of the VNA.

Besides these qualitative, there are also quantitative requirements for the system:

1. The operating frequency range must be at least $4-8$ GHz.
2. Integration time per measurement point:
   - upper limit: up to 1 second per point (1 Hz IF bandwidth).
   - lower limit: down to 1 millisecond per point (1 kHz IF bandwidth).
3. Transfer overhead time to transmit the data from the FPGA to the client must be less than 10 % of the total measurement time.
4. Spurs of the signal going to the device under test must be less than 40 dBc.

Then there are some optional objectives, or should-have features, that the project should aim to achieve:

1. The system should be responsive for a human user by having a time under 100 ms between a user input/output event and a physical event happening.
2. As much open-source software as possible should be used for the project.

The specific functional requirements for the current subteam will be covered in next chapter.

1.5. Materials

A Red Pitaya STEMlab 125-14 board was used for the FPGA section, which is described as a signal acquisition and generation platform. This device contains a Xilinx Zynq 7010 System on Chip (SoC) and several connectors, such as an ethernet port, USB port, GPIO pins and four RF SMA connectors (2 input; 2 output). The SoC contains both programmable logic (PL), which uses the technology of a field programmable gate array (FPGA), and a processing system (PS), which contains an ARM dual-core processor.

For the RF section, SMA coaxial cables, RF mixers and power splitters from Mini-Circuits have been used, as well as the following RF signal generators:

- A SynthHD (V2) 10MHz - 15GHz Dual Channel Microwave Generator by Windfreak Technologies, LLC. With its two output channels, it produced both the RF stimulus signal and the RF LO signal, in one package with a single API. This generator degraded to an extent which made in unusable for the VNA, which is why it was replaced halfway the project by the following two RF generators:
- An HMC-T2100 10 MHz - 20 GHz synthesized signal generator by Hittite Microwave Corporation (now from Analog Devices, Inc.), which was used for the stimulus signal.
- An APUASYN20 8 kHz - 20 GHz Ultra-Agile Signal Source by AnaPico AG, which was used as LO.

1.6. Problem definition

To achieve the functional requirements, several engineering problems had to be solved. For this, three teams or subgroups of two students each have been formed: the RF team, the FPGA team and the software team. The RF team had to downconvert the RF signals going to the DuT and REF to IF, which then could be digitised by the ADC on the Red Pitaya and used as digital input for the FPGA team. Generators, mixers and power splitters had to be chosen which would work best to achieve the requirements. Moreover, the behaviour of these components had to be measured and documented as well as the entire power budget throughout the system. The signals that were digitised at the input of the Red Pitaya had to be converted into IQ signals by the FPGA team. Averaging was done on the FPGA to achieve the IF bandwidth requirements. Another engineering problem for the FPGA team, together with the software team, was the communication between the PL and the PS. Data from the PL had to be sent to the software team while control instructions from the software team had to be read by the PL. The software team also had to create an interface between the user and the VNA. An API and a graphical user interface (GUI) were developed for this interaction, which were part of a client program written in Python. This client also had to communicate with a Python server program running on the PS of the Red Pitaya's SoC. A schematic of the entire arrangement is shown in figure 1.6.

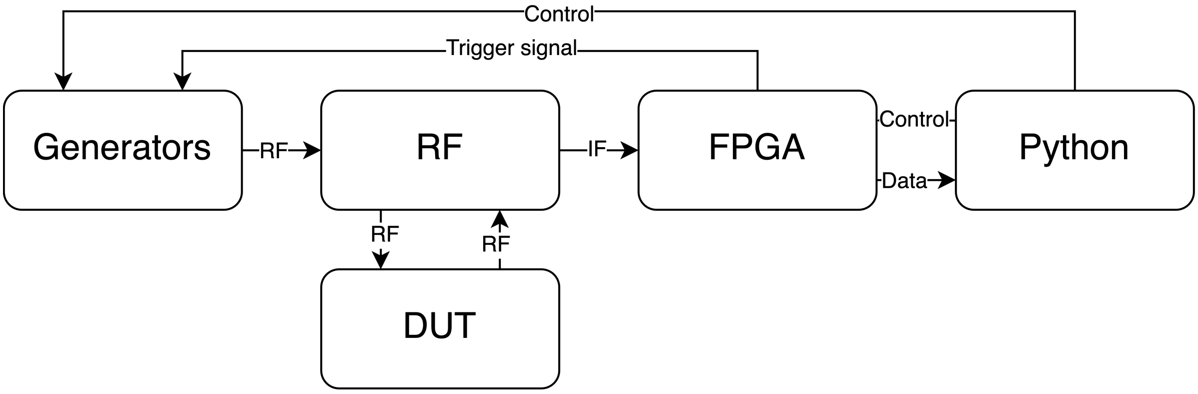

Figure 1.6: Simplified structure of input/output scheme of each team in the project

2 Specific requirements

This thesis covers the interfacing of the users with the VNA. This is done using software, consisting of a client program running on a PC, and a server-side program running on the ARM processor, from now referred to as processing system (PS), of the Red Pitaya. The programme of requirements covered in this chapter applies to these two programs.

2.1. Functional requirements

The qualitative requirements of the software, which are must-have features and must-meet constraints, are as follows:

1. The client must be able to control the RF generator(s).
2. The server must be able to transfer data from the programmable logic (PL) to the client.
3. The client must be capable of distinguishing and using only data from the PL that is obtained with all components in steady state.
4. The client must be able to send the following acquisition configuration parameters to the programmable logic (PL): time per measurement point and RF generator dead time[1].
5. The client-side program must use Python as programming language, as its intended users in the research group are familiar with it, which makes it understandable and extensible with new functionality.
6. The software should be developed in the time span of at most 280 hours (7 full working weeks), since the project is worth 10 European Credits, each of which is equivalent to 28 study hours.

The software also has these quantitative requirements to achieve, which are must-meet constraints as well:

1. The transfer of data from the PL to the client should not limit the measurement rate. Because of that, the throughput through the server-side program and the receiving part of the client-side program should at least be large enough to handle 1000 measurement points per second.
2. The total overhead in transfer time from the Red Pitaya's memory (DRAM) to the client-side data storage should be less than 10 %. In other words, when a series of measurement points of $1000\ \mathrm{ms}$ is transferred, the VNA should not have to wait for more than $100\ \mathrm{ms}$ to start a new measurement.

[1]These are defined in table 4.1

2.2. Objectives

The software developed in this project has the following objectives, which are should-have features. Although these objectives are optional and less strict than the requirements, they are still important for creating intuitive software programs for the users, and the project should therefore aim to achieve them.

1. The server and client should use and be developed as open-source software.
2. The total time from the trace (a series of IQ-measurement points) being ready to collect from the Red Pitaya's memory (DRAM) to it being transferred to the client should be less than 100 ms to make the VNA responsive enough for a human user.
3. There should be functions for for saving data and metadata in a .h5 file, which is a systematic format known by the intended users.
4. An interactive Jupyter notebook GUI should be created, which displays the data that the client receives from the server in a plot and allows users to set parameters for a measurement. This removes the necessity of manually creating programming code for quick first tests.

3

Program structures

This chapter covers the general layout of the server-side and client-side programs. Both programs were written in Python code, which is a programming language that is used on a regular basis by the intended users. This makes it easy for the researchers to extend the programs for their specific purposes. This is especially important for the client-side program, because it allows for attaching different RF generators, adding measurement configurations or collecting the measurement data in specific formats.

3.1. Server-side program

The problem that the server-side Python program has to solve, is to get the accumulated raw data from the programmable logic to the client. As mentioned in chapter 2, the requirement on the total transfer time is to have an transfer overhead of less than 10 percent. To make the transfer time as low as possible, it was chosen to build a data pipeline in the server-side program. The objective to make the VNA responsive for the user, in which the total data transfer should take less than 100 milliseconds, is also met by using a pipeline. This is the case since the data is not sent all at once to the client, but in parts with a certain number of measurement points, which makes the latency per measurement point smaller. Therefore, when the pipeline is optimised, the total data transfer time only depends on the last part of data being transferred from server to client.

In figure 3.1, an overview of the implementation is shown. The raw data comes from the programmable logic and ends at the client. It is required for the throughput through the server-side program to be at least a thousand measurement points per second. To achieve this, Direct Memory Access (DMA) is used for the first stage of the pipeline. DMA is specifically designed for fast and continuous data stream-

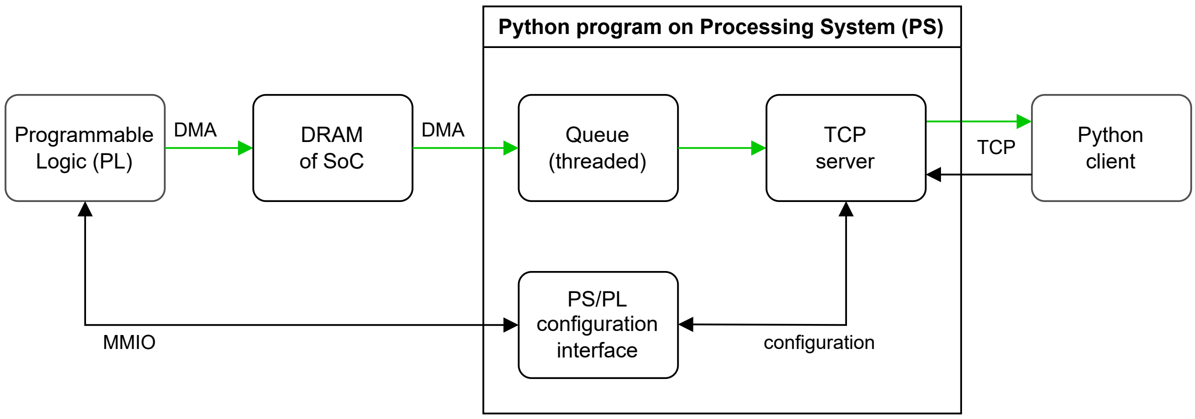

Figure 3.1: Overview of the server-side Python program with its in- and outputs; stream of acquired data indicated by green arrows

ing [14]. The programmable logic sends each measurement point of raw data to the main memory (DRAM) of the SoC. A worker thread of the Python server continuously fetches data from the memory and stores it into a queue, which is a first-in-first-out data structure built into Python.

The next stage of the pipeline is to transfer the data from the queue over the network to the client. This happens via the Transmission Control Protocol (TCP). The requirement on a throughput of at least a thousand points per second also holds for the communication between server and client. Therefore, it was chosen to optimise the amount of measurement points sent at once via the network. This topic is elaborated upon in chapter 5.

Next to the data stream, which flows from programmable logic to client, there is need for communication in the opposite direction. This concerns the parameters used inside the programmable logic, which are needed to properly configure how data is acquired. These are sent by the client via TCP to the server. The PS/PL configuration interface, explained in chapter 4, translates them into 32-bit values, such that the programmable logic can understand them. These values are stored in registers for Memory-Mapped Input/Output (MMIO). The MMIO interface was chosen because it is independent of the data stream via DMA and simplifies the configuration of the programmable logic. It is not suitable for continuous streaming at high rates, but commonly used where performance is not critical [15].

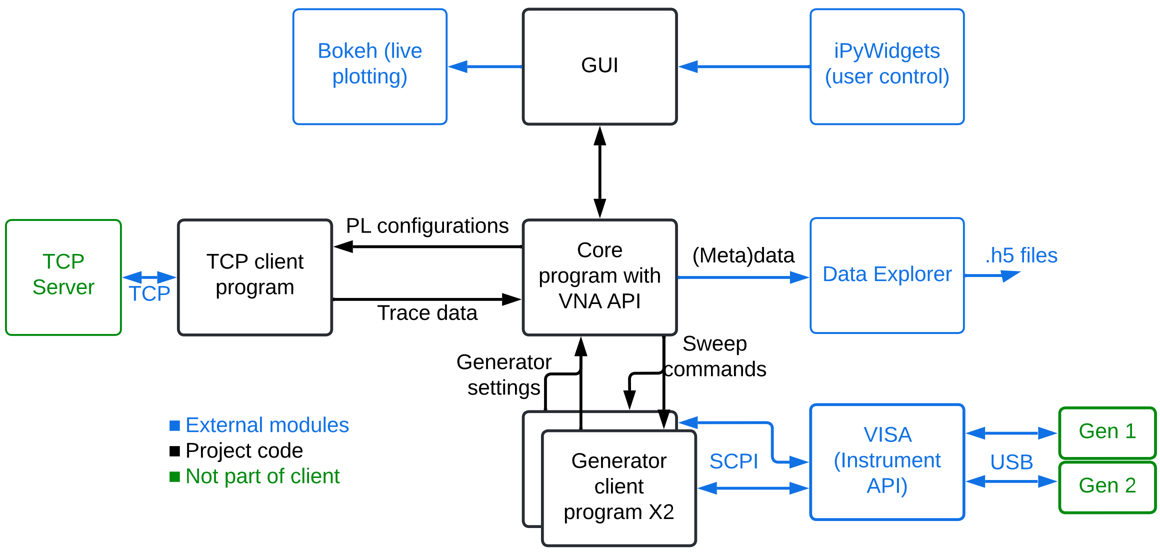

Figure 3.2: Overview of the client-side Python program

3.2. Client-side program

Figure 3.2 shows in a schematic way the different modules that form the full client-side Python program. The core program takes care of:

1. Sending the configuration commands for controlling the server-side program.
2. Sending sweep commands to the RF generators in the correct order.
3. Arranging the data for availability in the GUI and for systematic storage into external files.
4. Implementing an Application Programming Interface (API), which is a manageable list of VNA functions which can directly be called from external (Python) programs.

The core program is mainly important for doing all these tasks in the correct order, such that the user only has to give measurement commands via the API, in a manner which is done on conventional VNAs. Such a command would be for example to execute a frequency sweep from $4\ \mathrm{GHz}$ to $8\ \mathrm{GHz}$, with frequency steps of $1\ \mathrm{MHz}$, a time step of $10\ \mathrm{ms}$ and a constant power of $0\ \mathrm{dBm}$. This command is sent using the *sweep_acquire_2_generators* function as seen in section A.2.1, with the corresponding parameters. After this command has been received, the program starts with execution in the following order:

1. Opening the connections to the server and to the generators using Python's "with"-statements.
2. Sending the required settings for the generators with *hardware_freq_sweep* functions (from the RF generator client programs).
3. Requesting the useful settings back with *read_status* functions (generator client programs) to save them later in the metadata file.
4. Sending the settings for the PL using the functions *send_tpp*, *send_dead_time*, *send_trigger_length* and *send_trigger_config* (TCP client program).
5. Turning on the generators using *perform_sweep* functions (generator client programs).
6. Requesting data from the PL using *start_acquisition* (TCP client program).
7. Putting data received via TCP into a Python queue that will temporarily store the entire measurement in memory using the function *receive_data*.
8. Automatically closing the connection to both server and generators after exiting the "with"-blocks.
9. Calculating and arranging the data using the function *construct_output_data* (discussed below).
10. Saving the data using the function *save_data* (discussed below).

The *construct_output_data* function first calculates the S_{21} values for every frequency, by doing the complex division from chapter 1, repeated in equation (3.1). The function then stores the real part, imaginary part, magnitude (in dB) and phase of S_{21} in a matrix. This matrix also contains the frequency steps and the real part, imaginary part, magnitude and phase of both the reference signal and DuT signal.

$$S_{21} = \frac{b_2}{a_1} = \frac{I_{\text{trans}} + \text{j}Q_{\text{trans}}}{I_{\text{ref}} + \text{j}Q_{\text{ref}}} \tag{3.1}$$

The *save_data* function takes this matrix with information and only saves the frequency steps, the magnitude of S_{21} in dB and the phase of S_{21}. The option to save the other data is left to the end user. This saving is done using the DataExplorer Python package [16], which has been made by members of the research group. This package takes care of generating a time-tagged folder containing the Python code used by the end-user and saving the data as "Xarray" [17] in .h5 data format. The *save_data* function then manually adds the metadata in the same folder, also as "Xarrays" in .h5 data format.

The TCP client program takes care of communication with the server-side program using the Transmission Control Protocol over ethernet. This communication consists of sending the configurations for the PL, and receiving measurement data which have been processed by the server-side program. The TCP data throughput, discussed in chapter 5, has been optimised to achieve the requirement from section 2.1.

The RF generator client program takes care of translating a command, such as "do a sweep with certain parameters", into the right order of commands [18] for the specific generator, and sending it using the VISA API [19]. It also collects generator status information using the same procedure. There is a generator client program for each of the two generators that are used in the VNA. The working and implementation of SCPI and VISA in our code are discussed in chapter 6.

The GUI is implemented in the interactive Python notebook format (.ipynb). It provides a ready-to-use interface to the VNA, so no code has to be written to perform basic frequency sweeps. It does this by interacting with the API of the client's core. The controlling of the VNA is done using buttons and interactive text-boxes created using the iPyWidgets library [20]. Showing the trace as the data is coming in is implemented using the Bokeh plotting library [21]. A more detailed description of the GUI is discussed in chapter 7.

4
PS/PL interface

The Python server-side program running on the processing system needs to be able to communicate with the programmable logic. This communication is performed with a set of Python functions that use the PYNQ library, which is written for platforms like the AMD Xilinx Zynq 7010 System on Chip on the Red Pitaya used in this project (containing both the PS and PL). PYNQ provides the ability to control and communicate with the PL [22] in different ways. This chapter explains how communication via DMA and MMIO registers is implemented, and discusses the performance of the resulting server-side program in relation to the requirements.

4.1. Before communication

The PL has to be configured before the Python server is able to communicate with it. With PYNQ, this can be done by loading a hardware library, also called overlay. An overlay consists of a .bit file and a .hwh file, produced by the hardware designers in the FPGA subteam. The .bit file contains the information on how the hardware should be configured, and the .hwh file contains metadata about the cells, for example which interfaces (DMA, MMIO) are connected to which hardware blocks. PYNQ has a class *Overlay*, which, when initialised with the location of a .bit file, loads the overlay onto the PL.

4.2. DMA: continuous data streaming

Direct Memory Access [14] is a fast method for continuously transferring data from the programmable logic via the memory of the SoC to the processing system. In this project, it is used for the raw acquired data from the PL, which must be able to flow at least at a rate of one measurement point per millisecond, as per the requirement in section 2.1. DMA allows communication in both directions, but since the PL does not need large amounts of data at high rates, only the direction from PL to PS is implemented.

4.2.1. Averaging

The user running the client-side Python program expects data in the form of four values at each measurement point (frequency): I_{trans}, Q_{trans}, I_{ref} and Q_{ref}, with which the parameter S_{21} can be calculated (see equation (3.1)).

The PL provides for each I and Q value the sum accumulated during the measurement point, and a counter value representing the amount of additions the PL did to arrive at that sum. Hence, the raw acquired data has to be converted to the wanted format somewhere in the data pipeline. It was decided to split this conversion into two parts: averaging the I and Q values inside the Python server, and performing the complex division to get S_{21} inside the Python client. The reason for not performing all calculations inside the server was to give the users of the VNA the option to save the four I and Q values for DuT and reference separately, together with the S_{21} values. Averaging was performed inside the Python server since it depends on the data format provided by the PL, of which the client has no knowledge.

The raw data for a single I or Q value consists of two signed integers and one unsigned integer. The two signed integers have to be added together to get the sum a, and then divided by the unsigned integer (counter, c) to get an average for that I or Q value, as shown in equation (4.1) (which holds for all mentioned I and Q values).

$$I = \frac{\varphi a}{c} \text{ (V)} \tag{4.1}$$

The constant scalar φ is used to convert the units to volts. It depends on the maximum input of the analog-to-digital converter (ADC) of the incoming signal, being 1 V, and the amount of bits of precision the ADC has to represent this maximum input, being 13 (the sign bit is excluded because it does not add precision). Due to the digital demodulation and filtering (for details, refer to the thesis of the FPGA subteam), a factor of $1/2$ is applied to the value of a and the precision is doubled to 26 bits. The maximum input signal of $I_{\text{max}} = 1$ V gives the average

$$\frac{a}{c} = \left(\frac{1}{2}\right)(2^{26}) = 2^{25}$$

and therefore,

$$\varphi = \frac{cI_{\text{max}}}{a} = 2^{-25} \text{ (V)}$$

4.2.2. Python implementation

In the Python server-side program, a worker thread can be enabled to continuously perform DMA data transfers. Each data transfer concerns $4 \cdot 3 \cdot 32 = 384$ bits, being three 32-bit integers for each of I_{trans}, Q_{trans}, I_{ref} and Q_{ref}. A data transfer involves calling two PYNQ methods: *dma.recvchannel.transfer*, which lets the PL know it can write data to an address in DRAM specified by a given allocated buffer, and *dma.recvchannel.wait*, which waits until the PL has finished writing the data. When a transfer is completed, the data is averaged by applying equation (4.1) four times: to calculate I_{trans}, Q_{trans}, I_{ref} and Q_{ref}. These four values are stored inside a Python queue, which is a built-in data structure suitable for multithreading [23]. The TCP server, discussed in chapter 5, running in the main thread, can retrieve data from this queue when the client requests it.

4.2.3. Throughput

A time per point of one millisecond is the lowest data acquisition period for this project. For this value, the requirement stated in section 2.1 is that the throughput of the data transfer from PL to PS should be at least 1000 points per second. To verify whether the DMA method and the Python implementation are fast enough to achieve this data transfer rate, an experiment has been performed. All entries were removed from the Python queue, the time per point was configured and the worker thread was started. A timer measured how long it took to fetch and average 15000 data points, which can be used to estimate the throughput. This experiment has been repeated for different values for the time per point, and repeated when averaging the four values was skipped.

In figure 4.1, the results of the experiment are shown. For values of the time per point below one millisecond, the effect of including the averaging is visible. Without averaging, the maximum theoretical throughput of continuous DMA transfers (shortest time per point) is

$$\frac{384 \text{ b}}{2872.3 \text{ µs}} = 668.5 \text{ kb s}^{-1} = 1741 \text{ transfers per second}$$

At a time per point of 1 ms or longer, the overhead due to averaging is negligible, since the bars have almost the same height. This justifies the choice to directly perform averaging after DMA transfers before storing data in the queue. The implementation is capable of averaging the values fast enough before the next DMA transfer is completed, and with this, the requirement for the throughput of 1000 points per second from PL to PS is met.

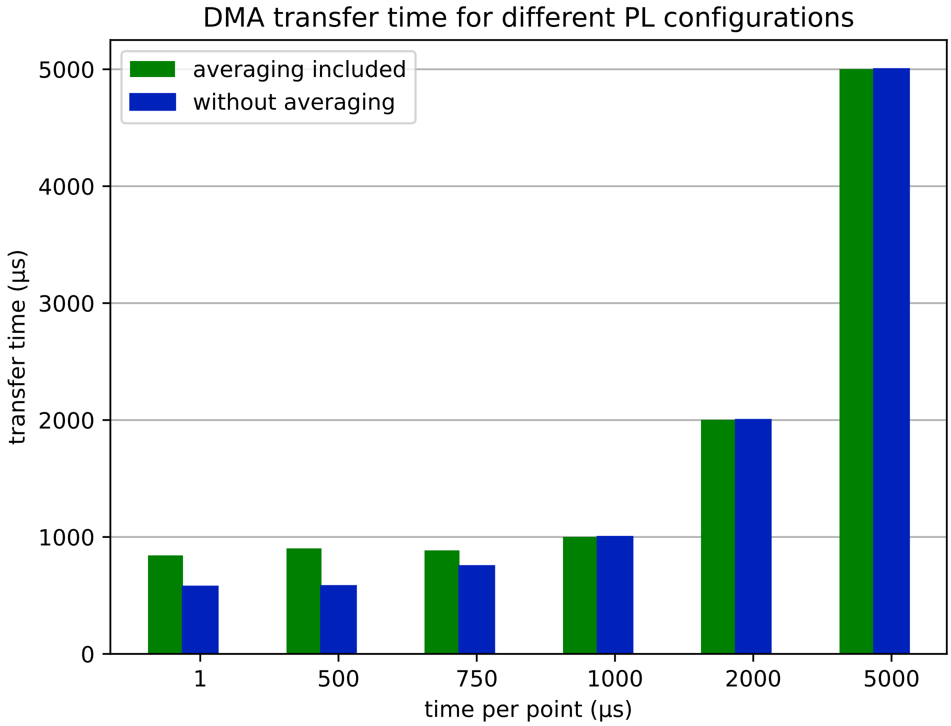

Figure 4.1: Average transfer time via DMA for different values of the time per measurement point; each average based on 15000 transfers; averaging is applying equation (4.1) four times.

4.3. MMIO: configuration

Sending configuration parameters to the PL is done using Memory-Mapped Input/Output registers. This is a simple method suitable for communication in which high data rates are not required. The configuration parameters, all listed in table 4.1, need to be known by the PL before starting a series of measurement points. The PS/PL configuration interface consists of functions inside the Python class *PLInterface* (found in section A.1.1). These functions need to translate the parameters to the correct binary values and write them to the MMIO registers, which are PYNQ objects with corresponding methods to read and write data.

The parameters time per point, generator dead time and trigger pulse length can be controlled with a precision of one microsecond. The PL does not count in microseconds but in clock cycles, which is why the input values in microseconds need to be multiplied by the PL clock frequency of $125\ \mathrm{MHz}$. These multiplications are defined in the class *PLConfig* (found in section A.1.3) and can be customised for each parameter.

Table 4.1: Configuration parameters needed by the programmable logic

parameter	description
time per point	clock cycles per measurement point during which the PL digitises, demodulates and accumulates
generator dead time	clock cycles representing switching time at start of each measurement point when the frequency is not stable and the PL does not yet accumulate
triggers pulse length	pulse length in clock cycles for the two digital output signals used as triggers for the RF generators
triggers configuration	settings for the two output triggers: active-high or active-low pulse, pulse on each point and/or pulse only at the start of the measurement
data acquisition status	control bit to enable or disable the continuous data acquisition inside the PL and transferring the data via DMA

The reason for including an option in the trigger configuration to give a pulse on each point was specifically included for the SynthHD generator [24], with the reasoning that this would provide better synchronisation. However, not every RF generator supports such an option, as is the case with the HMC-T2100 [25]. The description on how this changed the code will be discussed in section 6.4.

The configuration parameter which controls the PL's outputs for the two digital trigger signals is special, since it contains three parts inside one MMIO register: bits 28-31 for the trigger configuration of trigger 0, bits 24-27 for the trigger configuration of trigger 1, and bits 0-23 (least significant) for the pulse length, which applies to both trigger outputs. Control logic has been written in the function *change_config* (found in section A.1.2) to write the correct values to the MMIO register. Each sequence of bits can be written independently and leaving the others unchanged. The other MMIO registers, which each contain only one parameter, need to be completely overwritten when this one parameter changes.

Adding another parameter for the number of measurement points that the PL takes before pausing itself has been considered. However, this has not been implemented, since it makes the programmable logic less flexible, as an additional digital counter is required and an option to take an infinite amount of measurement points. Instead, when a measurement is finished, the Python client detects that the correct number of points have been transmitted and sends a signal – details are in chapter 5 – to stop the acquisition.

4.4. Software testing

For software projects like this, automated testing of written code is an useful tool to verify that modifications do not cause issues. Especially for the server-side Python program, which uses the PYNQ library that only works on platforms like the AMD Xilinx Zynq 7010 SoC to be able to interface with the PL, tests are important. Debugging a multithreaded program becomes increasingly more difficult with additions to the code. Python testing involves writing small test functions that run (part of) the source code and check using “assert” statements whether the source code has executed correctly and does not generate exceptions, also in edge cases.

Pytest [26] is a tool that performs testing of Python test functions written by the user. It can be run with a single command “pytest”, gives detailed feedback when exceptions occur and allows manual debugging. In combination with mocking, a technique commonly used by software engineers to modify the functionality of parts of the source code, the server-side Python program, including DMA, MMIO and TCP transfers, has been tested. The code can be found in section A.1.7 and A.1.8. On platforms which do not have access to programmable logic, like the computers of the developers, mocking has been implemented to simulate parts of the PYNQ libary. Custom versions of the used PYNQ classes and methods have been written by the developers, for example the function *dma.recvchannel.transfer* that simulates a DMA transfer. Instead of accessing the PL, this function sets the allocated buffer to a known test value. When the test functions are executed by pytest, the mocked functions (see section A.1.6) are called instead of the functions from the actual PYNQ library.

5
TCP

This chapter explains how the Python server-side program and a module inside the Python client-side program communicate with each other, and discusses the data throughput of the connection.

5.1. Data transmission protocol

The Transmission Control Protocol (TCP) is a basic protocol for communication between two devices via a network connection [27]. It is protocol at a lower level than for example the HyperText Transfer Protocol (HTTP), which means TCP has less overhead and can therefore be optimised for a specific application. TCP guarantees that the packets sent over the network will arrive in the same order as they were sent. Similar network protocols, for example UDP (User Datagram Protocol), do not guarantee this. For sending large amounts of data via TCP, the need to label each data point is therefore redundant, just as checking if all packets actually have arrived. Because of this, implementing a TCP client-server model in Python is straightforward, meaning more time can be invested into other aspects of the project, reflecting the time constraint from section 2.1.

5.2. Communication protocol

The pipeline inside the Python server-side program from figure 3.1 shows that the TCP server governs the communication between the PS and the Python client. In order to have functional communication, both server and client need to obey a set of rules. This set of rules, or communication protocol, has been written with the objectives of extensibility and regularity. For the purpose of regularity, a basic agreement is that a client always initiates a connection with the server, upon which the server performs a task based on the client's command and always responds to the client. The tasks are split into configuration changes and requests, and are shown together with server responses in table 5.1. The server sends configuration changes to the programmable logic, and responds to requests with data from the queue.

The configuration changes are sent with a value to the server via TCP. For example, "p5000" is a command to set the time per point to $5000\ \mu s$, during which the PL accumulates before the data is transferred via DMA. The server will respond with the "OK" message if a configuration change was successfully handled. The Python methods for sending commands for configuration changes can be found in the class *TCPClient* in section A.2.3. The method in the server that handles the configuration changes is *change_config*, found in section A.1.2. A special implementation was required to implement the commands "r1" (to enable the data acquisition) and "r0" (to pause the data acquisition). When one of these commands is sent, the server needs to start or pause DMA data transfers. Starting a DMA transfer is simple, but correctly pausing is more difficult, since the programmable logic does not support aborting a currently running transfer. If this is accidentally done, no more DMA transfers can be performed until the PL overlay is completely reloaded, which is to be avoided since it deletes all data inside. Hence, the current DMA transfer first had to be completed before being able to pause. What

Table 5.1: Communication protocol used between client and server; configuration commands require a numerical value behind the letter

configuration commands	description
p	Sets time per measurement point (μs) during which the PL accumulates.
g	Sets generator dead time (μs): length of period at start of each measurement point during which the PL does not yet accumulate.
r	Enables/disables continuous data acquisition and transferring via DMA in PL.
t	Sets (primary & secondary) trigger output pulse length (μs)
c	Primary trigger output configuration (active-high or active-low, trigger per point and/or trigger per frequency sweep)
o	Secondary trigger output configuration (active-high or active-low, trigger per point and/or trigger per frequency sweep)
requests	**description**
d	Request data from the server queue.
q	Request the size of the server queue.
T	Request the SoC temperature (℃) of the server[1].
server responses	**description**
*	OK: server performed the task successfully.
?	Server received an unknown command or an error occured during execution of a task.

made it harder was the fact that DMA transfers are continuously being performed by a worker thread, as described in section 4.2.2. The problem that arose, concerned the signalling between the main thread and worker thread. If a signal is sent to the worker thread, telling it to pause after completing the current DMA transfer, the time before the worker thread actually received this signal is unknown to the main thread. This can cause disruptions if a new DMA transfer is started during this time. A solution was to let the worker thread reply with a different signal that it had indeed completed the DMA transfer, so the main thread could continue responding to the client via TCP. Although this signalling takes more time, the requirement on the data throughput does not apply when sending configurations to the PL, since these configurations are only sent at the start and end of a measurement.

When a client performs a request, for example a data request, “d”, it only sends the letter via TCP. The server will respond with data from the queue. The code, found in section A.1.2, has been written in such a way that it is extensible. All commands are defined in the file *protocol.py* (found in section A.1.3), and new commands can be added to the class *TCPCommandProtocol*, while new functionality can be added by creating methods in the class *TCPDataServer* in *tcp_server.py*.

5.3. Python implementation

The Python class *TCPDataServer* contains an implementation of a TCP server, based on [28]. This server listens to a Python socket object, accepting the connection from any client on the network. To avoid complexity, the server is not able to communicate with multiple clients simultaneously. For this project, connecting to multiple clients is not required, since only one measurement can be performed at once, hence the acquired data only needs to be sent to one client. As mentioned, the server performs a task based on the client’s command, and then sends a response. For tasks that are performed the most often, like requesting measurement data, the implementation should be time-efficient. This task consists therefore only of fetching the averaged data from the queue, encoding the values in bytes and sending them over the network to the client. The data is encoded to reduce the size of the TCP packet. Each measurement point, consisting of four double-precision (64-bit) floating point Python objects, are converted into one sequence of 256 bits ($256\ \mathrm{b}$) or equivalently 32 bytes ($32\ \mathrm{B}$) using the UTF-8 encoding. This is done using the built-in *struct.pack* function, for which the code can be found in section A.1.4. To reduce the amount of TCP packets with data the server needs to send to the client, each

TCP packet consists of at most 45 measurement points. This optimal value has been experimentally determined in section 5.4. The client has to decode the sequence of bytes before performing floating-point operations on it and saving it.

The server is implemented with enough exception handling, such that it will keep running when a client sends an unexpected command or when the connection fails. During a connection, the TCP client is able to send multiple commands one after another. The client implemented in this project uses the following order of operations for successful data transfer: it connects to the TCP server using its hostname and port and transmits all necessary configuration parameters for the PL, which are set by the user. The next command it sends is “r1” to start data acquisition, which will enable the PL and start filling the server’s data queue. During the next phase, the client sends data requests (“d”) until it has received enough data points for the current measurement. The server has no knowledge of this number, and therefore, the client will send the stop acquisition command “r0” when it is ready, to disable the PL and data transfer via DMA. The PL gives an output trigger to the generators and knows the dead time specified with the configuration command “g”. Therefore, the PL will only collect data while all components are in steady state. Although this data is propagating through the pipeline and therefore received at a later moment in time by the client, the client still receives the correct data, and therefore fulfills functional requirement 3 from section 2.1. The data still left in the pipeline after the client has sent its stop acquisition command, is discarded when starting a new measurement.

5.4. Throughput

One of the requirements described in section 2.1 is that the client must receive new measurement data (if available) with a throughput of at least a thousand points per second. TCP defines a default size for sending small packets via the network. If the data size is larger, the packet is split up into multiple packets. This introduces a decrease in the transmission efficiency of information. The throughput therefore depends on the amount of measurement points sent in one packet. An experiment has been performed to estimate this throughput. A total of ten thousand points, each of $32\ \mathrm{B}$ long, was stored inside the server-side data queue and transferred to the client. This has been repeated for different amounts of points per packet. The average time needed for sending one point and the average time needed for sending one packet has been plotted in figure 5.1.

The green curve in this figure clearly shows the inverse proportional relationship between the number of points per packet and the transfer time. The requirement for the TCP throughput was to be able to handle one thousand measurement points per second. Even when sending only one point ($32\ \mathrm{B}$) in each packet, which is where the green and blue curve in the figure intersect, the theoretical data throughput is approximately

$$\frac{32 \cdot 8\ \mathrm{b}}{91\ \mathrm{\mu s}} = 2.8\ \mathrm{Mb\ s^{-1}} = 11 \text{ thousand points per second}$$

This number only considers the throughput from server to client, assuming the client only receives data. In this project, the client asks for every data packet, which limits the throughput. When sending multiple points per packet, this effect becomes negligible since the amount of bytes sent from server to client becomes much larger than the one byte (the command “d”) the client sends to the server. The blue curve shows that for an increasing number of points per packet, the average transfer time slowly rises. This is an effect of pipelining in the network: TCP allows a so-called window of multiple packets to be sent in succession without the first packet having arrived at the destination.

After the boundary of 45 points per packet, which amounts to $45 \cdot 32 = 1440\ \mathrm{B}$ (a common value for TCP), the data becomes too large to fit in one TCP packet. With more points, the data is internally split up before being sent, and recombined upon being received. Sending packets which are almost empty decreases the transportation efficiency and is therefore to be avoided. An objective mentioned in section 2.2 is that the client should receive data, if available, at least every 100 milliseconds. For the smallest time per point, where new data is available every millisecond, the amount of points per packet

[1] The temperature of the SoC will be saved in the metadata attached to a measurement.

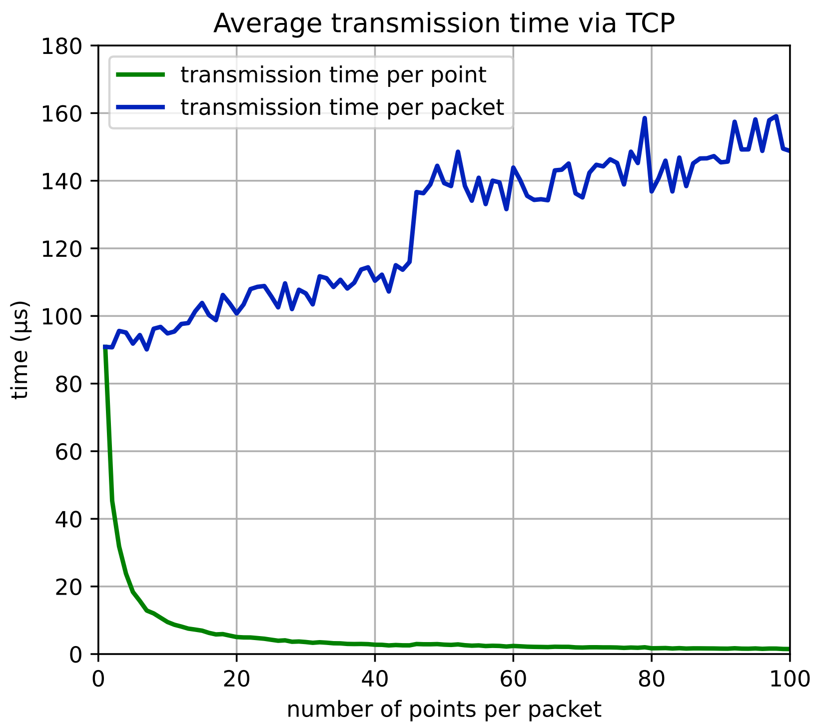

Figure 5.1: Average transmission time via TCP for different amounts of points (32 B) per packet; server and client connected via FritzBox 4020 router; each average based on sending 10^4 points

is 100 at maximum. It was chosen, for optimum transmission efficiency and to reach this objective, to set the maximum amount of points per packet to 45. The corresponding data throughput is given in equation (5.1).

$$\frac{32 \cdot 8\ \mathrm{b}}{2.57746\ \mathrm{\mu s}} = 99.32\ \mathrm{Mb\ s^{-1}} = 388\ \text{thousand points per second} \tag{5.1}$$

This value is certainly limited by the used router (Fritzbox 4020), which allows data rates up to $100\ \mathrm{Mb\ s^{-1}}$ via ethernet. It also does not take into account the overhead due to storing data into the server queue, since in this experiment, the queue was filled before transmitting any data over the network.

Using 45 measurement points per packet satisfies the requirement of a throughput of one thousand points per second. The maximum of 45 points per packet does not apply at higher times per point. If only one point is available after 50 milliseconds, it will be sent to the client directly. For a time per point higher than 100 milliseconds, the objective of at most 100 milliseconds between received data point is not achievable, since the data is not available at this rate. In this case, the data is sent to the client as soon it is available in the server queue.

The final functional requirement from section 2.1 states that the total transfer time overhead from DRAM to client should be less than 10 %. Because the data is pipelined inside the Python server, the overhead is approximately equal to the transfer time of the last packet that is sent to the client. This transfer time, assuming 45 points per packet, is $116\ \mathrm{\mu s}$. The requirement is therefore satisfied for any measurement that takes longer than $1.16\ \mathrm{ms}$. This applies to almost all useful measurements, since the time per point is at minimum $1\ \mathrm{ms}$.

6

Generator communication

The client-side program communicates with the RF generators using commands written in “Standard Commands for Programmable Instruments” (SCPI) language [18], and communicates it via the “Virtual Instrument Software Architecture” (VISA) API [19]. Both are maintained by the Interchangeable Virtual Instruments (IVI) foundation. This chapter describes in short what these standards are, and goes into detail about how these are implemented in the client-side Python program.

6.1. SCPI

SCPI is a standardised set of syntax and commands used for instructing any kind of programmable test or measurement instrument. It uses keywords which are grouped in several levels of subsystems, which leads to a command being expressed in a hierarchical representation. The AnaPico APUASYN has, for example, a reference oscillator (ROSCillator) subsystem, which contains an external oscillator (EXTernal) subsystem, which has an expected frequency (FREQuency) command [29]. The total command for changing a setting (configuration command) of this SCPI-compatible device is then:

```
ROSC:EXT:FREQ 10MHz
```

The capitalised abbreviations can be used, or the full keywords can be written, and all commands are not case-sensitive. In this command, a setting is given after a whitespace (in this case a value in MHz). SCPI also supports reading settings of the connected device (a query command). This is done by including a question mark, such as:

```
ROSC:EXT:FREQ?
```

The configuration and query commands have the same syntax for every instrument, and if those instrument have the same parameters to be set, the commands for those parameters are the same. The last group of commands are the mandated commands, which have to be available for all SCPI-conforming instruments according to IEEE 488.2 [30], which is a standard for instrument communication that acts as a precursor of the SCPI [18] standard. These commands have configuration and query formats, but are never in a subsystem. They also always start with an asterisk. One of the commands used in the generator client programs is:

```
*IDN?
```

This command is used to obtain the name of the generator with which is being communicated. In this project, it is included when saving the settings of generators in metadata files.

The choice of using SCPI was made because this happened to be the protocol that the two RF generators that were used in the final prototype are equipped with [25][29]. Besides this, a large advantage of SCPI is that it is the industry standard, so most RF generators use it for communication with a client [31]. This means that the generator client programs can easily be adjusted to work with other SCPI-compatible RF generators, which contributes to the extensibility of the VNA. This interchangeability of the commands was used by reusing a large part of the commands for the HMC-T2100 in the generator client program of the APUASYN.

For the implementation in Python, the commands were written as strings, with the PyVISA package taking care of getting the commands to the generators (also see next section). The code in sections A.2.5 and A.2.4 show this method, with ".write" functions from PyVISA taking care of sending the configuration commands and ".query" functions sending the query commands and returning the requested values. For performing the frequency sweep as described for the core program in section 3.2, the commands for the generators have been grouped in three functions, which have the same name in the code in sections A.2.5 and A.2.4. These functions should be implemented for each new generator that is attached to the system, and may slightly differ in the commands they use.

The first function is *hardware_freq_sweep*, which sends all necessary settings for a frequency sweep, without turning the generator output on yet. Between the code for both generators, some commands were shared, like:

```
POW:AMPL {power}DBM
```

This command was used to set the output power for each generator, with *{power}* being replaced by a Python string with the required numeric value. A remarkable difference was the fact that the APUASYN, in contrast to the HMC-T2100, did not seem to accept the command to set the size of each step in the frequency sweep:

```
FREQ:STEP {freqstep}MHz
```

Thus, the number of steps in the sweep was calculated in the code of section A.2.4 with the help of the required step size, and was then sent using the following command:

```
SWE:POIN {points}
```

This command, on its turn, does not exist for the HMC-T2100. The APUASYN also required additional settings for turning on the external reference clock, which the HMC-T2100 detected automatically after being connected to such a clock signal. One such command was shown at the start of this section, the other is:

```
ROSC:SOUR EXT
```

This command explicitly tells the APUASYN to use the external reference clock instead of the internal clock.

The second function is *perform_sweep*, which is a separate function to turn on the generator output. This was made a separate function to give more control on the moment to turn both generators on, so that this could happen as close in time as possible. Two commands take care of this:

```
OUTP 1 and INIT:IMM
```

For the APUASYN, one command for setting the sweep type had to be sent after the last 2 commands to work:

```
FREQ:MODE SWE
```

The reason why this command did not work before the *OUTP* and *INIT:IMM* commands is not clear.

The third function is *read_status*, which takes care of generating a dictionary with all the settings of the generators. The number of queries made by this function is larger for the APUASYN, because this generator has more settings than the HMC-T2100. One such extra query is:

```
ROSC:LOCK?
```

This query gives assurance that the RF output is generated with a clock locked to the indicated reference clock, which in the case of this project is the external clock. With the HMC-T2100, there was no way to know this using a query.

6.2. VISA

VISA is an API that provides an universal method to communicate with instruments over different interfaces and bus systems, such as USB, PXI, GPIB and ethernet [19]. Each instrument that is attached to a program using VISA is classified as a “resource” [32]. The API offers an option to open the required resource, which establishes a connection. The API then offers the functions to interact with the resource, such as writing or querying. After all communication has been done, the resource can also be closed with a function. Besides this basic functionality which is used in this project, VISA also offers many specific functions that are used for locking, getting streams of data from instruments, specifying the format of data from a large list of options, changing bus settings, and so on [19].

The use of VISA in the client-side program is done with the help of two Python packages. PyVISA-Py [33] is the implementation of the VISA specification in Python [34]. The other package, PyVISA, provides access to the commands of PyVISA-Py and other VISA libraries, such as NI-VISA [32]. This means that PyVISA can be configured to use the VISA library that is required by the user. It are only the commands of PyVISA that are directly called from the code written for this project. Using PyVISA, a resource manager object is created every time the generator client program is started. This object has a list of all resources. These resources contain in principle all devices that are connected using one of the protocols that are supported by VISA, so a small loop was made to find the name corresponding to the RF generator. The rest of the steps as described above, such as opening a resource and writing to it, are done with single Python functions.

The choice for VISA was made because it removes the need to make separate programs for communication with specific protocols, which makes it easier for the user to choose the protocols and busses that best fit the application. The implementation in Python also requires just a few lines of code, which makes it easier to understand and adjust the code to the needs of the user, and requires less time to test and debug. The choice for using PyVISA-Py specifically was made because it is open-source and free.

6.3. Physical connection

For the AnaPico as well as the Hittite, both ethernet and USB were available [35][36] for the communication. The Hittite also has the GPIB-bus available, but this port was not used because it was less familiar.

For USB, the AnaPico uses the USB Test and Measurement Class (USBTMC) [37] protocol over its USB-bus. This protocol is specifically made to make the USB-bus work with VISA without additional configurations [38]. Devices with USBTMC use the USB INSTR resource class, which is supported by the PyVISA-Py implementation of VISA [33]. For making USB work with PyVISA-Py, the PyUSB Python module has to be downloaded [39], which in its turn relies on an USB driver library such as the open-source libusb written in C [40]. The Hittite actually uses a non-USB serial communication protocol over the USB interface.

For ethernet, the AnaPico supports several Local Area Network (LAN) interface protocols: TCP sockets, Telnet and VXI-11, while the Hittite only supports the former two [25]. However, to enable communication via ethernet, first a connection via USB has to be made anyway, to write and query the DHCP or static IP settings of the device. Because of this, USB was chosen over ethernet for communication during this project. Furthermore, the limited amount of working hours for the project, as stated in the requirements in section 2.1, let to this decision.

6.4. Windfreak SynthHD

As mentioned before, the SynthHD was used before using the Hittite and AnaPico. For this generator, a generator client program was also written (see section A.2.7), that works with the project code as is shown in 3.2, but does not use SCPI and VISA. The two SynthHDs available during the project both started malfunctioning after a lot of time was already spent on getting these generators to correctly perform a frequency sweep. The code created for the communication with this generator will be discussed in short below. It makes use of Python code written by Windfreak Technologies, which provides an API, shown in section A.2.8.

Instead of using VISA, the SynthHD client program makes a direct connection with USB. It does this by using the PySerial module [41], which has been implemented in the Python code behind the API. Although this could have been implemented using VISA, it was not needed yet by that point. It would not have been possible to use SCPI for the SynthHD in that case, because the SynthHD uses serial communication with a communication protocol defined by Windfreak Technologies itself [24]. This protocol differs from SCPI by using single characters instead of 3- or 4-character commands with colons, and by not using any subsystem hierarchy. Like SCPI, it uses a question mark for queries and a value given after a whitespace if applicable. This protocol was fully implemented in the API program provided by Windfreak Technologies, so in practice, commands for interaction with the SynthHD were set by calling the functions seen in A.2.8 with the required parameters in the code of section A.2.7.

The generator client program of the SynthHD had the option for triggering the frequency sweep for every step implemented, as can be seen in the function *triggered_diff_freq_sweep* in section A.2.7, where the setting for *trig_function* is set to 3 to enable step triggering. This option was also available on the APUASYN [29], but not on the HMC-T2100 [25]. After the SynthHD generator was no longer used, it was decided to not implement this option in the other generator client programs, because it would require extra effort to get the generators to sweep synchronously when one of them would use this “step triggering”, and the “full sweep trigger” worked anyway.

7 GUI

One of the objectives of the project was to create a Graphical User Interface (GUI). This chapter describes in short the choices that have been made for this, and the implementation of it in Python.

7.1. PySide

For showing the GUI, the choice was made between two methods. The first method was showing a separate window on the local client, with a GUI made using Qt and its Python implementation PySide. PySide comes with a designer application, in which a .ui file can be edited in a graphical environment. But after the .ui file is converted into a Python file, it is quite extensive and requires the knowledge of the PySide functions to be able to edit it. A start was made on this method, with a GUI showing buttons (found in section A.2.10) and a GUI showing a live updating plot (found in section A.2.11). Debugging and editing the code of these converted files was difficult and not flexible, hence it was decided to abandon this method for the GUI.

7.2. Jupyter

The second method was locally hosting a Jupyter Python notebook, and showing the GUI in the output window. This option was chosen in the end because it offers more flexibility in customising the GUI and because the intended users are already familiar with Jupyter notebooks and several packages and libraries that are used in combination with it. Below, the choices for setting the VNA command and showing the data are motivated.

7.2.1. Implementation

For setting the VNA commands, widgets with buttons and text boxes were chosen over a command line interface, because it gives a better overview of the settings when doing quick measurements, or demonstrating the VNA. The widgets come from the iPyWidgets package, also known as just “Jupyter Widgets” [20]. Again, this choice was also motivated by the familiarity of the end users with iPyWidgets.

Each of the widgets is implemented as an object of the corresponding widget class, as can be seen in the code of section A.2.6 under the notebook header “Initiate widgets”. They are then grouped and showed using *iPython.display*. Updates of the values of the widgets are handled by the function *interactive_output*, which then calls a custom update function. When the button for turning on the VNA is pressed, the function *sweep_acquire* from the code of section A.2.1 is called.

For showing the live plot, the Bokeh plotting library was chosen [21]. This choice was made because Bokeh has been used in the past by the intended users in the research group, who favour Bokeh for its responsiveness. This responsiveness is also favourable for the GUI of the VNA, to remain below the $100\ \mathrm{ms}$ delay between the trace data generation and showing it in a plot. The knowledge on how to use Bokeh already being available with the users also means that the GUI is easily extensible, which gives an advantage over most other plotting libraries.

An object for the plot frame is created with the function *plt.figure*, which is seen under the header “Run GUI” in section A.2.6. This object can then be modified with plot settings, and by adding the plots themselves. For generating the plots, the function *line* is used. The values of this plot are then adjusted using the *.data_source.data* attribute. This is done in the *update_plot* function, which is running an infinite loop in a separate thread when the frequency sweep has started. This way, the data of the plots is continuously updated and shown to the user while new measurement data is flowing into the core program via the TCP client.

An example of the Graphical User Interface is shown in figure 7.1, which is a screenshot of part of the Jupyter notebook. The live plot shows that the measurement points for frequencies up to 6.3 GHz have been received, while the higher frequencies are still to be measured. The RF inputs of the Red Pitaya were terminated with a passive load (instead of being fed with an IF signal from the generators), meaning the digital downconversion should return a magnitude close to zero for all frequencies. The phase of any phasor with magnitude close to zero is undefined, therefore the blue phase curve in the figure shows some jumps. The entire system, with the RF generators, splitters and mixers connected could not be tested due to time constraints and issues with triggering both generators simultaneously.

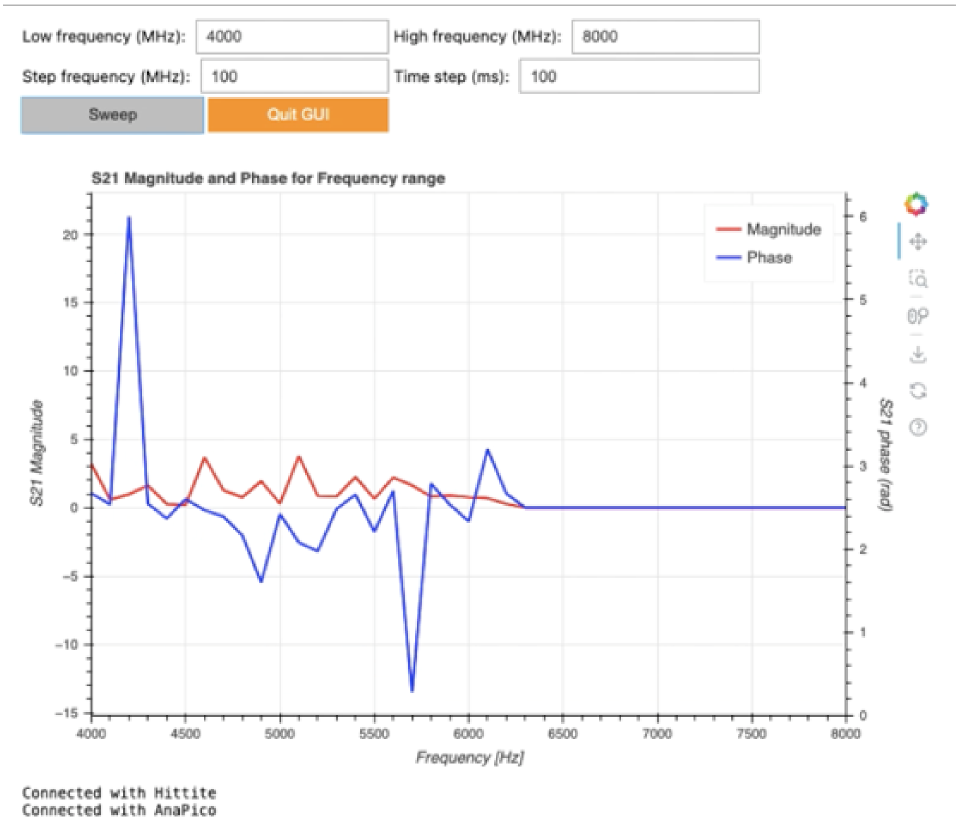

Figure 7.1: Graphical User Interface consisting of iPyWidgets buttons and input fields and Bokeh live plot, showing the magnitude and phase of the S_{21} parameter

8

Conclusion & discussion

8.1. Conclusions about the Python software

Two Python programs have been written by the software team to make it possible for users to interface with the VNA. The main task of the server-side program, which will be running on the processing system inside the Xilinx Zynq 7010 System on Chip, is to fetch data from the programmable logic using DMA and transmit it over the network to the client, relying on the speed and simplicity of the Transmission Control Protocol. The requirements for the server in terms of throughput and overhead have been met, as experiments have shown. The open-source code has been written in a way that is extensible and understandable for the intended users.

The client-side Python program controls the RF generators with SCPI, which makes them perform a frequency sweep with configurable settings. The generators are triggered from outputs of the programmable logic. The client sends to the programmable logic how long the generators have unstable output during switching, which is called the dead time. This way, the programmable logic ensures that data is collected only with all components in steady state, which means the client receives the correct data. The client also sends other measurement parameters like time per point and trigger configuration to the server, which are forwarded to the programmable logic via MMIO registers. The client receives the acquired measurement data in the form of I and Q values, giving complex representations of the electromagnetic waves going into and out of the device under test. With this data, the S_{21} parameter is calculated for every frequency. The client saves the measurement data together with metadata in a systematic format that is known by the intended users. The objectives of implementing a Graphical User Interface and being responsive enough also have been achieved.

8.2. Recommendations

The following considerations can be made to improve and extend the work in the future.

- The VNA can be extended with power sweep functionality. The Python client needs to control the generators to perform triggered power sweeps at a fixed frequency. The metadata saved by the client then has to change from frequency to power. However, the configuration to the programmable logic does not need to be altered, as the currently used parameters (table 4.1) are sufficient. The PL does not need additional knowledge to be able to acquire data for a power sweep, since the calculation of the I and Q values will stay the same. The RF subteam has to ensure the power level at the input of the analog-to-digital converters is constant during the sweep, for example using a programmable attenuator.

- The generators have a specific dead time during which they give unstable output. The current minimum value for the time per measurement point is one millisecond. Bringing this value closer to the dead time could be implemented, provided that the DMA data transfers can have a higher throughput, which can be realised with larger data buffers. The current TCP throughput is high enough for a time per point of 3 microseconds, as given in equation (5.1).
- The system could be modified to get vector signal analyser (VSA) capabilities. This type of device measures an EM spectrum as absolute value, so without sending any signals itself with a (RF) generator, to compare it with. It then shows the absolute amplitude and phase relative to an LO over a certain frequency spectrum. For the software to be ready for this, it would be important to implement IQ data streaming with high enough throughput to show the entire required spectrum within a certain refresh rate, to have a live view of the frequencies of signals coming in.

References

[1] *What is a Vector Network Analyzer and How Does it Work?* [Online]. Available: https://www.tek.com/en/documents/primer/what-vector-network-analyzer-and-how-does-it-work.

[2] L. Zhong, R. Yu, and X. Hong, " Review of carbon-based electromagnetic shielding materials: film, composite, foam, textile," *Textile Research Journal*, vol. 91, p. 004 051 752 096 828, Oct. 2020. DOI: 10.1177/0040517520968282.

[3] F. Caspers, " RF engineering basic concepts: S-parameters," CERN, Tech. Rep., 2013.

[4] *What Can You Do With a VNA?* 2023. [Online]. Available: https://coppermountaintech.com/what-can-you-do-with-a-vna/.

[5] L. Dicarlo, *Introduction to circuit QED*, 2023.

[6] Keysight. "Product page: P9372A Keysight Streamline USB Vector Network Analyzer, 9 GHz." (2024), [Online]. Available: https://www.keysight.com/us/en/product/P9372A/keysight-streamline-usb-vector-network-analyzer-9-ghz.html.

[7] "About NanoVNA." (), [Online]. Available: https://nanovna.com.

[8] "SHFQA+ 8.5 GHz Quantum Analyzer." (), [Online]. Available: https://www.zhinst.com/europe/en/products/shfqa-quantum-analyzer.

[9] "OPX+: Ultra-Fast Quantum Controller." (), [Online]. Available: https://www.quantum-machines.co/products/opx/#.

[10] A. Raza, A. Jabbar, D. A. Sehrai, H. Atiq, and R. Ramzan, "SDR Based VNA for Characterization of RF Sensors and Circuits," in *2021 1st International Conference on Microwave, Antennas & Circuits (ICMAC)*, 2021, pp. 1–4. DOI: 10.1109/ICMAC54080.2021.9678273.

[11] H. Forstén, "Improved homemade VNA," Tech. Rep., 2017. [Online]. Available: https://hforsten.com/improved-homemade-vna.html.

[12] J. Mower and Y. Kuga, "A FPGA-Based Replacement for a Network Analyzer in an Instrumentation-Based 200 GHz Radar," *High Frequency Electronics*, pp. 30–40, Sep. 2013.

[13] Y. Xu, G. Huang, N. Fruitwala, *et al.*, *QubiC 2.0: An Extensible Open-Source Qubit Control System Capable of Mid-Circuit Measurement and Feed-Forward*, 2023. arXiv: 2309.10333 [quant-ph].

[14] *DMA — PYNQ: Python productivity for Zynq (Pynq)*, version 3.0.0. [Online]. Available: https://pynq.readthedocs.io/en/v3.0.0/pynq_libraries/dma.html.

[15] *MMIO — PYNQ: Python productivity for Zynq (Pynq)*, version 3.0.0. [Online]. Available: https://pynq.readthedocs.io/en/v3.0.0/pynq_libraries/mmio.html.

[16] F. Schmidt and G. Steele. "Data Explorer," Steele Lab, Kavli Institute of Nanoscience. (2024), [Online]. Available: https://gitlab.tudelft.nl/steelelab/data-explorer.

[17] *Xarray documentation*, version v2024.06.0. [Online]. Available: https://docs.xarray.dev/en/stable/.

[18] *SCPI Specification*, IVI Foundation, May 1999. [Online]. Available: https://www.ivifoundation.org/specifications/default.html.

[19] *VPP-4.3: The VISA Library*, version 7.2.1, IVI Foundation, Jan. 2024. [Online]. Available: https://www.ivifoundation.org/specifications/default.html.

[20] *Jupyter Widgets Documentation*, version 8.1.3. [Online]. Available: https://ipywidgets.readthedocs.io/en/stable/.

[21] *Bokeh Documentation*, version 3.4.1. [Online]. Available: https://docs.bokeh.org/en/latest/.

[22] *PYNQ Overlays — Python productivity for Zynq (Pynq)*, version 3.0.0. [Online]. Available: https://pynq.readthedocs.io/en/v3.0.0/pynq_overlays.html.

[23] *queue — A synchronized queue class*, 2024. [Online]. Available: https://docs.python.org/3.11/library/queue.html.

[24] *Windfreak Technologies SynthHD Programming Interface*, version v1.0b. [Online]. Available: https://windfreaktech.com/product/microwave-signal-generator-synthhd/.

[25] *Programmer's Manual; Installation, Operation & Maintenance Guide for HMC-T2100 & HMC-T2100B*, version 04.0811, Hittite Microwave Corporation.

[26] *Get Started — pytest documentation*, 2023. [Online]. Available: https://www.pytest.org/en/7.4.x/getting-started.html.

[27] "Transmission Control Protocol," Wikipedia. (2024), [Online]. Available: https://en.wikipedia.org/wiki/Transmission_Control_Protocol.

[28] N. Jennings. "Socket Programming in Python (Guide)," Real Python tutorial team. (2018), [Online]. Available: https://realpython.com/python-sockets/#echo-client-and-server.

[29] *Programmer's Manual V2.24 Signal Source Models*, AnaPico AG. [Online]. Available: https://www.anapico.com/products/frequency-synthesizers/single-output-frequency-synthesizers/apuasyn20-up-to-20-ghz/.

[30] [Online]. Available: https://standards.ieee.org/ieee/488.2/718/.

[31] *Following the SCPI Learning Process and Using the Tool*, Keysight Technologies. [Online]. Available: https://www.keysight.com/us/en/assets/9921-01868/miscellaneous/FollowtheSCPILearningProcessandUsingtheTool.pdf.

[32] *NI-VISA Overview*, National Instruments Corp. [Online]. Available: https://www.ni.com/en/support/documentation/supplemental/06/ni-visa-overview.html.

[33] *PyVISA-py: Pure Python backend for PyVISA*, version 0.7.2. [Online]. Available: https://pyvisa.readthedocs.io/projects/pyvisa-py/en/latest/.

[34] *PyVISA: Control your instruments with Python*, version 1.14.1. [Online]. Available: https://pyvisa.readthedocs.io/en/latest/.

[35] *User manual; Installation, Operation & Maintenance Guide for HMC-T2100 & HMC-T2100B*, version 04.0710, Analog Devices, inc. [Online]. Available: https://www.analog.com/media/en/technical-documentation/user-guides/hmc-t2100_user_manual_125790.pdf.

[36] *User's Manual V3.07 Signal Source Models*, AnaPico AG. [Online]. Available: https://www.anapico.com/products/frequency-synthesizers/single-output-frequency-synthesizers/apuasyn20-up-to-20-ghz/.

[37] *Universal Serial Bus Test and Measurement Class Specification (USBTMC)*, version 1.00, USB Implementers Forum, Inc., 2003. [Online]. Available: https://www.usb.org/document-library/test-measurement-class-specification.

[38] *NI-VISA User Manual*, National Instruments Corp., 2024. [Online]. Available: https://www.ni.com/docs/en-US/bundle/ni-visa/page/user-manual-welcome.html.

[39] *PyUSB – Easy USB access for Python*, 2024. [Online]. Available: https://github.com/pyusb/pyusb/blob/master/README.rst.

[40] *libusb-1.0 API Reference*, 2024. [Online]. Available: https://libusb.sourceforge.io/api-1.0/.

[41] *pySerial 3.0 documentation*, 2015. [Online]. Available: https://pythonhosted.org/pyserial/index.html.

A
Source code

This appendix contains all code that has been written by the software team to define the behaviour of the server-side and client-side Python programs[1]. The open-source repository of the project will be located at `https://gitlab.tudelft.nl/steelelab/bep-steelelab-vna-2024`.

A.1. Server-side program

A.1.1. Data processing module

```python
"""Classes and functions for fetching and storing acquired data from programmable logic (PL)"""

from queue import Full, Queue
from time import sleep

from pynq import MMIO, Overlay, allocate

import helpers
from protocol import PLConfig


class PLInterface(PLConfig):
    """Functions related to the interface between the processing subsystem (PS) and programmable logic (PL)."""

    class DMANotAllowed(Exception):
        """Special exception raised when data transfer via DMA will hang due to PL being in reset"""

    DEBUG = True
    """Whether to print debugging information"""

    def __init__(self):
        self.ol = Overlay(PLInterface.OVERLAY_PATH)
        self.mmios = {key: MMIO(value) for key, value in PLInterface.MMIO_DICT.items()}
        self.dma_channel = self.ol.dma.recvchannel
        self.dma_output_buffer = allocate(shape=(PLConfig.DMA_DATA_SIZE, ), dtype=PLInterface.DMA_DTYPE)
        self._enabled = False
        self.dma_status = 2  # default: after DMA transfer

    @property
    def enable(self):
        """Checks whether the programmable logic is enabled."""
        return self._enabled

    @enable.setter
    def enable(self, value):
```

[1] In some Python files, the term OpenVQA is used, which refers to the product name Open Vector Qubit Analyser.

```python
        """Enables the programmable logic data acquisition.
        Only change this if previous DMA requests are properly closed!
        """
        if not isinstance(value, bool) or (self._enabled and value):
            return
        if not value:
            self.disable()
            return
        mmio_general = self.mmios[PLInterface.MMIO_GENERAL]

        # Read current bits of general MMIO and write reset bit.
        current = mmio_general.read(0)
        mmio_general.write(0, current | PLInterface.PL_RUNNING_BIT)

        # First do a DMA request of 16 words to get rid of misformed packet.
        if PLInterface.DEBUG: helpers.printd("Starting first DMA data request...")
        temp_buffer = allocate(shape=(16, ), dtype=PLInterface.DMA_DTYPE)
        self.dma_channel.transfer(temp_buffer)
        self.dma_channel.wait()
        del temp_buffer
        self._enabled = True
        if PLInterface.DEBUG: helpers.printd("Started programmable logic data acquisition.")

    def disable(self):
        """Puts the programmable logic in reset."""
        if not self._enabled:
            return
        self._enabled = False
        mmio_general = self.mmios[PLInterface.MMIO_GENERAL]

        # Read current bits of general MMIO and write reset bit.
        current = mmio_general.read(0)
        mmio_general.write(0, current & ~PLInterface.PL_RUNNING_BIT)
        if PLInterface.DEBUG: helpers.printd("Stopped programmable logic data acquisition.")

    def get_data(self):
        """Returns processed data retrieved from PL using DMA."""
        return self.preprocess_raw_dma_data(self.raw_dma_data_request())

    def raw_dma_data_request(self):
        """Reads data from a direct memory access channel."""
        if not self.enable:
            raise PLInterface.DMANotAllowed("PL not enabled; cannot transfer data via DMA.")
        try:
            # No timeout available in `wait`; use dma_channel.stop() outside thread to stop.
            self.dma_status = 0
            self.dma_channel.transfer(self.dma_output_buffer)
            self.dma_status = 1
            self.dma_channel.wait()
            self.dma_status = 2
        except RuntimeError as err:
            # This occurs when the programmable logic just started after reset and did not yet configure the DMA channel.
            raise PLInterface.DMANotAllowed from err
        return self.dma_output_buffer

    def preprocess_raw_dma_data(self, buffer):
        """Divides integer I and Q values by sample count. Buffer is an array containing a multiple of three
        elements: I value, Q value, count. The I and Q values are divided by count
        and multiplied by a conversion factor to get the unit of volts.
        """
        volts = [
            (
                helpers.uint64_to_signed_int(
                    int(buffer[i])  # to Python integer(first entry: unsigned 32-bit integer
                    + (int(buffer[i + 1]) << 32)  # adding second unsigned integer shifted left 32 bits)
                ) / buffer[i + 2]  # dividing by third entry (count)
                * PLInterface.RAW_TO_VOLTS  # scaling to units of volts
            ) for i in range(0, len(buffer), 3)  # i = 0, 3, 6, 9
```

```python
        ]
        return volts

    @staticmethod
    def _get_mmio_idx(cmd):
        """Retrieves the index of the MMIO that is used for a given configuration command."""
        mmio_idx = PLInterface.TCP_MMIO_DICT.get(cmd)
        if mmio_idx is None:
            raise KeyError(f"Cannot write using MMIO; command {cmd} does not exist in TCP_MMIO_DICT in the protocol!")
        return mmio_idx

    def write_mmio(self, cmd, value):
        """Finds the MMIO corresponding to the given command to write the value to."""
        idx = PLInterface._get_mmio_idx(cmd)
        mmio = self.mmios[idx]
        mmio.write(offset=0, data=value)

    def read_mmio(self, cmd):
        """Finds the MMIO corresponding to the given command and reads its value."""
        idx = PLInterface._get_mmio_idx(cmd)
        mmio = self.mmios[idx]
        return mmio.read(offset=0)

    def get_mmio_status(self):
        """Returns dictionary with hexadecimal addresses and corresponding current binary contents of all MMIO registers."""
        return {f"0x{m.base_addr:08x}": f"0b{m.read():>032b}" for m in self.mmios.values()}

    def verify_mmio(self):
        """Checks using assert statements that the current MMIO configuration does not cause problems in the PL,
        such as invalid counter values leading to DMA transfers becoming impossible to perform.
        """
        dead_time = self.mmios[PLConfig.MMIO_DEAD_TIME].read()
        trigger_length = self.mmios[PLConfig.MMIO_TRIG].read() & ((1 << 24) - 1)  # lowest 24 bits of register
        tpp = self.mmios[PLConfig.MMIO_TPP].read()
        assert tpp > 0, f"time per point {tpp} should be greater than zero"
        assert dead_time > 0, f"generator dead time {dead_time} should be greater than zero"
        assert tpp > dead_time, f"time per point {tpp} should be longer than generator dead time {dead_time}"
        assert tpp > trigger_length, f"time per point {tpp} should be longer than trigger pulse length {trigger_length}"

    @property
    def dma_status(self):
        """For debugging purposes. Status code 0: when a transfer is about to start; 1: when waiting
        for the data to become available; 2: when a transfer has been completed.
        """
        return self._dma_status

    @dma_status.setter
    def dma_status(self, value):
        if helpers.VERBOSE: helpers.printd(f"[DMA] status {value}.")
        self._dma_status = value


class DataQueue(Queue):
    """First-in first-out structure storing acquired data"""

    MAXSIZE_BITS = 16
    """Maximum size of the queue is 2 ** BITSIZE - 1"""

    QUEUE_TIMEOUT = 50E-3
    """Timeout in seconds for waiting while getting from and putting data into the queue"""

    DEBUG = True
    """Whether to print debugging information"""
```

```python

    def __init__(self):
        super().__init__(2 ** DataQueue.MAXSIZE_BITS - 1)
        self.is_fetching = False
        self.is_waiting = True
        self.fetching_paused = True  # signal to other threads

    @property
    def is_fetching(self):
        """Keeps sending requests to get data via DMA."""
        return self._is_fetching

    @is_fetching.setter
    def is_fetching(self, value):
        if not isinstance(value, bool):
            raise TypeError(f"Cannot set property `is_fetching` with type {type(value)}!")
        if helpers.VERBOSE: helpers.printd(f"[DMA] {'not ' if not value else ''}fetching into queue.")
        self._is_fetching = value

    def flush(self):
        """Removes all items in the queue."""
        with self.mutex:
            self.queue.clear()
        if not self.empty():
            raise RuntimeError(f"Emptying queue failed; size is {self.queue.qsize()} > 0.")
        if DataQueue.DEBUG: helpers.printd("Queue is now empty.")

    def keep_fetching(self, fetch_func, overwrite_when_full=True):
        """Fetch via a provided function and store in the queue,
        as long as `self.is_fetching` and `self.is_waiting` are True.
        By default, overwrites the oldest data when full, else does nothing.
        """
        i = 0  # debug counter

        # Loop to keep fetching from the queue.
        while True:
            # If not waiting nor fetching, return.
            while not self.is_fetching and self.is_waiting:
                sleep(0.0005)
            if not self.is_waiting:
                return  # Keep `fetching_paused` True.
            self.fetching_paused = False

            if DataQueue.DEBUG and i % 1000 == 0:
                self.check()

            # Execute fetch function.
            try:
                new = fetch_func()
            except PLInterface.DMANotAllowed:
                if helpers.VERBOSE: helpers.printd("[DMA] thread tried to fetch while DMA channel not ready.")
            else:
                # Store result in the queue.
                i += 1
                try:
                    self.put(new, timeout=DataQueue.QUEUE_TIMEOUT)
                except Full:
                    if overwrite_when_full:
                        self.get_nowait()
                        self.put(new)
                        if DataQueue.DEBUG: helpers.printd("Queue is full; overwritten oldest data item!")

            # If still fetching, continue immediately.
            if self.is_fetching:
                continue

            # Wait for new signal to start fetching.
            helpers.printd("Fetching data via DMA to queue paused.")
```

```
            self.fetching_paused = True  # signal for other threads

    def check(self):
        """Gives debugging warnings if queue is almost empty or almost full."""
        if not DataQueue.DEBUG:
            return
        size, maxsize = self.qsize(), self.maxsize
        if 0 < size < 0.03 * maxsize:
            helpers.printd(f"Data␣queue␣is␣almost␣empty:␣{size}␣of␣{maxsize}.")
        if 0.95 * maxsize < size < maxsize:
            helpers.printd(f"Data␣queue␣is␣almost␣full:␣{size}␣of␣{maxsize}.")
```

A.1.2. TCP server module

```
"""Simple TCP server that runs on the processing system (PS) to receive configuration and send acquired data"""

from queue import Empty
import socket
from threading import Thread
from time import sleep

from data_processing import DataQueue, PLInterface
import helpers
from protocol import TCPCommandProtocol


class TCPDataServer(TCPCommandProtocol):
    """Simple host:port socket server based on https://realpython.com/python-sockets"""

    class ServerStop(Exception):
        """Graceful stop for the TCP server"""

    BUFSIZE = 16
    """Receiving buffer size"""

    DEBUG = True
    """Whether to print debugging information"""

    USE_QUEUE = True
    """Whether to store the acquired and preprocessed data in a Python queue"""

    TCP_CONFIG_CMDS = {
        TCPCommandProtocol.DEAD_TIME, TCPCommandProtocol.TPP, TCPCommandProtocol.TRIG_LEN, TCPCommandProtocol.TRIG_0_CONF,
        TCPCommandProtocol.TRIG_1_CONF
    }
    """All configuration commands used in client-server communication"""

    TCP_REQUEST_CMDS = {TCPCommandProtocol.DATA, TCPCommandProtocol.CPU_TEMP, TCPCommandProtocol.QUEUE_SIZE}
    """All commands a client can use to request data from the TCP server"""

    def __init__(self, host, port):
        self.host, self.port = host, port
        if TCPDataServer.DEBUG: helpers.printd(f"Configuring␣programmable␣logic␣{PLInterface.OVERLAY_PATH}...")
        self.pl_interface = PLInterface()

        # Create a queue as data buffer.
        self.queue = DataQueue()
        if TCPDataServer.USE_QUEUE:
            if TCPDataServer.DEBUG: helpers.printd("Starting␣data␣fetch␣thread...")
            self.fetch_thread = Thread(
                target=self.queue.keep_fetching, args=(self.pl_interface.get_data, ), name="vna_fetch_dma"
            )
            self.fetch_thread.start()

    def get_data(self):
        """Reads the I and Q data (points) from the data queue,
```

```
        groups it into larger packets and converts to bytes.
        """
        if not self.pl_interface.enable:
            raise RuntimeError("Cannot get new data as PL not enabled.")
        data_packet = []

        # Loop for creating larger packets from individual data/point requests.
        while True:

            # This gets data from the queue.
            if TCPDataServer.USE_QUEUE:
                try:
                    data = self.queue.get(timeout=DataQueue.QUEUE_TIMEOUT)
                except Empty:
                    # Behaviour when timeout occured: send immediately if at least one packet retrieved from queue.
                    if len(data_packet) > 0:
                        if helpers.VERBOSE:
                            helpers.printd(f"[TCP] queue timeout occured; sending {len(data_packet) // 4} point(s) now.")
                        break
                    if helpers.VERBOSE:  # Else, keep waiting for data.
                        helpers.printd("[TCP] queue timeout occured; still got no data to send.")
                    continue

            # This gets data directly from memory (DMA between PL and memory).
            else:
                data = self.pl_interface.get_data()

            # Send the data if the maximum amount of points per packet has been reached.
            data_packet.extend(data)
            if len(data_packet) // len(data) == TCPDataServer.POINTS_PER_PACKET:
                break

        # Convert 64-bit floats to bytes.
        if len(data_packet) == 0:
            raise RuntimeError("No data could be acquired.")
        return helpers.floats64_to_bytes(data_packet)

    def change_config(self, config):
        """Changes hardware configuration for programmable logic with provided dictionary."""
        for cmd, value in config.items():
            # Determine scalar to multiply value with.
            scalar = PLInterface.MMIO_VALUE_SCALING_DICT.get(cmd, 1)
            try:
                scalar_int = int(scalar)
            except ValueError as err:
                raise TypeError(f"Cannot convert {scalar} from MMIO_VALUE_SCALING_DICT in protocol to integer!") from err
            value *= scalar_int

            # Special case: trigger configurations: keep certain contents of register.
            if cmd in {TCPCommandProtocol.TRIG_0_CONF, TCPCommandProtocol.TRIG_1_CONF}:
                current = self.pl_interface.read_mmio(cmd)
                # Discard (set to zero) the current bits; then OR with the scaled value.
                value_to_write = (current & ~(0b1111 * scalar_int)) | value
            else:  # Overwrite complete 32-bit register.
                value_to_write = value

            # Write new value.
            self.pl_interface.write_mmio(cmd, value_to_write)
            if TCPDataServer.DEBUG:
                helpers.printd(f"Config {cmd} changed to {value // scalar}.")
            if helpers.VERBOSE:
                helpers.printd(f"[TCP] written config value was {value_to_write:032b}.")

    def control_on_off(self, data):
        """Turns on or off the programmable logic and stops or starts fetching data into the queue via DMA.
        Argument `data` should be '0' or '1'.
```

```
        """
        if data not in {"0", "1"}:
            raise ValueError(f"Unknown value: {data} is not '0' or '1'.")
        enable_pl = data == "1"

        # Check that when enabling, current configuration does not cause infinite (especially DMA).
        if enable_pl:
            self.pl_interface.verify_mmio()

        # Pause DMA transfers, empty the queue, enable programmable logic and queue fetching. The order is important!
        self.pause_dma()
        if enable_pl:
            self.queue.flush()
        try:
            if helpers.VERBOSE: helpers.printd("Current MMIO contents:", self.pl_interface.get_mmio_status())
            self.pl_interface.enable = enable_pl
            self.queue.is_fetching = enable_pl
        except (RuntimeError, PLInterface.DMANotAllowed) as err:
            raise PLInterface.DMANotAllowed(
                f"PL enabling failed; DMA not gracefully stopped? DMA status = {self.pl_interface.dma_status}"
            ) from err

        return TCPDataServer.RESPONSE_OK if enable_pl == self.pl_interface.enable else TCPDataServer.RESPONSE_ERR

    def pause_dma(self):
        """Pauses DMA thread after current transfer has been completed.
        Waits for thread to return special `fetching_paused` signal.
        """
        if self.queue.is_fetching or not self.queue.fetching_paused:
            self.queue.is_fetching = False  # Send pause fetching signal.
            while self.pl_interface.dma_status != 2 or not self.queue.fetching_paused:
                pass  # Wait until thread replies with fetching paused signal and DMA status stays at 2.

    def determine_response(self, data):
        """Logic for the server's response based on received decoded data"""
        if data == TCPDataServer.DATA:
            return self.get_data()
        if data[0] == TCPDataServer.RUN_PL:  # control reset
            return self.control_on_off(data[1:])
        if data == TCPDataServer.QUEUE_SIZE:  # server DMA buffer queue size
            return self.queue.qsize().to_bytes(length=DataQueue.MAXSIZE_BITS // 8, byteorder="big")
        if data == TCPDataServer.CPU_TEMP:  # server cpu temperature
            return helpers.floats64_to_bytes((helpers.cpu_temp(), ))
        if data == TCPDataServer.STOP_SERVER:
            self.stop()
        if len(data) > 1:
            if data[0] not in TCPDataServer.TCP_CONFIG_CMDS:
                raise ValueError(f"Unknown config {data[0]}.")
            self.change_config({data[0]: int(data[1:])})
            return TCPDataServer.RESPONSE_OK
        return TCPDataServer.RESPONSE_ERR

    def serve_one_client(self):
        """Sends acquired data to one client."""
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as self.sock:
            try:
                self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
                self.sock.bind((self.host, self.port))
                self.sock.listen(1)
            except OSError as err:
                helpers.printd(f"Cannot start server: {str(err)}")
                return self.stop()
            if TCPDataServer.DEBUG:
                helpers.printd(f"Started TCP data server on {self.host}:{self.port}.")
```

```python

            # Wait until client accepts connection.
            while True:
                server_waits_for_client = True
                self.pause_dma()  # First close the currently running DMA transfer.
                self.pl_interface.enable = False  # Then disable data acquisition.
                try:
                    if helpers.VERBOSE: helpers.printd("[TCP] waiting for client.")
                    conn, addr = self.sock.accept()  # blocking
                    client = f"{addr[0]}:{addr[1]}"
                except KeyboardInterrupt:
                    if TCPDataServer.DEBUG:
                        helpers.printd("Keyboard interrupt; stopping TCP server...")
                    return self.stop(quiet=True)
                with conn:
                    if TCPDataServer.DEBUG: helpers.printd(f"{client} connected to the TCP server.")

                    # Loop until client disconnects.
                    received_data = b""
                    while True:
                        try:
                            received_data = conn.recv(TCPDataServer.BUFSIZE)
                        except (ConnectionResetError, BrokenPipeError) as err:
                            if TCPDataServer.DEBUG:
                                helpers.printd(f"{client} caused exception: {err}")
                                continue
                        if not received_data:
                            if server_waits_for_client:
                                sleep(0.1)
                                continue
                            if TCPDataServer.DEBUG: helpers.printd(f"{client} disconnected.")
                            break

                        # Start processing commands when client sends them.
                        server_waits_for_client = False
                        try:
                            response = self.determine_response(received_data.decode())
                        except Exception as err:
                            if isinstance(err, TCPDataServer.ServerStop):
                                return
                            response = TCPDataServer.RESPONSE_ERR
                            if TCPDataServer.DEBUG:
                                helpers.printd(
                                    f"Exception occured when processing command {received_data.decode()}: "
                                    f"{type(err).__name__}: {' '.join(err.args)}"
                                )

                        # Respond to the client.
                        try:
                            conn.sendall(response)
                        except (ConnectionResetError, BrokenPipeError):
                            if TCPDataServer.DEBUG:
                                helpers.printd(f"{client} reset the connection.")
                        except TCPDataServer.ServerStop:
                            return  # End the loop if server stop requested.

    def stop(self, quiet=False):
        """Closes the TCP server, stops the threads that were started and raises the ServerStop exception."""
        if not hasattr(self, "sock"):
            return
        if TCPDataServer.DEBUG: helpers.printd("Stopping TCP server...")
        self.sock.close()
        if TCPDataServer.USE_QUEUE:
            if TCPDataServer.DEBUG: helpers.printd("Stopping data fetch thread...")
            self.queue.is_waiting = False  # No longer accept new client connections.
            self.pause_dma()  # No longer fetch via DMA.
            self.pl_interface.dma_channel.stop()  # Stop DMA channel if currently waiting.
            self.fetch_thread.join(timeout=5)
```

```
        if not quiet:
            raise TCPDataServer.ServerStop
```

A.1.3. Communication and PS/PL protocol module

```
"""Definitions for proper communication and programmable logic (PL) configuration parameters"""

from os.path import join, dirname

from numpy import uint32


class TCPCommandProtocol:
    """Defines the commands used for communication between Python TCP client
    and Python TCP server.
    """

    TCP_PORT = 2024
    """Port on which the Python TCP server listens for clients"""

    DEAD_TIME = "g"
    """Generator dead time in microseconds"""

    TPP = "p"
    """Time per point in microseconds for averaging inside the PL"""

    RUN_PL = "r"
    """Enable data acquisition in programmable logic"""

    TRIG_LEN = "t"
    """Trigger pulse length in microseconds"""

    TRIG_0_CONF = "c"
    """Trigger output 0 configuration; expecting an integer below 8"""

    TRIG_1_CONF = "o"
    """Trigger output 1 configuration; expecting an integer below 8"""

    DATA = "d"
    """Request IQ data in volts"""

    CPU_TEMP = "T"
    """Request server cpu temperature in degrees Celsius"""

    QUEUE_SIZE = "q"
    """Request server data queue size"""

    RESPONSE_OK = b"*"
    """Server understood client's command"""

    RESPONSE_ERR = b"?"
    """Server did not understand client's command or an internal error occured"""

    STOP_SERVER = "!"
    """Command to stop the TCP server remotely"""

    POINTS_PER_PACKET = 45
    """Number of IQ measurements (points) per TCP transfer; for optimal throughput and response time / interactivity"""


class PLConfig:
    """Defines programmable logic configuration parameters"""

    OVERLAY_PATH = join(dirname(dirname(dirname(__file__))), "pl", "vna_v1_7.bit")
    """Path to .bit file to be used as overlay on programmable logic; .hwh file should also be in this directory"""

    MMIO_DEAD_TIME = 0
    """MMIO used for configuring generator dead time"""
```

```python

    MMIO_TPP = 1
    """MMIO used for configuring time per point"""

    MMIO_TRIG = 2
    """MMIO used for configuring trigger output"""

    MMIO_GENERAL = 3
    """MMIO used for general programmable logic configuration"""

    MMIO_DICT = {MMIO_TRIG: 0x41200000, MMIO_GENERAL: 0x41200008, MMIO_DEAD_TIME: 0x42000000,
         MMIO_TPP: 0x42000008}
    """All used memory-mapped input and output interfaces"""

    TCP_MMIO_DICT = {
        TCPCommandProtocol.TPP: MMIO_TPP,
        TCPCommandProtocol.DEAD_TIME: MMIO_DEAD_TIME,
        TCPCommandProtocol.RUN_PL: MMIO_GENERAL,
        TCPCommandProtocol.TRIG_LEN: MMIO_TRIG,
        TCPCommandProtocol.TRIG_0_CONF: MMIO_TRIG,
        TCPCommandProtocol.TRIG_1_CONF: MMIO_TRIG
    }
    """Translation dictionary between TCP commands and memory-mapped programmable logic
        interfaces"""

    MMIO_VALUE_SCALING_DICT = {
        TCPCommandProtocol.TPP: 125,  # clock frequency 125 MHz
        TCPCommandProtocol.DEAD_TIME: 125,  # clock frequency 125 MHz
        TCPCommandProtocol.TRIG_LEN: 125,  # clock frequency 125 MHz
        TCPCommandProtocol.TRIG_0_CONF: 16777216,  # equivalent to x left shift 24
        TCPCommandProtocol.TRIG_1_CONF: 268435456  # equivalent to x left shift 28
    }
    """Scaling of values before writing to programmable logic memory"""

    RAW_TO_VOLTS = 2 ** -25
    """Conversion of raw DMA output to volts"""

    DMA_DATA_SIZE = 12
    """Length (in nrs. of DMA_DTYPE) of data packet received via DMA"""

    DMA_DTYPE = uint32
    """Data type coming from DMA"""

    PL_RUNNING_BIT = 0x1
    """Active-high bit to set programmable logic running"""
```

A.1.4. Helper module

```python
"""Helper functions related to debugging the server"""

from datetime import datetime
import os
from struct import pack
from subprocess import run

VERBOSE = False
"""Whether to spam your console with messages"""


def printd(*args, **kwargs):
    """Prints date and time in front of message."""
    out = datetime.now().strftime("%Y-%m-%d %H:%M:%S.%f")
    if "flush" not in kwargs:
        print(out, *args, flush=True, **kwargs)
    else:
        print(out, *args, **kwargs)


def cpu_temp():
    """Returns cpu temperature of PYNQ server."""
    path = os.path.join(os.path.dirname(os.path.realpath(__file__)), "sh", "cpu_temp.sh")
```

```python
    if not os.path.isfile(path):
        raise FileNotFoundError(f"Script␣'{path}'␣does␣not␣exist.")
    process = run(f"{path}␣-F", shell=True, capture_output=True, check=False)
    return float(process.stdout.decode())


def floats64_to_bytes(values):
    """Converts iterable of 64-bit Python floats to bytes object. Source:
    https://stackoverflow.com/questions/9940859/fastest-way-to-pack-a-list-of-floats-into-bytes-in-python.
    """
    return pack(f"{len(values)}d", *values)


def uint64_to_signed_int(unsigned):
    """Converts 64-bit unsigned integer to signed integer. By Bit Twiddling Hacks; see
    https://stackoverflow.com/questions/1375897/how-to-get-the-signed-integer-value-of-a-long-in-python.
    """
    unsigned &= (1 << 64) - 1  # Keep only the lowest 64 bits.
    return (unsigned ^ 0x8000000000000000) - 0x8000000000000000  # Swap and shift down.
```

A.1.5. Main server script

```python
#!/usr/local/share/pynq-venv/bin/python3
"""Main server script"""

from helpers import printd
from sys import argv

if len(argv) == 0:
    MOCK_PYNQ = False
elif argv[1] == "-M":
    MOCK_PYNQ = True
else:
    raise ValueError(f"Program␣argument␣{argv[1]}␣not␣understood.␣Options␣are:\n\t-M\tmock␣'pynq'␣library")

if MOCK_PYNQ:  # mocking the pynq library
    printd("Mocking␣'pynq'␣library...")
    from os import getcwd
    from sys import path
    path.insert(0, getcwd())
    from tests.server import mocked_pynq
    mocked_pynq.mock_pynq_module(mocked_pynq)

from tcp_server import TCPDataServer


def main():
    printd("Started␣main␣server␣script.")
    tds = TCPDataServer(host="", port=2024)
    tds.serve_one_client()
    printd("Stopped␣main␣server␣script.")


if __name__ == "__main__":
    main()
```

A.1.6. Mocked PYNQ module

```python
"""Part of a mocked version of the pynq libary for testing on systems that do not have it installed"""

import os
import sys
from types import ModuleType
from typing import Any

from numpy import array, empty, ndarray, uint32, zeros

```

```python
from project.server import helpers, protocol


def mock_pynq_module(mocked_module: ModuleType) -> None:
    # Add directory to path to be able to find 'helpers.py'.
    sys.path.insert(0, os.path.dirname(helpers.__file__))

    # Reassign pynq module to the given mocked module.
    sys.modules["pynq"] = mocked_module


def allocate(shape: Any, dtype: str = "u4", **kwargs) -> Any:
    """Mocked version of pynq.allocate."""
    helpers.printd(f"[TEST]␣MockedPynq:␣allocated␣array␣of␣shape␣{shape}.")
    ALLOCATED_BUFFER = zeros(shape, dtype)
    return ALLOCATED_BUFFER


ALLOCATED_BUFFER = empty(protocol.PLConfig.DMA_DATA_SIZE, dtype=protocol.PLConfig.DMA_DTYPE)
"""Returned upon calling `pynq.allocate`"""


class Overlay:
    """Mocked version of pynq.Overlay"""

    PL_ENABLED = False
    """Programmable login in reset by default"""

    def __init__(self, bitfile_name, *args) -> None:
        helpers.printd(f"[TEST]␣MockedPynq:␣loaded␣overlay␣{bitfile_name};␣PL␣{'not␣'␣if␣Overlay.PL_ENABLED␣else␣''}enabled.")

    class dma:
        """Mocked dma class"""

        class recvchannel:
            """Mocked recvchannel class"""

            BUFFER = array((
                4289555807, 4294967295, 14383, 1139606, 0, 14383, 4291721347, 4294967295, 14383, 4292855430, 4294967295, 14383
            ),
                          dtype=protocol.PLConfig.DMA_DTYPE)
            """Example buffer from programmable logic"""

            def transfer(buffer: ndarray[uint32]) -> None:
                """Simulates that the PL writes data into the allocated buffer."""
                new = Overlay.dma.recvchannel.BUFFER
                if new.shape == buffer.shape:
                    buffer += new
                helpers.printd("[TEST]␣MockedPynq:␣started␣DMA␣transfer.")

            @staticmethod
            def wait() -> None:
                """When testing, wait returns immediately unless an error occured."""
                if not Overlay.PL_ENABLED:
                    raise RuntimeError("[TEST]␣MockedPynq:␣DMA␣transfer␣wait␣will␣hang␣since␣PL␣still␣in␣reset.")

            @staticmethod
            def stop() -> None:
                """Stops the current DMA transfer"""
                helpers.printd("[TEST]␣MockedPynq:␣stopped␣DMA␣transfer.")


class MMIO:
    """Mocked version of pynq.MMIO"""

    def __init__(self, base_addr: int, length: int = 4, device: None = None, **kwargs):
        self.base_addr = base_addr
        self.content = 0x0
```

```python
        helpers.printd(f"[TEST]␣MockedPynq:␣MMIO␣object␣initialised␣at␣address␣0x{base_addr:8x}.")

    def write(self, offset: int, data: int | bytes):
        helpers.printd(f"[TEST]␣MockedPynq:␣writing␣{data}␣to␣MMIO␣address␣0x{self.base_addr:8x}.")
        if isinstance(data, bytes):
            data = int.from_bytes(data)
        self.content = data
        if self.base_addr == protocol.PLConfig.MMIO_DICT[protocol.PLConfig.MMIO_GENERAL] and data == 1:
            Overlay.PL_ENABLED = True

    def read(self, offset: int = 0, length: int = 4, word_order="little") -> int:
        helpers.printd(f"[TEST]␣MockedPynq:␣reading␣value␣from␣MMIO␣address␣0x{self.base_addr:8x}.")
        return self.content
```

A.1.7. Tests for data processing module

```python
"""Tests data processing classes and functions, also on systems that do not have `pynq` installed"""

from threading import Thread
from time import sleep

from pytest import fail, raises

from tests.server import mocked_pynq

# Apply the mocked pynq module before importing the classes to be tested.
mocked_pynq.mock_pynq_module(mocked_pynq)
from project.server.data_processing import DataQueue, PLInterface


def test_dma_hang() -> None:
    # Create PL interface and do not enable acquisition but try a DMA request.
    pl_i = PLInterface()
    assert not pl_i.enable, "PL␣incorrectly␣enabled␣by␣default"
    with raises(PLInterface.DMANotAllowed):
        pl_i.get_data()
        fail("PL␣not␣enabled;␣wait␣for␣DMA␣transfer␣should␣hang.")


def test_dma_get_data() -> None:
    # Create PL interface and start mocked acquisition.
    pl_i = PLInterface()
    assert pl_i.dma_status == 2, "DMA␣status␣should␣be␣2␣since␣no␣transfer␣running"
    pl_i.enable = True
    assert pl_i.enable, "PL␣enable␣failed"

    # Request data via Direct Memory Access.
    raw_data = pl_i.raw_dma_data_request()
    assert len(raw_data) == pl_i.DMA_DATA_SIZE, "raw␣data␣length␣incorrect"
    data = pl_i.preprocess_raw_dma_data(raw_data)
    for el in data:
        assert isinstance(el, float), "data␣element␣should␣be␣float"
    assert pl_i.dma_status == 2, "DMA␣status␣should␣be␣2␣again␣after␣transfer␣finished"


def test_mmio() -> None:
    # Test writing and reading memory-mapped input/output registers.
    pl_i = PLInterface()
    test_data = 0b10001001
    for cmd in PLInterface.TCP_MMIO_DICT:
        idx = PLInterface._get_mmio_idx(cmd)
        mmio_register = pl_i.mmios[idx]
        mmio_register.write(offset=0, data=test_data)
    assert list(pl_i.get_mmio_status().values()
                ) == ([f"0b{test_data:032b}"] * 4), "written␣content␣not␣available␣to␣read␣in␣MMIO␣register(s)"
```

```python


def test_queue() -> None:
    # Create small data queue.
    DataQueue.MAXSIZE_BITS = 3
    dq = DataQueue()
    assert dq.fetching_paused, "new data queue should be paused"

    # Fetch test data in thread and read it.
    test_data_func = lambda: (-1., -.5, .5, 1)
    thr = Thread(target=dq.keep_fetching, args=(test_data_func, ))
    thr.start()
    dq.is_fetching = True
    assert dq.get(timeout=15) == test_data_func(), "queue is not filled with test data"

    # Send signal to thread and wait for response pause signal.
    dq.is_fetching = False
    for _ in range(1500):
        sleep(0.01)
        if dq.fetching_paused:
            break
    else:
        assert dq.fetching_paused, "pause signal still not received 15 seconds after `is_fetching` set to False"

    # Send stop signal and check if thread ended.
    dq.is_waiting = False
    thr.join(timeout=15)
    assert not thr.is_alive(), "thread should exit after setting attribute `is_waiting` to False"
```

A.1.8. Tests for TCP server module

```python
"""Tests for the TCP server"""

from threading import Thread
from types import TracebackType
from typing import Type

# Apply the mocked pynq module before importing the classes to be tested.
from tests.server import mocked_pynq
mocked_pynq.mock_pynq_module(mocked_pynq)

from project.client.connection.tcp_client import TCPClient
from project.server.tcp_server import TCPDataServer


class TCPClient(TCPClient):
    """Modification of TCPClient class for testing purposes"""

    def __exit__(
        self, exc_type: Type[BaseException] | None, exc_val: BaseException | None, exc_tb: TracebackType | None
    ) -> None:
        """Stops server when exiting a `with` block."""
        try:
            self._stop_server(True)
        except ConnectionAbortedError:
            pass  # socket already closed
        super().__exit__(exc_type, exc_val, exc_tb)


def test_tcp_commands() -> None:
    dead_time = 100  # microseconds
    tpp = 1000  # microseconds
    trig_length = 10  # microseconds
    tds = TCPDataServer(host="localhost", port=2024)
    thr = Thread(target=tds.serve_one_client, name="test_tcp_data_server")

    # Start local test server and connect with client.
    thr.start()
```

```python
    with TCPClient(host="localhost", port=2024) as tc:
        # Send config parameters.
        tc.send_dead_time(dead_time)
        tc.send_tpp(tpp)
        tc.send_trigger_length(trig_length)
        trig_length_clock_cyles = trig_length * tds.pl_interface.MMIO_VALUE_SCALING_DICT[tds.TRIG_LEN]
        tc.send_trigger_config(trig_nr=0, positive=True, sweep=True, step=False)

        # Check that trigger config arrived in MMIO register.
        bits = tds.pl_interface.read_mmio(tds.TRIG_0_CONF)
        assert (  # binary trig conf OR trig length in clock cycles
            bits == (0b0010 << 24) | trig_length_clock_cyles
        ), "trigger config not correctly written to MMIO register"
        tc.send_trigger_config(trig_nr=0, positive=False, sweep=False, step=True)
        bits = tds.pl_interface.read_mmio(tds.TRIG_0_CONF)
        assert (  # binary trig conf OR trig length in clock cycles
            bits == (0b0101 << 24) | trig_length_clock_cyles
        ), "trigger config not correctly updated in MMIO register"

        # Check that dead time and time per point arrived in MMIO registers.
        bits = tds.pl_interface.read_mmio(tds.DEAD_TIME)
        assert (  # binary clock cycles for dead time = time * clock frequency
            bits == dead_time * tds.pl_interface.MMIO_VALUE_SCALING_DICT[tds.DEAD_TIME]
        ), "dead time not correctly written to MMIO register"
        bits = tds.pl_interface.read_mmio(tds.TPP)
        assert (  # binary clock cycles per point == time * clock frequency
            bits == tpp * tds.pl_interface.MMIO_VALUE_SCALING_DICT[tds.TPP]
        ), "time per point not correctly written to MMIO register"

        # Test queue size.
        qs = tc.get_queue_size()
    assert isinstance(qs, int), "queue size request did not return integer"
    assert qs == 0, "queue size is not zero before start data acquisition (invalid data in queue)"
    thr.join(timeout=15)
    assert not thr.is_alive(), "server thread did not finish in 15 seconds after receiving stop command"


def test_tcp_request_data() -> None:
    tds = TCPDataServer(host="localhost", port=2024)
    thr = Thread(target=tds.serve_one_client, name="test_tcp_data_server")

    # Start local test server and connect with client.
    thr.start()
    with TCPClient(host="localhost", port=2024) as tc:
        tc.send_tpp(2)  # minimal settings for no error
        tc.send_dead_time(1)  # dead_time < tpp < 0
        assert (  # try data request
            tc.send_receive(TCPClient.DATA) == TCPClient.RESPONSE_ERR
        ), "acquisition not started should return error response"

        # Start acquisition and request data again.
        tc.start_acquisition()
        data = tc.request_data()
    assert len(data) > 0, "data should not be empty"
    assert len(data) <= TCPDataServer.POINTS_PER_PACKET * 4, "data should not be longer than 4 * points per packetwt"
    assert isinstance(data[0], float), "data element should be float"
    thr.join(timeout=15)
    assert not thr.is_alive(), "server thread did not finish in 15 seconds after receiving stop command"
```

A.2. Client-side program

A.2.1. Application programming interface

```python
"""Application Programming Interface for the OpenVQA project"""

from collections.abc import Callable
```

```python
from datetime import datetime
import os
from queue import Queue
from time import strftime

import numpy as np
import pandas as pd
import xarray as xr

from project.client.application.dexplore.data_folder import DataFolder
from project.client.connection.tcp_client import TCPClient
from project.client.generator.base_controller import BaseSCPIGeneratorController


class OpenVQA:
    """Core functions of the Open Vector Qubit Analyser"""

    IP = "vna11"
    """Red Pitaya's IP address or hostname"""

    DEADTIME = 100
    """Generator dead time in microseconds (consider setting this constant in a generator controller class)"""

    TRIGGER_PULSE_LENGTH = 10
    """Trigger pulse length in microseconds (consider setting this constant in a generator controller class)"""

    IF = 7.8125
    """Intermediate frequency in megahertz"""

    SAVE_PATH: str = os.path.join("project", "client", "application", "data")
    """Where to save the acquired (meta)data (.h5 files and script)"""

    PATH_TO_NOTEBOOK: str | None = None
    """Set this inside a Jupyter notebook to also save this."""

    def __init__(
        self, generator_a: BaseSCPIGeneratorController | None, generator_b: BaseSCPIGeneratorController | None
    ) -> None:
        self.generator_a = generator_a
        self.generator_b = generator_b

    def sweep_acquire_2_generators(
        self, trigger_per_step: bool, freq_low: float, freq_high: float, freqstep: float, timestep: int
    ) -> None:
        """Creates the sweeps on the generator, communicates with the server, and saves the data.

        Args:
            trigger_per_step (bool): should only be set to True for generators that support this functionality
            freq_low (float): lower frequency bound for sweep
            freq_high (float): higher frequency bound for sweep
            freqstep (float): frequency step size in the sweep
            timestep (float): time to remain at each frequency step
        """
        if timestep < OpenVQA.DEADTIME:
            raise ValueError(f"Timestep {timestep} should not be smaller than dead time {OpenVQA.DEADTIME}.")
        if timestep < OpenVQA.TRIGGER_PULSE_LENGTH:
            raise ValueError(
                f"Timestep {timestep} should not be smaller than trigger pulse length {OpenVQA.TRIGGER_PULSE_LENGTH}."
            )
        if self.generator_a is None or self.generator_b is None:
            raise TypeError("Both generator A and B have to be not None.")

        with self.generator_a:  # connect with RF generator a
```

```python
            print(f"Connected␣with␣{self.generator_a.name()}.")
            with self.generator_b:  # connect with RF generator b
                print(f"Connected␣with␣{self.generator_b.name()}.")
                with TCPClient(host=OpenVQA.IP, port=TCPClient.TCP_PORT) as self.tcp:  # connect with the Red Pitaya via tcp
                    print(f"Connected␣with␣{OpenVQA.IP}:{TCPClient.TCP_PORT}␣via␣TCP.")
                    num_frequencies = int((freq_high - freq_low) / freqstep + 1)

                    #Prepare all settings of the RF generators
                    self.generator_a.hardware_freq_sweep(freq_low, freq_high, freqstep, timestep, power=13)  #for through-DuT
                    self.generator_b.hardware_freq_sweep(
                        freq_low + OpenVQA.IF, freq_high + OpenVQA.IF, freqstep, timestep, power=13
                    )  #for LO

                    self.queue = Queue(maxsize=num_frequencies)

                    start_time = datetime.now().strftime("%Y-%m-%e␣%H:%M:%S.%f")  #Time kept for later referencing

                    #send configuration to PL
                    self.tcp.send_tpp(timestep)  #time per point in us
                    self.tcp.send_dead_time(OpenVQA.DEADTIME)  #settling time in us
                    self.tcp.send_trigger_length(OpenVQA.TRIGGER_PULSE_LENGTH)  #trigger time length in us
                    self.tcp.send_trigger_config(trig_nr=0, positive=True, sweep=True, step=trigger_per_step)
                    self.tcp.send_trigger_config(trig_nr=1, positive=True, sweep=True, step=trigger_per_step)

                    self.generator_a.perform_sweep()  #start sweep on both generators
                    self.generator_b.perform_sweep()

                    self.tcp.start_acquisition()  #start data acquisition on PL

                    data = np.zeros((num_frequencies, 4))

                    # Request data via TCP and puts in queue until all data has been collected.
                    OpenVQA.receive_data(num_frequencies, self.tcp.request_data, self.queue, data)
                    self.tcp.stop_acquisition()

                    # Save the generator settings and server temperature as metadata.
                    setting_a = self.generator_a.read_status()
                    setting_b = self.generator_b.read_status()
                    temperature = self.tcp.get_server_cpu_temp()

        stop_time = datetime.now().strftime("%Y-%m-%d␣%H:%M:%S.%f")

        print("Time␣between␣sending␣configs␣and␣having␣all␣data:\nStart␣time:␣", start_time, "\nStop␣time:␣", stop_time)

        frequency_axis = np.linspace(freq_low, freq_high, int(num_frequencies))

        # Construct full data array: [:,0]: frequency, [:,1]: real s21,[:,2]: imaginairy s21, [:,3]: magnitude s21,
        # [:,4]: phase s21, 5: re DuT, 6: im DuT, 7: mag DuT, 8: ph DuT, 9: re Ref, 10: im Ref, 11: mag Ref, 12: ph Ref
        full_data = OpenVQA.construct_output_data(frequency_axis, data)
        OpenVQA.save_data(
            data=full_data,
            gen1_setting=setting_a,
            gen2_setting=setting_b,
            temperature=temperature,
            save_path=OpenVQA.SAVE_PATH,
            path_to_notebook=OpenVQA.PATH_TO_NOTEBOOK
        )

    def __enter__(self) -> "OpenVQA":
```

```python
        """Enters the `with` block; returns itself to the variable after the `as` keyword."""
        return self

    def __exit__(self, *args, **kwargs):
        """Leaves the `with` block."""
        print("OpenVQA exited.")

    @staticmethod
    def receive_data(num_measurements: int, request_data: Callable, queue: Queue, data: np.ndarray) -> None:
        """Asks and waits for data from tcp, then puts it in queue and repeats.
        Args:
            num_measurements (int): number of measurement points
            request_data: (callable): function that waits for requests and waits for data from tcp
            queue (queue.Queue): queue object containing the data received via tcp (for GUI thread)
            data: numpy array also containing the data received via tcp
        """
        total_nr_points = 0
        while num_measurements > 0:
            try:
                new = request_data()  #waits for data via tcp
            except RuntimeError:
                continue
            nr_points_received = len(new) // 4  #four entries in received data are from 1 point (Idut,Qdut,Iref,Qref)
            if num_measurements < nr_points_received:
                new = new[:4 * num_measurements]  #cuts when more data is received from tcp than needed
                nr_points_received = len(new) // 4
            num_measurements -= nr_points_received
            queue.put(new)  #puts data in queue, then continues

            data[total_nr_points:total_nr_points +
                 nr_points_received] = np.array(new).reshape(nr_points_received, data.shape[1])
            total_nr_points += nr_points_received

    @staticmethod
    def construct_output_data(freq: np.ndarray, data: np.ndarray) -> np.ndarray:
        """Performs the complex division to obtain S21. Also arranges the frequency, S21, ref and dut data.
        Args:
            freq (np.ndarray(num_frequencies)): array with all frequency steps of the sweep
            data (np.ndarray(num_frequencies,4)): IQ data from the DuT [:, :2] and the Ref[:, 2:4]
        Returns:
            np.ndarray(num_frequencies,13): a numpy matrix containing all frequency, S-parameters, REF and DuT values
        """

        s21 = (data[:, 0] + 1j * data[:, 1]) / (data[:, 2] + 1j * data[:, 3])
        s21_re = np.real(s21)
        s21_im = np.imag(s21)

        magnitude = OpenVQA.get_magnitude(data)
        s21_mag = 10 * np.log10(magnitude[:, 0] / magnitude[:, 1])  #the S21 in dB

        phase = OpenVQA.get_phase(data)
        s21_ph = np.mod(phase[:, 0] - phase[:, 1], 2 * np.pi)  #phase in range ([0,2pi))

        dut_re = data[:, 0]
        dut_im = data[:, 1]
        dut_mag = magnitude[:, 0]
        dut_ph = phase[:, 0]

        ref_re = data[:, 2]
        ref_im = data[:, 3]
        ref_mag = magnitude[:, 1]
        ref_ph = phase[:, 1]
```

```python

        return (
            np.vstack(
                (freq, s21_re, s21_im, s21_mag, s21_ph, dut_re, dut_im, dut_mag, dut_ph, ref_re, ref_im, ref_mag, ref_ph)
            ).T
        )

    @staticmethod
    def get_magnitude(iq_values: np.ndarray) -> np.ndarray:
        """Calculates magnitude data from the four I and Q values.
        Args:
            iq_values (np.ndarray(num_frequencies,4)): matrix with IQ trace data (o_dut, m_dut, i_ref, q_ref)
                Returns:
                    np.ndarray: column 1: magnitude of IQ[:, 0:2] (DuT), column 2: magnitude of IQ[:, 2:4] (Ref)
        """
        vertical = iq_values.reshape((iq_values.shape[0] * 2, iq_values.shape[1] // 2))
        norm = np.linalg.norm(vertical, axis=1)
        magnitude = norm.reshape(iq_values.shape[0], iq_values.shape[1] // 2)
        return magnitude

    @staticmethod
    def get_phase(iq_values: np.ndarray) -> np.ndarray:
        """Calculates phase data from the four I and Q values.
        Args:
            iq_values (np.ndarray(num_frequencies,4)): matrix with IQ trace data (o_dut, m_dut, i_ref, q_ref)
        Returns:
            np.ndarray: column 1: phase of IQ[:, 0:2] (DuT), column 2: phase of IQ[:, 2:4] (Ref)
        """
        vertical = iq_values.reshape((iq_values.shape[0] * 2, iq_values.shape[1] // 2))
        angle = np.angle(1j * vertical[:, 1] + vertical[:, 0])
        phase = angle.reshape(iq_values.shape[0], iq_values.shape[1] // 2)
        return phase

    @staticmethod
    def stlab_dataframe(full_data: np.ndarray):
        """convert a data 2D array to a pandas DataFrame as used by Steele lab
        Args:
            full_data (np.ndarray(num_frequencies,13)): a numpy matrix containing all frequency,
                S-parameters, REF and DuT values
        Returns:
            pd.DataFrame: pandas dataframe with only frequency, S21 amplitude and S21 phase
        """
        freq = full_data[:, 0]
        s21_decibel = full_data[:, 3]
        s21_phase = full_data[:, 4]
        data = {"Frequency (Hz)": freq, "S21dB (dB)": s21_decibel, "S21 phase": s21_phase}
        return pd.DataFrame(data)

    @staticmethod
    def save_data(
        data: np.ndarray,
        gen1_setting: dict,
        gen2_setting: dict,
        temperature: float,
        save_path: str,
        path_to_notebook: str | None = None
    ) -> None:
        """Saves data and metadate in seperate timestamped folders, all in .h5 format.
        Also stores the notebook file. General instructions of the DataFolder class:
        https://gitlab.tudelft.nl/steelelab/data-explorer/-/blob/master/example_notebooks/Data%20Folder%20Usage%20Examples.ipynb
        Args:
            data (np.ndarray(num_frequencies,13)): [num_frequencies,13] a numpy matrix containing all frequency, S-parameters,
                REF and DuT values
```

```
            gen1_setting, gen2_setting (dict[str, str]): dictionaries with some of the settings of the generators
            save_path (str): the directory to save the metadata file to
            path_to_notebook (str | None): option to manually give the path to a Jupyter notebook that also will be saved next
                as metadata
        """
        if len(os.listdir(save_path)) == 0:  # Create a first dummy folder, else DataFolder does not work.
            os.mkdir(os.path.join(save_path, f"{strftime('%Y-%m-%e_%H.%M.%S')}_0000"))
        dfol = DataFolder(save_path)  #only works in a non-empty save_path folder

        # Save the notebook file with the current settings.
        if path_to_notebook is not None:
            with open(path_to_notebook, "r", encoding="utf-8") as file:
                script = file.read()  # Read the ipynb notebook.
            with open(os.path.join(dfol.folder_full_path, os.path.basename(path_to_notebook)), "w", encoding="utf-8") as file:
                file.write(script)  # Save the ipynb notebook in the dexplore generated folder.

        dataset_name = "dataset_openvqa"  # Change this to whatever you like.
        stlab_data = OpenVQA.stlab_dataframe(data)  # Convert our data 2D array to a pandas DataFrame as used by Steele Lab.
        dfol.create_1d_from_stlab_trace(dataset_name, stlab_data)  # Create the dataset.

        # Save metadata here with xarrays.
        gen1_setting_x = xr.DataArray()
        gen2_setting_x = xr.DataArray()
        red_pitaya_temperature_x = xr.DataArray()

        # Store generator data in x_array (_x).
        gen1_setting_x["Generator_1"] = gen1_setting["idn"]
        gen1_setting_x["Generator_1_power"] = gen1_setting["power"]
        gen1_setting_x["Generator_1_start_frequency"] = gen1_setting["start_freq"]
        gen1_setting_x["Generator_1_stop_frequency"] = gen1_setting["stop_freq"]
        gen1_setting_x["Generator_1_step_frequency"] = gen1_setting["freqstep"]
        gen1_setting_x["Generator_1_dwell_time"] = gen1_setting["dwell_time"]

        gen2_setting_x["Generator_1"] = gen2_setting["idn"]
        gen2_setting_x["Generator_1_power"] = gen2_setting["power"]
        gen2_setting_x["Generator_1_start_frequency"] = gen2_setting["start_freq"]
        gen2_setting_x["Generator_1_stop_frequency"] = gen2_setting["stop_freq"]
        gen2_setting_x["Generator_1_step_frequency"] = gen2_setting["freqstep"]
        gen2_setting_x["Generator_1_dwell_time"] = gen2_setting["dwell_time"]

        red_pitaya_temperature_x["temperature"] = temperature

        dfol.datasets["generator_1_settings"] = gen1_setting_x
        dfol.datasets["generator_2_settings"] = gen2_setting_x
        dfol.datasets["red_pitaya_temperature"] = red_pitaya_temperature_x
        dfol.save_data()

    @staticmethod
    def save_metadata(generator_setting, start_time: str, stop_time: str, save_path: str) -> None:
        """Alternative method for saving metadata
        Args:
            generator_setting (dict[str, str]): dictionary with some of the settings of the generator
            start_time (str): the approximate starting time of the sweep
            stop_time (str): the approximate stopping time of the sweep
            save_path (str): the path to save the metadata file to
        """
        content = []
        # For data-exploreer plotting software
        content.append(f"#␣{strftime('%Y_%m_%e_%H.%M.%S')}_OUR_VNA\n")
        content.append("#␣Info␣for␣data␣explorer\n")
        content.append("#␣Frequency␣sweep\n")
        content.append(f"{generator_setting['sweep_freq_step']}\n")
        content.append(f"{generator_setting['sweep_freq_low']}\n")
```

```
        content.append(f"{generator_setting['sweep_freq_high']}\n")
        content.append("frequency␣(Hz)\n")
        # Measurement parameters
        content.append("\n#␣Parameters\n")
        content.append(f"generator:␣{generator_setting['model_type']}␣{generator_setting['
            hw_version']}\n")
        content.append(f"generator␣A␣power:␣{generator_setting['power']}\n")
        content.append(f"sweep␣time␣step:␣{generator_setting['sweep_time_step']}\n")
        content.append(f"time␣start:␣{start_time}\n")
        content.append(f"time␣stop:␣{stop_time}\n")
        # For spyview plotting software
        content.append("\n#␣Column␣labels\n")
        content.append("1\n")
        content.append("Frequency␣(Hz)\n")
        content.append("2\n")
        content.append("S21␣Re␣()\n")
        content.append("3\n")
        content.append("S21␣Im␣()\n")
        content.append("4\n")
        content.append("S21␣magnitude␣(dB)\n")
        content.append("5\n")
        content.append("S21␣phase␣(rad)")
        content.append("6\n")
        content.append("Through-DuT␣Re␣()\n")
        content.append("7\n")
        content.append("Through-DuT␣Im␣()\n")
        content.append("8\n")
        content.append("Through-DuT␣magnitude␣(dB)\n")
        content.append("9\n")
        content.append("Through-DuT␣phase␣(rad)\n")
        content.append("10\n")
        content.append("Reference␣Re␣()\n")
        content.append("11\n")
        content.append("Reference␣Im␣()\n")
        content.append("12\n")
        content.append("Reference␣magnitude␣(dB)\n")
        content.append("13\n")
        content.append("Reference␣phase␣(rad)\n")

        # Write to file.
        with open(f"{save_path}/{strftime('%Y_%m_%e_%H.%M.%S')}_OUR_VNA.meta.txt", "w",
            encoding="utf-8") as file:
            file.writelines(content)
```

A.2.2. Plotting module (for testing)

```
"""Plotting functions useful for visualisation during debugging; not currently in use"""

import glob
import os

import matplotlib.pyplot as plt
import numpy as np
import xarray as xr


def abs_magnitude_plot(frequency: np.ndarray, magnitude: np.ndarray, logarithmic: bool) ->
    None:
    """Plot the incoming DuT and Ref magnitudes"""
    fig, ax1 = plt.subplots()

    ax1.plot(frequency, magnitude[:, 0], label="DuT", color="purple", linestyle="-")
    ax1.plot(frequency, magnitude[:, 1], label="Ref", color="green", linestyle="-")
    ax1.set_yscale("log" if logarithmic else "linear")
    ax1.xlabel("Frequency␣[GHz]")
    ax1.ylabel("Magnitude␣[V]")

    fig.legend()
    fig.tight_layout()
    #fig.savefig("project/ui/figures/abs_magnitude.png")
    plt.show()
```

```python


def abs_phase_plot(frequency: np.ndarray, phase: np.ndarray) -> None:
    """Plot the incoming DuT and Ref phases"""
    fig, ax1 = plt.subplots()

    ax1.plot(frequency, phase[:, 0], label="DuT", color="purple", linestyle="-")
    ax1.plot(frequency, phase[:, 1], label="Ref", color="green", linestyle="-")
    ax1.xlabel("Frequency␣[GHz]")
    ax1.ylabel("Phase␣[rad]")

    fig.legend()
    fig.tight_layout()
    #fig.savefig("project/ui/figures/abs_phase.png")
    plt.show()


def abs_mag_phase_plot(frequency: np.ndarray, magnitude: np.ndarray, phase: np.ndarray, logarithmic: bool = False) -> None:
    """Plot the incoming DuT and Ref magnitudes and phases seperately"""
    fig, ax = plt.subplots(1, 2, figsize=(15, 4))

    ax[0].plot(frequency, magnitude[:, 0], label="DuT", color="purple", linestyle="-")
    ax[0].plot(frequency, magnitude[:, 1], label="Ref", color="green", linestyle="-")
    ax[0].set_yscale("log" if logarithmic else "linear")
    ax[0].set_xlabel("Frequency␣[GHz]")
    ax[0].set_ylabel("Magnitude␣[V]")
    ax[0].set_title("Magnitudes␣through␣DuT␣and␣Ref")
    ax[0].legend()

    ax[1].plot(frequency, phase[:, 0], label="DuT", color="purple", linestyle="-")
    ax[1].plot(frequency, phase[:, 1], label="Ref", color="green", linestyle="-")
    ax[1].set_xlabel("Frequency␣[GHz]")
    ax[1].set_ylabel("Phase␣[rad]")
    ax[1].set_title("Phases␣through␣DuT␣and␣Ref")
    ax[1].legend()

    fig.tight_layout()
    #fig.savefig("project/ui/figures/abs_mag_phase.png")
    plt.show()


def rel_mag_phase_plot(frequency: np.ndarray, magnitude: np.ndarray, phase: np.ndarray) -> None:
    """Plot the relative magnitude and phase in 1 figure"""
    fig, ax1 = plt.subplots()
    ax1.plot(frequency, magnitude, label="Magnitude", color="red", linestyle="-")
    ax1.tick_params(axis="y", colors="red")
    ax1.set_xlabel("Frequency␣[MHz]")
    ax1.set_ylabel("Magnitude␣[dB]")

    ax2 = ax1.twinx()  #share x-axis
    ax2.plot(frequency, phase, label="Phase", color="blue", linestyle="-")
    ax2.tick_params(axis="y", colors="blue")
    ax2.set_ylabel("Phase␣[rad]")

    fig.legend()
    fig.tight_layout()
    #fig.savefig("project/ui/figures/rel_mag_phase.png")
    plt.show()


def rel_mag_phase_plot2(frequency: np.ndarray, magnitude: np.ndarray, phase: np.ndarray, logarithmic: bool) -> None:
    """Plot the relative magnitude and phase seperately"""
    fig, ax = plt.subplots(1, 2, figsize=(15, 4))

    ax[0].plot(frequency, magnitude, label="DuT", color="red", linestyle="-")
    ax[0].set_yscale("log" if logarithmic else "linear")
    ax[0].set_xlabel("Frequency␣[GHz]")
    ax[0].set_ylabel("Magnitude␣[V]")
```

```python
    ax[0].set_title("Magnitudes through DuT and Ref")
    ax[0].legend()

    ax[1].plot(frequency, phase, label="DuT", color="blue", linestyle="-")
    ax[1].set_xlabel("Frequency [GHz]")
    ax[1].set_ylabel("Phase [rad]")
    ax[1].set_title("Phases through DuT and Ref")
    ax[1].legend()

    fig.tight_layout()
    #plt.savefig("project/ui/figures/rel_mag_phase2.png")
    plt.show()


if __name__ == "__main__":
    folder = os.path.join(
        "project", "client", "application", "data", "2024-05-30_11.03.54_0016___main__", ""
    )  # don"t forget last ""
    h5_files = glob.glob(folder + "*.h5")

    data = xr.load_dataset(h5_files[1])  #[0] is generator settings, [1] is data
    print("data: ", data)
    data_variables = list(data.data_vars)
    #print("data_variables: ", data_variables)
    s21_mag = data[data_variables[0]]
    print("s21_mag: ", s21_mag)
    s21_ph = data[data_variables[1]]
    #print("s21_ph: ", s21_ph)
    dimensions = data[data_variables[0]].dims
    #print("dimensions: ", dimensions)
    frequencies = data[dimensions[0]]
    #print("frequencies: ", frequencies)

    rel_mag_phase_plot2(frequency=frequencies, magnitude=s21_mag, phase=s21_ph, logarithmic=True)
```

A.2.3. TCP client module

```python
"""Basic TCP client for retrieving measurement data"""

import socket
from struct import unpack
from types import TracebackType
from typing import Type

from project.server.protocol import TCPCommandProtocol


class TCPClient(TCPCommandProtocol):
    """Simple host:port socket client; use `with TCPClient(host, port) as c` for proper disconnect!"""

    BUFSIZE = TCPCommandProtocol.POINTS_PER_PACKET * 32
    """Receiver buffer size in bytes = optimal packet size times the size of four (64-bits) floats"""

    DEBUG = True
    """Whether to print debugging information"""

    def __init__(self, host: str, port: int) -> None:
        self.socket = socket.create_connection((host, port))
        self._reset_trigger_config()

    def __enter__(self) -> "TCPClient":
        """Enters the `with` block."""
        return self

    def send_receive(self, data: str) -> bytes:
        """Simplest form of useful communication.
        Server expects a command from client and expects client to wait for response.
        """
```

```
        if len(data) == 0:
            return b""
        if len(data) > TCPClient.BUFSIZE:
            raise ValueError(f"Data␣{data}␣is␣too␣long␣(>␣{TCPClient.BUFSIZE}).")
        self.socket.sendall(data.encode("utf-8"))
        return self.socket.recv(TCPClient.BUFSIZE)

    def start_acquisition(self) -> None:
        """Requests programmable logic to start acquisition."""
        if self.send_receive(f"{TCPClient.RUN_PL}1") != TCPCommandProtocol.RESPONSE_OK:
            raise RuntimeError("Could␣not␣start␣data␣acquisition␣on␣programmable␣logic.␣Configuration␣possibly␣incorrect.")

    def stop_acquisition(self) -> None:
        """Requests programmable logic to stop acquisition."""
        self.send_receive(f"{TCPClient.RUN_PL}0")

    def request_data(self) -> tuple[float] | tuple[float, float, float, float]:
        """Asks server for acquired data."""
        out = self.send_receive(TCPClient.DATA)
        # This should be 32 bytes or an integer multiple (in case of multiple samples).
        if len(out) % 32 != 0:
            raise RuntimeError(f"Received␣data␣not␣of␣correct␣length␣{len(out)}.")
        return TCPClient.bytes_to_float64(out)

    def send_tpp(self, time: int) -> None:
        """Sends time per point in microseconds."""
        if not isinstance(time, int):
            raise TypeError(f"Time␣per␣point␣{time}␣should␣be␣an␣integer␣in␣microseconds.")
        self.send_receive(f"{TCPClient.TPP}{time}")

    def send_dead_time(self, time: int) -> None:
        """Sends generator dead time in microseconds."""
        if not isinstance(time, int):
            raise TypeError(f"Dead␣per␣point␣{time}␣should␣be␣an␣integer␣in␣microseconds.")
        self.send_receive(f"{TCPClient.DEAD_TIME}{time}")

    def send_trigger_length(self, time: int) -> None:
        """Sends trigger pulse length in microseconds."""
        if not isinstance(time, int):
            raise TypeError(f"Trigger␣pulse␣length␣{time}␣should␣be␣an␣integer␣in␣microseconds.")
        self.send_receive(f"{TCPClient.TRIG_LEN}{time}")

    def _reset_trigger_config(self) -> None:
        """Resets trigger configuration inside PL to default values."""
        self.send_receive(f"{TCPClient.TRIG_0_CONF}0")  # no trigger output
        self.send_receive(f"{TCPClient.TRIG_1_CONF}0")  # no trigger output

    def send_trigger_config(self, trig_nr: int, positive: bool, sweep: bool = True, step: bool = True) -> None:
        """Configure an output trigger (either 0 or 1). `positive` controls the output
        (True = active-high; False = active-low). `sweep` controls whether to trigger on the first point only.
        `step` controls whether to trigger on each point of the trace (where the frequency should change).
        """
        if trig_nr not in {0, 1}:
            raise ValueError(f"Cannot␣configure␣trigger␣number␣{trig_nr};␣only␣0␣or␣1␣are␣allowed.")
        char = TCPClient.TRIG_1_CONF if trig_nr else TCPClient.TRIG_0_CONF
        bits = 0b0000
        if not positive:
            bits |= 0b0001
        if sweep:
            bits |= 0b0010
        if step:
            bits |= 0b0100
        self.send_receive(f"{char}{bits}")

    def get_queue_size(self) -> int:
```

```python
        """Queries DMA buffer queue size."""
        return int.from_bytes(self.send_receive(TCPClient.QUEUE_SIZE), byteorder="big")

    def get_server_cpu_temp(self) -> float:
        """Queries server's cpu temperature."""
        return TCPClient.bytes_to_float64(self.send_receive(TCPClient.CPU_TEMP))[0]

    @staticmethod
    def bytes_to_float64(by: bytes) -> tuple[float]:
        """Converts bytes to 64-bit floating point number."""
        return unpack(f"{len(by)␣//␣8}d", by)

    def _stop_server(self, really: bool = False) -> bool:
        """Stops TCP server. Be careful, you have to restart the server manually if stopped!
        Only use this for debugging.
        """
        if really:
            # Server should return empty byte string only if it shut down itself correctly.
            return self.send_receive(TCPClient.STOP_SERVER) == b""
        return False

    def __exit__(
        self, exc_type: Type[BaseException] | None, exc_val: BaseException | None, exc_tb: TracebackType | None
    ) -> None:
        """Leaves the `with` block."""
        if TCPClient.DEBUG:
            print("TCP␣client␣exiting.")
            if exc_type is not None:
                print(f"Exception␣occured:␣{type(exc_val).__name__}:␣{'␣'.join(exc_val.args)}")
        self.socket.close()
```

A.2.4. AnaPico APUASYN generator module

```python
"""Module for controlling generator(s) from AnaPico"""

import sys
from time import sleep
from types import TracebackType
from typing import Type

import pyvisa

try:
    from project.client.generator.hittite import HMCT2100Controller
except ModuleNotFoundError:
    from hittite import HMCT2100Controller


class APUASYN20Controller(HMCT2100Controller):
    """Programmer manual: https://www.anapico.com/download/pm_signal-generators/?wpdmdl=6829&refresh=665ecb7bc31c21717488507"""

    DEADTIME = 500  # deadtime in us, minimum setting according to manual for stable behaviour

    def __init__(self):
        ###select correct visa address###
        s = sys.platform
        if s.startswith("win"):
            # Windows
            rm = pyvisa.ResourceManager()  # no @py here for windows!
            lr = rm.list_resources()  #find the visa addresses
            usbs = [ss for ss in lr if "USB0" in ss]  #pick the usb address
        elif s.startswith("darwin"):
            # macOS
            rm = pyvisa.ResourceManager("@py")  # @py does work in macOS!
            lr = rm.list_resources()  #find the visa addresses
            usbs = [ss for ss in lr if "USB0" in ss]  #pick the usb address
        else:
```

```python
            raise EnvironmentError(f"Platform '{s}' not supported to control APUASYN20.")
        try:
            visa_address = usbs[0]
        except IndexError as err:
            raise IndexError("AnaPico USB connection not found!") from err

        ###Open the connection###
        try:
            self.gen = rm.open_resource(visa_address)
        except pyvisa.errors.Error as err:
            print(str(err), rm.list_resources())
            self.gen = rm.open_resource(rm.list_resources()[2])

        self.init()

    def __enter__(self) -> "APUASYN20Controller":
        return self

    def query(self, parameter: str) -> float | str:
        """Request data string via SCPI"""
        query = self.gen.query(parameter).strip()
        try:
            queryfloat = float(query)  # Type-casting the string to `float`.
            if queryfloat > 1e6:  # Pico returns in Hz, here make MHz representation.
                queryfloat /= 1e6  # This means that freqstep can be in Hz or MHz!!!
                return f"{format(queryfloat, '.4f')}e6"  #limit decimal places
            return str(format(queryfloat, '.4f'))  #limit decimal places
        except ValueError:
            return query

    def read_status(self) -> dict[str, str | float]:
        """Reads status and settings from device."""
        out = super().read_status()  #Inherit from hittite read_status

        out["point_count"] = self.query("SWE:POIN?")  #No option for Hittite
        out["sweep_delay"] = self.query("SWE:DEL?")  #Dead time
        #out["freq_mode"] = self.query("FREQ:MODE?")    #Frequency mode: FIXed or CW or SWEep or LIST or CHIRp
        #out["trig_source"] = self.query("TRIG:SOUR?")  #Trigger source: IMMediate or BUS or EXTernal or SYNChronous
        out["trig_type"] = self.query("TRIG:TYPE?")  #Trigger type: NORMal or POINt (no option for Hittite)
        out["locked"] = self.query("ROSC:LOCK?")  #Checks if generator is locked to exteral reference
        out["ext_freq"] = self.query("ROSC:EXT:FREQ?")
        return out

    def hardware_freq_sweep(self, start_freq: float, stop_freq: float, freqstep: float, timestep: float, power: float):
        """Sends configuration for hardware frequency sweep with external sweep trigger.
        Frequencies in megahertz; time step in microseconds; power in dBm.
        """
        dwell_time = timestep - APUASYN20Controller.DEADTIME  # Subtract dead time of 250 microseconds.
        points = int((stop_freq - start_freq) / freqstep + 1)

        self.gen.write(f"POW:AMPL {power}DBM")  # RF output power in dBm
        self.gen.write(f"SOUR:FREQ {start_freq}MHz")  # frequency in MHz
        self.gen.write(f"FREQ:STAR {start_freq}MHz")  # start frequency in MHz
        self.gen.write(f"FREQ:STOP {stop_freq}MHz")  # stop frequency in MHz
        self.gen.write(f"SWE:POIN {points}")  # nr of points in the sweep, doesn't exist for hittite
        #self.gen.write(f"FREQ:STEP {freqstep}MHz")     # frequency step size    Doesn't work?
        self.gen.write(f"SWE:DWEL {dwell_time}us")  # dwell time in microseconds
        self.gen.write(f"SWE:DEL {APUASYN20Controller.DEADTIME}us")  # dead time in microseconds

        self.gen.write("TRIG:SOUR IMM")  # trigger source: external (requires rising edge on trigger to initiate sweep)
                                    #TODO back to externaL?
```

```python
        self.gen.write("TRIG:TYPE NORM")  # trigger type: 1st trigger starts sweep
        self.gen.write("SWE:COUN 1")  # number of sweeps after a trigger

        # self.gen.write(f"ROSC:SOUR INT")                 # set internal reference clock
        # self.gen.write(f"ROSC:OUTP:FREQ 100MHz")         # only available output ref clock frequency? (pico's internal freq)
        # self.gen.write(f"ROSC:OUTP ON")                  # turn on output reference clock

        self.gen.write("ROSC:SOUR EXT")  # set external reference clock
        self.gen.write("EXT:FREQ 10MHz")  # set expected external clock frequency

    def perform_sweep(self):
        """Starts the sweep; if `block = True`, waits until it is complete."""
        super().perform_sweep()
        self.gen.write("FREQ:MODE SWE")  # frequency mode: sweep

    def __exit__(
        self, exc_type: Type[BaseException] | None, exc_val: BaseException | None, exc_tb: TracebackType | None
    ) -> None:
        """Exits the `with` block."""
        self.gen.write("OUTP OFF")
        self.gen.close()

        #how to correctly turn off? Because atm, we still have to manually turn on/off the pico every time


if __name__ == "__main__":
    with APUASYN20Controller() as pico:
        power = 13  #dBm
        freq = 4000  #for single freq, MHz

        start_freq = 6600  # MHz
        stop_freq = 6800  # MHz
        step_size = 1  # MHz
        timestep = 100  # Time step in milliseconds

        # pico.hardware_freq_sweep(start_freq, stop_freq, step_size, timestep, power)
        # pico.perform_sweep(block=False)
        # print(pico.read_status())

        # pico.single_freq(freq, power)
        # pico.read_status()
        # sleep(30)

        with HMCT2100Controller() as hmc:
            power_2 = 13  #dBm
            freq_2 = 4010  #MHz

            start_freq_2 = start_freq + 10  # MHz
            stop_freq_2 = stop_freq + 10  # MHz
            step_size_2 = step_size  # MHz
            timestep_2 = timestep  # Time step in milliseconds

            #pico.hardware_freq_sweep(start_freq, stop_freq, step_size, timestep, power)
            #hmc.hardware_freq_sweep(start_freq_2, stop_freq_2, step_size_2, timestep_2, power_2)
            #hmc.perform_sweep()
            #pico.perform_sweep()
            pico.single_freq(freq, power)
            hmc.single_freq(freq_2, power_2)
            print(hmc.read_status())
            print(pico.read_status())  # Often doesnt work? when that happens, sweep does work?
                                       # Maybe the pico needs to cool down, or need some reset before reading?

            extra_sleep_time = 20  #s
            print(f"Done!, now sleep {(timestep*1e-3*(stop_freq-start_freq)/step_size)+extra_sleep_time} seconds.")
```

```
            sleep((timestep * 1e-3 * (stop_freq - start_freq) / step_size) + extra_sleep_time)
```

A.2.5. Hittite HMC_T2100 generator module

```
"""Module for controlling generator(s) from Hittite"""

import sys
from types import TracebackType
from typing import Type

import pyvisa

from project.client.generator.base_controller import BaseSCPIGeneratorController

class HMCT2100Controller(BaseSCPIGeneratorController):
    """Programmer manual:
    https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=433deee96a655a14caf374d4fb3d9fe0e282cbd0
    """

    DEADTIME = 250     #deadtime in us given by the manual. Cannot be changed

    # https://pyvisa.readthedocs.io/projects/pyvisa-py/en/latest/

    def __init__(self):
        ###select correct visa address###
        s = sys.platform
        rm = pyvisa.ResourceManager("@py")
        if s.startswith("win"):
            # Windows
            lr = rm.list_resources()                    #find the visa addresses
            usbs = [ss for ss in lr if "ASRL11::" in ss]     #pick the "ASRL coming from usb" address
        elif s.startswith("darwin"):
            # macOS
            lr = rm.list_resources()                    #find the visa addresses
            usbs = [ss for ss in lr if "usb" in ss]     #pick the "ASRL coming from usb" address
        else:
            raise EnvironmentError(f"Platform '{s}' not supported to control HMCT2100.") # Just haven't tested linux yet
        try:
            visa_address = usbs[0]
        except IndexError as err:
            raise IndexError("Hittite USB connection not found!") from err

        ###Open the connection###
        try:
            self.gen = rm.open_resource(visa_address)
        except pyvisa.errors.Error as err:
            print(str(err), rm.list_resources())
            self.gen = rm.open_resource(rm.list_resources()[0])

        self.init()

    def __enter__(self) -> "HMCT2100Controller":
        return self

    def init(self) -> None:
        """Initialise some visa communication settings"""
        self.gen.baud_rate = 115200
        self.gen.data_bits = 8
        self.gen.parity = pyvisa.constants.Parity.none
        self.gen.stop_bits = pyvisa.constants.StopBits.one
        self.gen.read_termination = "\n"
        self.gen.write_termination = "\n"
        self.gen.timeout = 20000  # 20 seconds

        ###Important first commands###
        self.gen.write("*RST")                          # reset the generator, important!
```

```python
        self.gen.write("*CLS")                                  # clear status byte

        self.gen.write("OUTP␣OFF")                              # as precaution

    def query(self, parameter: str) -> float | str:
        """Request data string via SCPI"""
        return self.gen.query(parameter).strip()

    def read_status(self) -> dict[str, str | float]:
        """Reads status and settings from device."""
        out: dict[str, str | float] = {}
        out["idn"] = self.name()
        out["freq"] = self.query("FREQ?")                        #For fixed frequency, so no sweep
        out["start_freq"] = self.query("FREQ:STAR?")
        out["stop_freq"] = self.query("FREQ:STOP?")
        out["freqstep"] = self.query("FREQ:STEP?")
        out["power"] = self.query("POW:AMPL?")
        out["dwell_time"] = self.query("SWE:DWEL?")              #Dwell time
        out["sweep_count"] = self.query("SWE:COUN?")             #Number of full sweeps after triggering
        out["sweep_dir"] = self.query("SWE:DIR?")                #Direction of sweep (start->stop or stop->start or random)
        out["freq_mode"] = self.query("FREQ:MODE?")              #Frequency mode: CW or SWEep
        out["trig_source"] = self.query("TRIG:SOUR?")            #Trigger source: IMMediate or BUS or EXTernal
        out["oscillator"] = self.query("ROSC:SOUR?")             #Check the reference oscillator source
        return out

    def single_freq(self, freq: float, power: float):
        """Send configuration for static frequency and turn on output"""
        self.gen.write(f"POW:AMPL␣{power}DBM")                   # RF output power in dBm
        self.gen.write("FREQ:MODE␣FIX")                          # frequency mode: fixed frequency
        self.gen.write(f"SOUR:FREQ␣{freq}MHz")                   # frequency in MHz

        self.gen.write("TRIG:SOUR␣EXT")                          # trigger source: external
                                                                 # (requires rising edge on trigger to initiate sweep)

        self.gen.write("ROSC:SOUR␣EXT")                          # set external reference clock

        self.gen.write("OUTP␣1")                                 # enable the RF output.
        # self.gen.write(f"INIT:IMM")                            # immediately initiates the sweep, according to manual.
                                                                 # Not true, but is needed for actual sweeping?

    def hardware_freq_sweep(self, start_freq: float, stop_freq: float, freqstep: float, timestep: float, power: float):
        """Sends configuration for hardware frequency sweep, with external sweep trigger.
        Frequencies in megahertz; time step in microseconds; power in dBm.
        """
        dwell_time = timestep - HMCT2100Controller.DEADTIME    # Subtract dead time of 250 microseconds.

        self.gen.write(f"POW:AMPL␣{power}DBm")                   # RF output power to 0 dBm
        self.gen.write(f"SOUR:FREQ␣{start_freq}MHz")             # frequency in MHz
        self.gen.write(f"FREQ:STAR␣{start_freq}MHz")             # start frequency in MHz
        self.gen.write(f"FREQ:STOP␣{stop_freq}MHz")              # stop frequency in MHz
        self.gen.write(f"FREQ:STEP␣{freqstep}MHz")               # frequency step size, doesn't work for pico
        self.gen.write("FREQ:MODE␣SWE")                          # frequency mode: sweep
        self.gen.write(f"SWE:DWEL␣{dwell_time}us")               # dwell time in microseconds

        self.gen.write(f"TRIG:SOUR␣IMM")                         # trigger source: external  # TODO change to ext
                                                                 # (requires rising edge on trigger to initiate sweep)

    def perform_sweep(self):
        """Starts the sweep; if `block = True`, waits until it is complete."""
```

```
        self.gen.write("OUTP 1")                            # enable the RF output.
        self.gen.write("INIT:IMM")                          # immediately initiates the sweep, according to manual.
                                                            # Not true, but is needed for actual sweeping?

    def __exit__(
        self, exc_type: Type[BaseException] | None, exc_val: BaseException | None, exc_tb: TracebackType | None
    ) -> None:
        """Exits the `with` block."""
        self.gen.write("OUTP OFF")
        self.gen.close()


if __name__ == "__main__":
    with HMCT2100Controller() as hmc:
        start_freq = 1000  # MHz
        stop_freq = 2000  # MHz
        step_size = 100  # MHz
        timestep = 300  # Time step in milliseconds

        hmc.hardware_freq_sweep(start_freq, stop_freq, step_size, timestep, -20)
        print(hmc.read_status())
        hmc.perform_sweep()

        while True:
            pass
```

A.2.6. Jupyter GUI

```
# %% [markdown]
# # OpenVNA Notebook

# %% [markdown]
# ## Imports

# %%
import os
if "project" not in os.listdir():
    os.chdir(
        os.path.dirname(os.path.dirname(os.path.dirname(os.path.dirname(os.getcwd()))))
    )  # change working directory to parent folder of 'project'
if "project" not in os.listdir():
    raise EnvironmentError("You are not in the correct working directory; 'project' folder was not found. Cwd:", os.getcwd())

import glob
import random
import xarray as xr
import numpy as np
import threading
import ipywidgets as widgets
import bokeh.plotting as plt
import bokeh.models as model
from bokeh.io import output_notebook, reset_output
from jupyter_bokeh.widgets import BokehModel  #used this to finally get live plot updates in ipynb in vscode and jupyterlab, doesn't work in colab
from time import sleep
from IPython.display import display, clear_output

import project.client.application.dexplore as dx
from project import generator, OpenVQA

# %% [markdown]
# ## Setup

# %%
GEN_A = generator.HMCT2100Controller
GEN_B = generator.APUASYN20Controller
# GEN_B = generator.BaseSCPIGeneratorController  # for testing
```

```
# Generator(s) to control during experiment.

OpenVQA.PATH_TO_NOTEBOOK = os.path.join("project", "client", "ui", "notebook", "OpenVQA.ipynb
    ")
# Full path to this Jupyter notebook; it will be saved as metadata unless set to None.

# %% [markdown]
# ## Initiate widgets

# %%
freq_low = widgets.BoundedFloatText(
    value=5000,
    min=0,
    max=10000,
    step=100,
    description='Low frequency (MHz):',
    disabled=False,
    continuous_update=False,
    style={'description_width': 'initial'}
)

freq_high = widgets.BoundedFloatText(
    value=7000,
    min=0,
    max=10000,
    step=100,
    description='High frequency (MHz):',
    disabled=False,
    continuous_update=False,
    style={'description_width': 'initial'}
)

freq_step = widgets.BoundedFloatText(
    value=500,
    min=0.001,
    max=5000,
    step=0.001,
    description='Step frequency (MHz):',
    disabled=False,
    continuous_update=False,
    style={'description_width': 'initial'}
)

time_step = widgets.BoundedIntText(
    value=10000,
    min=0,
    max=10000,
    step=1,
    description='Time step (ms):',
    disabled=False,
    continuous_update=False,
    style={'description_width': 'initial'}
)

start_button = widgets.ToggleButton(value=False, description='Sweep', disabled=False,
    button_style='')

quit_button = widgets.Button(description='Quit GUI', tooltip='Quit GUI', disabled=False,
    button_style='warning')

#grouping the widgets
items = [freq_low, freq_high, freq_step, time_step]
left_box = widgets.HBox([items[0], items[1]], description="Frequency range")
right_box = widgets.HBox([items[2], items[3]], description="Step settings")
button_box = widgets.HBox([start_button, quit_button], description="Buttons")
controls = widgets.VBox([left_box, right_box, button_box], description="Sweep settings")

#print(controls.keys)

#getting some basic values and arrays
#num_frequencies = int( (freq_high.value-freq_low.value)/freq_step.value )
```

```
#frequencies = np.linspace(freq_low.value, freq_high.value, int(num_frequencies))

#outarray = np.zeros((num_frequencies,13))       #array where the data will be put into

# %% [markdown]
# ## Run GUI

# %%
output_notebook()  #loads Bokeh


class GUI:

    def __init__(self) -> None:
        self.thread_on = False
        self.started_sweep = False
        self.vqa = OpenVQA(GEN_A(), GEN_B())
        start_button.value = False

    def __enter__(self) -> "GUI":
        return self

    # update the button values and actions
    def actionupdate(self, freq_low, freq_high, freq_step, time_step, start_button) -> None:
        sleep(0.01)
        self.freq_low = freq_low
        self.freq_high = freq_high
        self.freq_step = freq_step
        self.time_step = time_step

        self.num_frequencies = int((freq_high - freq_low) / freq_step + 1)
        self.frequencies = np.linspace(self.freq_low, self.freq_high, int(self.num_frequencies))
        self.S21mag = np.zeros(self.num_frequencies)
        self.S21ph = np.zeros(self.num_frequencies)

        # update the start button
        if start_button:
            print('On! ')
            if self.started_sweep == False:
                self.started_sweep = True

                # create a new plot with a title and axis labels
                self.plot = plt.figure(
                    title="S21 Magnitude and Phase for Frequency range",
                    sizing_mode="stretch_width",
                    height=500,
                    x_range=model.Range1d(self.freq_low, self.freq_high),
                    x_axis_label='Frequency [Hz]',
                    y_range=model.Range1d(0, 0.1),
                    # y_axis_type="log",
                    y_axis_label='S21 Magnitude'
                )  #TODO change y range or make it logarithmic

                # add a line renderer with legend and line thickness to the plot
                self.mag_line = self.plot.line(
                    self.frequencies, self.S21mag, legend_label="Magnitude", line_width=2, color='red'
                )

                self.plot.extra_y_scales = {"linear_phase": model.LinearScale()}
                #self.plot.extra_x_ranges['linear_phase'] = model.Range1d(0, 2*np.pi)
                self.plot.extra_y_ranges = {
                    "linear_phase": model.Range1d(start=0, end=2 * np.pi)
                }  #is required, so automatic scaling is more difficult?
                self.plot.add_layout(model.LinearAxis(y_range_name="linear_phase", axis_label="S21 phase (rad)"), 'right')

                self.ph_line = self.plot.line(
                    self.frequencies,
                    self.S21ph,
```

```
                        y_range_name="linear_phase",
                        legend_label="Phase",
                        line_width=2,
                        color='blue'
                    )

                    display(BokehModel(self.plot))

                    self.thread_on = True
                    #start live plotting thread
                    self.thread = threading.Thread(target=self.update_plot, name="liveplotting")
                    self.thread.start()
                    #start a sweep with the settings
                    self.vqa.sweep_acquire_2_generators(
                        trigger_per_step=False,
                        freq_low=self.freq_low,
                        freq_high=self.freq_high,
                        freqstep=self.freq_step,
                        timestep=1000 * self.time_step
                    )
        else:
            print('OFF')
            self.thread_on = False
            self.started_sweep = False
            #if self.thread.is_alive():    #thread is not alive when self.startedsweep is set to False
            #    print('join thread!')
            #    self.thread.join()

    def button_quit(self, b) -> None:
        #leave GUI on quit button. Does this fully and safely close everything?
        print("stop!!")
        if self.thread.is_alive():
            self.thread_on = False
        #    self.thread.join()          #thread is not alive when self.startedsweep is set to False
        #stop showing the controls
        widgetout.close()
        controls.close()
        #self.vqa.close()     #does not work
        #BokehModel(p).close()#does not work
        #clear_output()        #does nothing?
        reset_output()

    #update function for live plotting
    def update_plot(self) -> None:
        """Executed by worker thread."""
        total_nr_points = 0
        while True:
            # print(threading.enumerate())  # useful to check whether a ton of threads were started
            if not self.thread_on:  #stop this updater loop
                print("THREAD STOPPED!")
                return
            if hasattr(self.vqa, "queue"):
                new = self.vqa.queue.get()  #takes 1 data packet from the queue (or waits for it)
                #new = np.random.rand(4*1)     # TODO debugging (does 1 point per loop cycle now)
                # sleep(0.3) # TODO debugging

                nr_points_received = len(new) // 4  #four entries in received data are from 1 point (Idut,Qdut,Iref,Qref)
                if self.num_frequencies < nr_points_received:
                    new = new[:4 * self.num_frequencies]  #cuts when more data is received from queue than needed
                    nr_points_received = len(new) // 4

                self.num_frequencies -= nr_points_received

                #sorts the 4 measurement over 2 columns (ref and dut interleaved under each
```

```
                    other)
                vertical = np.array(new).reshape(nr_points_received * 2, 2)
                rows = vertical.shape[0]
                columns = vertical.shape[1]
                #calculate ref and dut magnitude
                norm = np.linalg.norm(vertical, axis=1)
                magnitude = norm.reshape(rows // 2, columns)
                #calculate ref and dut phase
                angle = np.angle(1j * vertical[:, 1] + vertical[:, 0])
                phase = angle.reshape(rows // 2, columns)

                #calculate the S21 mag (not in dB!!!!) and phase ([0,2pi)) and add to existing array.
                self.S21mag[total_nr_points:total_nr_points + nr_points_received] = magnitude[:, 1] / magnitude[:, 0]
                self.S21ph[total_nr_points:total_nr_points + nr_points_received] = np.mod(phase[:, 1] - phase[:, 0], 2 * np.pi)

                total_nr_points += nr_points_received

                #print('\n temp_freq_axis: ', temp_freq_axis, '\n s21_mag: ', self.s21mag)
                # self.mag_line.data_source.data = dict(x=self.frequencies, y=self.S21mag)
                self.mag_line.data_source.data = dict(x=self.frequencies, y=self.S21mag)
                self.ph_line.data_source.data = dict(x=self.frequencies, y=self.S21ph)
                if self.num_frequencies <= 0:
                    print("all points acquired; thread stopped")
                    #print("S21mag ipynb: ",self.S21mag)    #TODO debugging the different S21s
                    #print("S21ph ipynb: ",self.S21ph)    #TODO debugging the different S21s
                    return

    def __exit__(self, *args, **kwargs) -> None:
        pass


with GUI() as ui:
    # updater function for widgets.interactive
    def buttonupdate(freq_low, freq_high, freq_step, time_step, start_button) -> None:
        ui.actionupdate(freq_low, freq_high, freq_step, time_step, start_button)

    #The items from the dict (which are the widget types) are read in int_outp by using item.value, then passed as arguments to the update fuction.
    widgetout = widgets.interactive_output(
        buttonupdate, {
            'freq_low': freq_low,
            'freq_high': freq_high,
            'freq_step': freq_step,
            'time_step': time_step,
            'start_button': start_button
        }
    )

    # show the results and iPywidgets
    display(controls, widgetout)

    #perform action when quit button is pressed
    quit_button.on_click(ui.button_quit)

#TODO after a sweep has finished, turning it off and then on again does load the thread again, but queue.get() does not work anymore.
#It has been tested that the queue does fill up in the second round, but queue.get() does not notice or is stuck in its waiting state. Stop testing now

#OKE Pico was not being triggered because of splitter in the trigger path
```

A.2.7. Windfreak SynthHD generator module

```
"""File with Python experiments controlling the WindfreakTech RF generator"""

from collections.abc import Iterable
from glob import glob
```

```python
import sys
from time import perf_counter_ns, sleep, strftime

from serial.serialutil import SerialException
from windfreak import SynthHD


class SynthHDController(SynthHD):
    """Extension of SynthHD with control functions.
    https://windfreaktech.com/wp-content/uploads/2016/12/WFT_SerialProgramming_API_10b.pdf
    explains the standard control function"""

    DEBUG = True
    """Whether to print debug information during execution of code in this class."""

    def __init__(self, synth_port_name: str | None = None) -> None:
        """Detects to which port the SynthHD is connected.

        Args:
            synth_port_name (str | None): (part of) name of the device for identification on Linux and macOS.
                On Windows, this is the COM number (f.i. COM7). If None, an auto search will
                be performed and the first SynthHD that is found will be connected to. Defaults to None.
        """
        self.connected = False
        all_ports = self.get_ports()
        if synth_port_name is None:
            self._auto_search_connect(all_ports)
        else:
            p = self._manual_connect(synth_port_name, all_ports)
            self.connect(p)
            if SynthHDController.DEBUG: print(f"Manual connect: connected to {p}.")
        self.connected = True
        super().init()

    def get_ports(self) -> list[str]:
        """Finds active serial ports. Method was based on
        https://stackoverflow.com/questions/12090503/listing-available-com-ports-with-python.

        Returns:
            list[str]: serial ports.
        """
        s = sys.platform
        if s.startswith("win"):
            # Windows
            try_ports = [f"COM{nr}" for nr in range(1, 256)]
        elif s.startswith('linux') or s.startswith("cygwin"):
            # This excludes your current terminal '/dev/tty'.
            try_ports = glob("/dev/tty[A-Za-z]*")
        elif s.startswith("darwin"):
            # macOS
            try_ports = glob("/dev/tty.*")
        else:
            raise EnvironmentError(f"Platform '{s}' not supported to control SynthHD.")
        return try_ports

    def connect(self, devpath: str) -> None:
        """Calls the SynthHD initialiser."""
        super().__init__(devpath)

    def _auto_search_connect(self, all_ports: Iterable[str]) -> None:
        for p in all_ports:
            try:
                self.connect(p)
                if SynthHDController.DEBUG: print(f"Auto search: connected to port {p}.")
                return
            except (SerialException, TimeoutError):
                if SynthHDController.DEBUG: print(f"Auto search: port {p} unavailable.")
        raise SerialException(f"No device attached to any of the ports {all_ports}.")

```

```python
    @staticmethod
    def _manual_connect(port_name: str, all_ports: Iterable[str]) -> str:
        possible = set()
        for p in all_ports:
            if port_name in p:
                possible.add(p)
            if port_name == p:
                possible = {p}
                break
        if len(possible) == 1:
            return possible.pop()
        elif len(possible) > 1:
            raise NameError(f"Device with name '{port_name}' is ambiguous; choose from {possible}.")
        else:
            raise NameError(f"Device with name '{port_name}' not found; all ports: {all_ports}")

    def _read_settings(self) -> dict[str, str]:
        """Reads current settings of the SynthHD."""
        di = {}
        for setting in self.API:
            try:
                di[setting] = str(super().read(setting))
            except Exception as err:
                di[setting] = str(err)
        return di

    def single_freq(self, channel: int, freq: float, power: float = -10.) -> None:
        """Enables a channel on a given frequency with a given power.

        Args:
            channel (int): RF generator channel number (A=0, B=1);
            freq (float): frequency in megahertz;
            power (float): output power in dBm. Defaults to power_low.
        """
        self[channel].write("frequency", freq)
        self[channel].power = power
        self[channel].enable = True

    def hardware_freq_sweep(
        self, channel: int, freq_low: float, freq_high: float, freqstep: float, timestep: float, single: bool = False
    ) -> None:
        """Write these settings only (no more no less) to get a correct frequency sweep.

        Args:
            channel (int): RF generator channel number (A=0, B=1);
            freq_low (float): start frequency in megahertz;
            freq_high (float): stop frequency in megahertz;
            freqstep (float): step frequency in megahertz;
            timestep (float): time to wait between frequencies in milliseconds;
            single (bool): single frequency sweep or continuous sweep. Defaults to False.
        """
        self[channel].write("sweep_freq_low", freq_low)  # MHz
        self[channel].write("sweep_freq_high", freq_high)
        self[channel].write("sweep_freq_step", freqstep)
        self[channel].write("sweep_time_step", timestep)  # ms
        self[channel].write("sweep_cont", 0 if single else 1)  #determines if cycle resumes after highest frequency.
        self[channel].write("sweep_single", 1)  # This is set to 1 for both single and continuous sweeps!
        self[channel].write("sweep_type", 0)  # linear

    def turn_off(self, channels: tuple[int, ...] = (0, 1)) -> None:
        """Quickly turn off both channels."""
        for i in channels:
            self[i].enable = False

    def triggered_diff_freq_sweep(self, trigger_per_step: bool, freq_low, freq_high, freqstep, timestep) -> None:
```

```
        """Perform a differential frequency sweep, triggered by external trigger
        For single step triggering, we do want sweep continuous. For continuous sweep triggering,
        the trigger seems to only start a new cycle as we want with sweep_cont=0
        """
        self[0].write("sweep_diff_freq", 0.001)  #only set this on 1 channel if you want a nonzero value
        self[0].write("sweep_diff_meth", 2)  # 0: no diff sweep 1:freqB = freqA-diff_freq 2: - => +
        self[1].write(
            "trig_function", 3 if trigger_per_step else 1
        )  #0: no trig, 1: trig full sweep, 2: trig 1 step, 3: stop all, 4: on/off
        self.hardware_freq_sweep(
            channel=0,
            freq_low=freq_low,
            freq_high=freq_high,
            freqstep=freqstep,
            timestep=timestep,
            single=not trigger_per_step
        )
        self.hardware_freq_sweep(
            channel=1,
            freq_low=freq_low,
            freq_high=freq_high,
            freqstep=freqstep,
            timestep=timestep,
            single=not trigger_per_step
        )
        self[0].enable = True
        self[1].enable = True

    def control_power(self, channel, power_low, power_high) -> None:
        self[channel].write("sweep_power_low", power_low)  # dBm
        self[channel].write("sweep_power_high", power_high)

    def measure_state_reset(self) -> dict[str, str]:
        """Measures the current state of the SynthHD, only when a frequency sweep is active.
        Important: if you just call _read_settings, this will count as a trigger,
        if the trigger was configured as frequency sweep or frequency step.
        Therefore, the current sweep needs to be interrupted, then the state can be read,
        and finally the sweep will reset to the lowest frequency.
        """
        self[1].write("sweep_single", 0)
        self[0].write("sweep_single", 0)
        settings = self._read_settings()  #now reads settings of channel A right?
        self[1].write("sweep_single", 1)
        self[0].write("sweep_single", 1)
        return settings

    def __del__(self) -> None:
        if self.connected:
            self.turn_off(channels=(0, 1))
            self.close()


if __name__ == "__main__":
    sy = SynthHDController()  #'/dev/tty.usbmodem206C34714E561')

    def _onefixed_onesweep() -> None:
        """With 3.7->10 GHz mixer; goal is to know when the lowest frequency is at the output.
        Beware: if the scope shows nonsense, unplug and replug the SynthHD device!"""
        sy.single_freq(0, 5000, power=-5)
        sleep(0.3)
        sy.hardware_freq_sweep(
            1, freq_low=4999.9911, freq_high=4999.9981, freqstep=0.003, timestep=1, single=False
        )  # Works but first ~~ 5 ms are unstable (first frequency of sweep sometimes invisible).

    def _twosweep_locked_or_not() -> None:
```

```
        # single_freq(sy, 0, 20)
        # single_freq(sy, 1, 22)
        sy.hardware_freq_sweep(0, freq_low=4999.9911, freq_high=4999.9981, freqstep=0.003,
            timestep=1, single=False)
        sy.hardware_freq_sweep(1, freq_low=4999.9911, freq_high=4999.9981, freqstep=0.003,
            timestep=1, single=False)
        # sy[0].write(API)  API not complete; edit dict if necessary?

    """
    # Measuring the lock time
    sy[1].read("rf_enable")
    t0 = perf_counter_ns()
    while not sy[1].lock_status:
        pass  # print(sy[1].frequency)
    t1 = perf_counter_ns()
    print((t1 - t0) * 1E-6, "ms")
    """

    sy.turn_off()  #first turn off both channels
    # sy.triggered_freq_sweep(step=False)    #Test sweeping of 2 channels simultaneously.
    # pprint.pprint(sy.read_settings())
    sy.turn_off()
    """if input("Save current script? Press space-enter!") == " ":
        save_current_file(os.path.join(r"C:/temp/upl/wft/", strftime("%Y_%m_%d-%H%M%S") + ".
            py"))
    # print(read_settings(sy))
    """
```

A.2.8. Windfreak SynthHD API

This code was written by Windfreak Technologies, to provide a Python API for the SynthHD dual output RF generator [24].

```
from .device import SerialDevice
from collections.abc import Sequence


class SynthHDChannel:

    def __init__(self, parent, index):
        self._parent = parent
        self._index = index
        model = self._parent.model
        if model == 'SynthHD␣v1.4':
            self._f_range = {'start': 53.e6, 'stop': 13999.999999e6, 'step': 0.1}
            self._p_range = {'start': -80., 'stop': 20., 'step': 0.01}
            self._vga_range = {'start': 0, 'stop': 45000, 'step': 1}
        else:
            self._f_range = None
            self._p_range = None
            self._vga_range = None

    def init(self):
        """Initialize device."""
        self.enable = False
        f_range = self.frequency_range
        if f_range is not None:
            self.frequency = f_range['start']
        p_range = self.power_range
        if p_range is not None:
            self.power = p_range['start']
        self.phase = 0.
        self.temp_compensation_mode = '10␣sec'

    def write(self, attribute, *args):
        self.select()
        self._parent.write(attribute, *args)

    def read(self, attribute, *args):
        self.select()
        return self._parent.read(attribute, *args)
```

```python

    def select(self):
        """Select channel."""
        self._parent.write('channel', self._index)

    @property
    def frequency_range(self):
        """Frequency range in Hz.

        Returns:
            dict: frequency range or None
        """
        return None if self._f_range is None else self._f_range.copy()

    @property
    def frequency(self):
        """Get frequency in Hz.

        Returns:
            float: frequency in Hz
        """
        return self.read('frequency') * 1e6

    @frequency.setter
    def frequency(self, value):
        """Set frequency in Hz.

        Args:
            value (float / int): frequency in Hz
        """
        if not isinstance(value, (float, int)):
            raise ValueError('Expected␣float␣or␣int.')
        f_range = self.frequency_range
        if f_range is not None and not f_range['start'] <= value <= f_range['stop']:
            raise ValueError('Expected␣float␣in␣range␣[{},␣{}]␣Hz.'.format(
                             f_range['start'], f_range['stop']))
        self.write('frequency', value / 1e6)

    @property
    def power_range(self):
        """Power range in dBm.

        Returns:
            dict: power range or None
        """
        return None if self._p_range is None else self._p_range.copy()

    @property
    def power(self):
        """Get power in dBm.

        Returns:
            float: power in dBm
        """
        return self.read('power')

    @power.setter
    def power(self, value):
        """Set power in dBm.

        Args:
            value (float / int): power in dBm
        """
        if not isinstance(value, (float, int)):
            raise TypeError('Expected␣float␣or␣int.')
        self.write('power', value)

    @property
    def calibrated(self):
        """Calibration was successful on frequency or amplitude change.

```

```
        Returns:
            bool: calibrated
        """
        return self.read('calibrated')

    @property
    def temp_compensation_modes(self):
        """Temperature compensation modes.

        Returns:
            tuple: tuple of str of modes
        """
        return ('none', 'on set', '1 sec', '10 sec')

    @property
    def temp_compensation_mode(self):
        """Temperature compensation mode.

        Returns:
            str: mode
        """
        return self.temp_compensation_modes[self.read('temp_comp_mode')]

    @temp_compensation_mode.setter
    def temp_compensation_mode(self, value):
        modes = self.temp_compensation_modes
        if not value in modes:
            raise ValueError('Expected str in set {}.'.format(modes))
        self.write('temp_comp_mode', modes.index(value))

    @property
    def vga_dac_range(self):
        """VGA DAC value range.

        Returns:
            dict: VGA DAC range or None
        """
        return None if self._vga_range is None else self._vga_range.copy()

    @property
    def vga_dac(self):
        """Get raw VGA DAC value

        Returns:
            int: value
        """
        return self.read('vga_dac')

    @vga_dac.setter
    def vga_dac(self, value):
        """Set raw VGA DAC value.

        Args:
            value (int): value
        """
        if not isinstance(value, int):
            raise TypeError('Expected int.')
        self.write('vga_dac', value)

    @property
    def phase_range(self):
        """Phase step range.

        Returns:
            dict: range
        """
        return {
            'start': 0.,
            'stop': 360.,
            'step': .001,
        }
```

```

    @property
    def phase(self):
        """Get phase step value.

        Returns:
            float: value in degrees
        """
        return self.read('phase_step')

    @phase.setter
    def phase(self, value):
        """Set phase step value.

        Args:
            value (float / int): phase in degrees
        """
        if not isinstance(value, (float, int)):
            raise TypeError('Expected␣float␣or␣int.')
        self.write('phase_step', value)

    @property
    def rf_enable(self):
        """RF output enable.

        Returns:
            bool: enable
        """
        return self.read('rf_enable')

    @rf_enable.setter
    def rf_enable(self, value):
        if not isinstance(value, bool):
            raise ValueError('Expected␣bool.')
        self.write('rf_enable', value)

    @property
    def pa_enable(self):
        """PA enable.

        Returns:
            bool: enable
        """
        return self.read('pa_power_on')

    @pa_enable.setter
    def pa_enable(self, value):
        if not isinstance(value, bool):
            raise ValueError('Expected␣bool.')
        self.write('pa_power_on', value)

    @property
    def pll_enable(self):
        """PLL enable.

        Returns:
            bool: enable
        """
        return self.read('pll_power_on')

    @pll_enable.setter
    def pll_enable(self, value):
        if not isinstance(value, bool):
            raise ValueError('Expected␣bool.')
        self.write('pll_power_on', value)

    @property
    def enable(self):
        """Get output enable.

        Returns:
```

```
            bool: enabled
        """
        return self.rf_enable and self.pll_enable and self.pa_enable

    @enable.setter
    def enable(self, value):
        """Set output enable.

        Args:
            value (bool): enable
        """
        if not isinstance(value, bool):
            raise TypeError('Expected␣bool.')
        self.rf_enable = value
        self.pll_enable = value
        self.pa_enable = value

    @property
    def lock_status(self):
        """PLL lock status.

        Returns:
            bool: locked
        """
        return self.read('pll_lock')


class SynthHDv2Channel(SynthHDChannel):

    def __init__(self, parent, index):
        self._parent = parent
        self._index = index
        model = self._parent.model
        if model == 'SynthHD␣v2':
            self._f_range = {'start': 10.e6, 'stop': 15000.e6, 'step': 0.1}
            self._p_range = {'start': -70., 'stop': 20., 'step': 0.01}
            self._vga_range = {'start': 0, 'stop': 4000, 'step': 1}
            self._cspacing_range = {'start': 0.1, 'stop': 1000., 'step': 0.1}
        elif model == 'SynthHD␣PRO␣v2':
            self._f_range = {'start': 10.e6, 'stop': 24000.e6, 'step': 0.1}
            self._p_range = {'start': -70., 'stop': 20., 'step': 0.01}
            self._vga_range = {'start': 0, 'stop': 4000, 'step': 1}
            self._cspacing_range = {'start': 0.1, 'stop': 1000., 'step': 0.1}
        else:
            self._f_range = None
            self._p_range = None
            self._vga_range = None
            self._cspacing_range = None

    @property
    def channel_spacing_range(self):
        """Channel Spacing Range in Hz.

           Returns:
               dict: channel spacing range or None
        """
        return None if self._cspacing_range is None else self._cspacing_range.copy()

    @property
    def channel_spacing(self):
        """Channel Spacing in Hz

           Returns:
               float: Channel Spacing setting in Hz
        """
        return self.read('channelspacing')

    @channel_spacing.setter
    def channel_spacing(self,value):
        """Set Channel Spacing in Hz.

```

```python
            Args:
                float: Channel spacing in Hz
        """
        if not isinstance(value, (float, int)):
            raise ValueError('Expected float or int.')
        cs_range = self.channel_spacing_range
        if cs_range is not None and not cs_range['start'] <= value <= cs_range['stop']:
            raise ValueError('Expected float in range [{}, {}] Hz.'.format(
                                 cs_range['start'], cs_range['stop']))
        self.write('channelspacing', value)


class SynthHD(SerialDevice, Sequence):

    API = {
        # name                type    write       read
        'channel':            (int,   'C{}',      'C?'),  # Select channel
        'frequency':          (float, 'f{:.8f}', 'f?'),  # Frequency in MHz
        'power':              (float, 'W{:.3f}', 'W?'),  # Power in dBm
        'calibrated':         (bool,  None,       'V'),
        'temp_comp_mode':     (int,   'Z{}',      'Z?'),
        'vga_dac':            (int,   'a{}',      'a?'),  # VGA DAC value [0, 45000]
        'phase_step':         (float, '~{:.3f}', '~?'),  # Phase step in degrees
        'rf_enable':          (bool,  'h{}',      'h?'),
        'pa_power_on':        (bool,  'r{}',      'r?'),
        'pll_power_on':       (bool,  'E{}',      'E?'),
        'model_type':         (str,   None,       '+'),   # Model type
        'serial_number':      (int,   None,       '-'),   # Serial number
        'fw_version':         (str,   None,       'v0'),  # Firmware version
        'hw_version':         (str,   None,       'v1'),  # Hardware version
        'sub_version':        (str,   None,       'v2'),  # Sub-version: "HD" or "HDPRO". Only Synth HD >= v2.
        'save':               ((),    'e',        None),  # Program all settings to EEPROM
        'reference_mode':     (int,   'x{}',      'x?'),
        'trig_function':      (int,   'w{}',      'w?'),
        'pll_lock':           (bool,  None,       'p'),
        'temperature':        (float, None,       'z'),   # Temperature in Celsius
        'ref_frequency':      (float, '*{:.8f}', '*?'),  # Reference frequency in MHz
        'channelspacing':     (float, 'i{:.1f}', 'i?'),  # Channel spacing in Hz

        'sweep_freq_low':     (float, 'l{:.8f}', 'l?'),  # Sweep lower frequency in MHz
        'sweep_freq_high':    (float, 'u{:.8f}', 'u?'),  # Sweep upper frequency in MHz
        'sweep_freq_step':    (float, 's{:.8f}', 's?'),  # Sweep frequency step in MHz
        'sweep_time_step':    (float, 't{:.3f}', 't?'),  # Sweep time step in [4, 10000] ms
        'sweep_power_low':    (float, '[{:.3f}', '[?'),  # Sweep lower power [-60, +20] dBm
        'sweep_power_high':   (float, ']{:.3f}', ']?'),  # Sweep upper power [-60, +20] dBm
        'sweep_direction':    (int,   '^{}',      '^?'),  # Sweep direction
        'sweep_diff_freq':    (float, 'k{:.8f}', 'k?'),  # Sweep differential frequency in MHz
        'sweep_diff_meth':    (int,   'n{}',      'n?'),  # Sweep differential method
        'sweep_type':         (int,   'X{}',      'X?'),  # Sweep type {0: linear, 1: tabular}
        'sweep_single':       (bool,  'g{}',      'g?'),
        'sweep_cont':         (bool,  'c{}',      'c?'),

        'am_time_step':       (int,   'F{}',      'F?'),  # Time step in microseconds
        'am_num_samples':     (int,   'q{}',      'q?'),  # Number of samples in one burst
        'am_cont':            (bool,  'A{}',      'A?'),  # Enable continuous AM modulation
        'am_lookup_table':    ((int, float), '@{}a{:.3f}', '@{}a?'),  # Program row in lookup table in dBm

        'pulse_on_time':      (int,   'P{}',      'P?'),  # Pulse on time in range [1, 10e6] us
        'pulse_off_time':     (int,   'O{}',      'O?'),  # Pulse off time in range [2, 10e6] uS
        'pulse_num_rep':      (int,   'R{}',      'R?'),  # Number of repetitions in range [1, 65500]
        'pulse_invert':       (bool,  ':{}',      ':?'),  # Invert pulse polarity
        'pulse_single':       ((),    'G',        None),
        'pulse_cont':         (bool,  'j{}',      'j?'),
        'dual_pulse_mod':     (bool,  'D{}',      'D?'),

        'fm_frequency':       (int,   '<{}',      '<?'),
        'fm_deviation':       (int,   '>{}',      '>?'),
        'fm_num_samples':     (int,   ',{}',      ',?'),
```

```
        'fm_mod_type':      (int,   ';{}',     ';?'),
        'fm_cont':          (bool,  '/{}',     '/?'),
    }

    def __init__(self, devpath):
        super().__init__(devpath)
        self._model = None
        self._model = self.model
        if 'v2' in self.model:
            channel_type = SynthHDv2Channel
        else:
            channel_type = SynthHDChannel
        self._channels = [channel_type(self, index) for index in range(2)]

    def __getitem__(self, key):
        return self._channels.__getitem__(key)

    def __len__(self):
        return self._channels.__len__()

    def init(self):
        """Initialize device: put into a known, safe state."""
        self.reference_mode = 'internal 27mhz'
        self.trigger_mode = 'disabled'
        self.sweep_enable = False
        self.am_enable = False
        self.pulse_mod_enable = False
        self.dual_pulse_mod_enable = False
        self.fm_enable = False
        for channel in self:
            channel.init()

    @property
    def model(self):
        """Model version. This is the binned version that dictates API support.

        Returns:
            str: model version or None if unsupported
        """
        if self._model is not None:
            return self._model
        hw_ver = self.hardware_version
        if 'Version 2.' in hw_ver:
            sub_ver = self.read('sub_version')
            if sub_ver == 'HD':
                return 'SynthHD v2'
            elif sub_ver == 'HDPRO':
                return 'SynthHD PRO v2'
            else:
                # Unsupported sub-version. Return None.
                return None
        elif 'Version 1.4' in hw_ver:
            return 'SynthHD v1.4'
        else:
            # Unsupported hardware version. Return None.
            return None

    @property
    def model_type(self):
        """Model type.

        Returns:
            str: model
        """
        return self.read('model_type')

    @property
    def serial_number(self):
        """Serial number

        Returns:
```

```
            int: serial number
        """
        return self.read('serial_number')

    @property
    def firmware_version(self):
        """Firmware version.

        Returns:
            str: version
        """
        return self.read('fw_version')

    @property
    def hardware_version(self):
        """Hardware version.

        Returns:
            str: version
        """
        return self.read('hw_version')

    def save(self):
        """Save all settings to non-volatile EEPROM."""
        self.write('save')

    @property
    def reference_modes(self):
        """Frequency reference modes.

        Returns:
            tuple: tuple of str of modes
        """
        return ('external', 'internal 27mhz', 'internal 10mhz')

    @property
    def reference_mode(self):
        """Get frequency reference mode.

        Returns:
            str: mode
        """
        return self.reference_modes[self.read('reference_mode')]

    @reference_mode.setter
    def reference_mode(self, value):
        """Set frequency reference mode.

        Args:
            value (str): mode
        """
        modes = self.reference_modes
        if not value in modes:
            raise ValueError('Expected str in set {}.'.format(modes))
        self.write('reference_mode', modes.index(value))

    @property
    def trigger_modes(self):
        """Trigger modes.

        Returns:
            tuple: tuple of str of modes
        """
        return (
            'disabled',
            'full frequency sweep',
            'single frequency step',
            'stop all',
            'rf enable',
            'remove interrupts',
            'reserved',
```

```
            'reserved',
            'am␣modulation',
            'fm␣modulation',
        )

    @property
    def trigger_mode(self):
        """Get trigger mode.

        Returns:
            str: mode
        """
        return self.trigger_modes[self.read('trig_function')]

    @trigger_mode.setter
    def trigger_mode(self, value):
        """Set trigger mode.

        Args:
            value (str): mode
        """
        modes = self.trigger_modes
        if not value in modes:
            raise ValueError('Expected␣str␣in␣set␣{}.'.format(modes))
        self.write('trig_function', modes.index(value))

    @property
    def temperature(self):
        """Temperature in Celsius.

        Returns:
            float: temperature
        """
        return self.read('temperature')

    @property
    def reference_frequency_range(self):
        """Reference frequency range in Hz.

        Returns:
            dict: frequency range in Hz
        """
        return {'start': 10.e6, 'stop': 100.e6, 'step': 1.e3}

    @property
    def reference_frequency(self):
        """Get reference frequency in Hz.

        Returns:
            float: frequency in Hz
        """
        return self.read('ref_frequency') * 1.e6

    @reference_frequency.setter
    def reference_frequency(self, value):
        """Set reference frequency in Hz.

        Args:
            value (float / int): frequency in Hz
        """
        if not isinstance(value, (float, int)):
            raise ValueError('Expected␣float␣or␣int.')
        f_range = self.reference_frequency_range
        if not f_range['start'] <= value <= f_range['stop']:
            raise ValueError('Expected␣float␣in␣range␣[{},␣{}]␣Hz.'.format(
                             f_range['start'], f_range['stop']))
        self.write('ref_frequency', value / 1.e6)

    @property
    def sweep_enable(self):
        """Get sweep continuously enable.
```

```
        Returns:
            bool: enable
        """
        return self.read('sweep_cont')

    @sweep_enable.setter
    def sweep_enable(self, value):
        """Set sweep continuously enable.

        Args:
            value (bool): enable
        """
        if not isinstance(value, bool):
            raise ValueError('Expected bool.')
        self.write('sweep_cont', value)

    @property
    def am_enable(self):
        """Get AM continuously enable.

        Returns:
            bool: enable
        """
        return self.read('am_cont')

    @am_enable.setter
    def am_enable(self, value):
        """Set AM continuously enable.

        Args:
            value (bool): enable
        """
        if not isinstance(value, bool):
            raise ValueError('Expected bool.')
        self.write('am_cont', value)

    @property
    def pulse_mod_enable(self):
        """Get pulse modulation continuously enable.

        Returns:
            bool: enable
        """
        return self.read('pulse_cont')

    @pulse_mod_enable.setter
    def pulse_mod_enable(self, value):
        """Set pulse modulation continuously enable.

        Args:
            value (bool): enable
        """
        if not isinstance(value, bool):
            raise ValueError('Expected bool.')
        self.write('pulse_cont', value)

    @property
    def dual_pulse_mod_enable(self):
        """Get dual pulse modulation enable.

        Returns:
            bool: enable
        """
        return self.read('dual_pulse_mod')

    @dual_pulse_mod_enable.setter
    def dual_pulse_mod_enable(self, value):
        """Set dual pulse modulation enable.

        Args:
```

```
            value (bool): enable
        """
        if not isinstance(value, bool):
            raise ValueError('Expected␣bool.')
        self.write('dual_pulse_mod', value)

    @property
    def fm_enable(self):
        """Get FM continuously enable.

        Returns:
            bool: enable
        """
        return self.read('fm_cont')

    @fm_enable.setter
    def fm_enable(self, value):
        """Set FM continuously enable.

        Args:
            value (bool): enable
        """
        if not isinstance(value, bool):
            raise ValueError('Expected␣bool.')
        self.write('fm_cont', value)
```

A.2.9. Old PySide windowed application

```
'''https://zetcode.com/gui/pysidetutorial/widgets/ Useful tutorial'''
import sys
from time import sleep
from threading import Thread
import numpy as np

from PySide6.QtUiTools import QUiLoader
from PySide6.QtWidgets import QApplication,  QMainWindow
from PySide6.QtCore import QFile, QIODevice, Slot

from project.client.application.api import sweep_acquire, receive_data, get_magnitude
from project.client.ui.app.draw_graph import Plotting
from project.client.ui.windows.main import Ui_MainWindow
from project.client.generator.wft import SynthHDController
from project.client.connection.tcp_client import TCPClient

class GUI:
    def __init__(self):
        self.sy = SynthHDController() #also connects with generator

        #GUI setup
        app = QApplication(sys.argv)  #Object that manages the GUI 'applications control flow
             and main settings
        mw = QMainWindow()
        self.window = Ui_MainWindow()
        self.window.setupUi(mw)

        #Change GUI appearance
        self.window.pushButton_startstop.setCheckable(True)

        #Connect GUI buttons
        self.window.doubleSpinBox_freqstart.valueChanged.connect(self.start_freq_spinbox)  #
            This function connects the output from this button
        self.window.doubleSpinBox_freqstop.valueChanged.connect(self.stop_freq_spinbox)
        self.window.doubleSpinBox_frequencystep.valueChanged.connect(self.step_freq_spinbox)
        self.window.doubleSpinBox_timestep.valueChanged.connect(self.step_time_spinbox)
        self.window.checkBox_steptrigger.clicked.connect(self.step_button)
        self.window.pushButton_startstop.clicked.connect(self.enable_disable)

        mw.show()

        sys.exit(app.exec())
```

```

    @Slot()
    def start_freq_spinbox(self):
        self.start_freq = self.window.doubleSpinBox_freqstart.value()
        print(self.start_freq)

    @Slot()
    def stop_freq_spinbox(self):
        self.stop_freq = self.window.doubleSpinBox_freqstop.value()
        print(self.stop_freq)

    @Slot()
    def step_freq_spinbox(self):
        self.step_freq = self.window.doubleSpinBox_frequencystep.value()
        print(self.step_freq)

    @Slot()
    def step_time_spinbox(self):
        self.step_time = self.window.doubleSpinBox_timestep.value()
        print(self.step_time)

    @Slot()                                  #dont know the function of this yet
    def step_button(self):
        self.trigger_per_step = self.window.checkBox_steptrigger.isChecked()
        print(self.trigger_per_step)

    @Slot()
    def enable_disable(self):
        #Initialise values
        self.trigger_per_step = self.window.checkBox_steptrigger.isChecked() #True if we want step triggering
        self.start_freq = self.window.doubleSpinBox_freqstart.value()
        self.stop_freq = self.window.doubleSpinBox_freqstop.value()
        self.step_freq = self.window.doubleSpinBox_frequencystep.value()
        self.step_time = self.window.doubleSpinBox_timestep.value()

        #Turn on or off
        if self.window.pushButton_startstop.isChecked():
            x,y = sweep_acquire(self.trigger_per_step, self.start_freq, self.stop_freq, self.step_freq, self.step_time, self.sy)[:2] #Does not do live updating of x and y yet
            plot = Plotting.plot_x_y(self.window.graphicsView, x, y)
            t = Thread(target=GUI.realtime_plot, args=(self, plot, self.start_freq, self.stop_freq, self.step_freq)).start() #Thread to do live plotting
        else:
            self.sy.turn_off((0,1))

    def realtime_plot(self, curve, freq_low, freq_high, freqstep) -> None:
        with TCPClient(host="10.0.0.11",port=2024) as tcp:
            while self.window.pushButton_startstop.isChecked():
                num_frequencies = int( (freq_high-freq_low) // freqstep )
                x = np.linspace(freq_low, freq_high, int(num_frequencies))

                data = receive_data(num_frequencies, tcp.request_data)
                magnitude = get_magnitude(data)
                rel_magnitude = magnitude[:,0]/magnitude[:,1]

                curve.setData(rel_magnitude)  # set y array
                # curve.setPos(i, 0)  # set x to 0? why?
                # curve.setXrange(0, 20)
                QApplication.processEvents()  # update plot
                sleep(0.1)

    '''Needed?'''
    def __del__(self) -> None:
        print(self.window.pushButton_startstop.isChecked())
        if self.window.pushButton_startstop.isChecked():
            self.window.pushButton_startstop.click()
        print(self.window.pushButton_startstop.isChecked())
```

A.2.10. Old PySide graphs

```python
# -*- coding: utf-8 -*-

################################################################################
## Form generated from reading UI file 'graph.ui'
##
## Created by: Qt User Interface Compiler version 6.7.0
##
## WARNING! All changes made in this file will be lost when recompiling UI file!
################################################################################

from PySide6.QtCore import (
    QCoreApplication, QDate, QDateTime, QLocale, QMetaObject, QObject, QPoint, QRect, QSize, QTime, QUrl, Qt
)
from PySide6.QtGui import (
    QBrush, QColor, QConicalGradient, QCursor, QFont, QFontDatabase, QGradient, QIcon, QImage, QKeySequence, QLinearGradient,
    QPainter, QPalette, QPixmap, QRadialGradient, QTransform
)
from PySide6.QtWidgets import (
    QAbstractSpinBox, QApplication, QDoubleSpinBox, QGroupBox, QLabel, QMainWindow, QMenuBar, QPushButton, QSizePolicy,
    QSlider, QStatusBar, QWidget
)

from pyqtgraph import PlotWidget


class Ui_MainWindow(object):

    def setupUi(self, MainWindow):
        if not MainWindow.objectName():
            MainWindow.setObjectName(u"MainWindow")
        MainWindow.resize(800, 600)
        MainWindow.setAcceptDrops(False)
        MainWindow.setWindowTitle(u"OpenVQA")
        MainWindow.setLocale(QLocale(QLocale.English, QLocale.UnitedKingdom))
        self.centralwidget = QWidget(MainWindow)
        self.centralwidget.setObjectName(u"centralwidget")
        self.groupBox_freq_sweep = QGroupBox(self.centralwidget)
        self.groupBox_freq_sweep.setObjectName(u"groupBox_freq_sweep")
        self.groupBox_freq_sweep.setGeometry(QRect(530, 320, 241, 221))
        self.horizontalSlider_frequencystep = QSlider(self.groupBox_freq_sweep)
        self.horizontalSlider_frequencystep.setObjectName(u"horizontalSlider_frequencystep")
        self.horizontalSlider_frequencystep.setGeometry(QRect(10, 110, 211, 22))
        self.horizontalSlider_frequencystep.setMinimum(1)
        self.horizontalSlider_frequencystep.setMaximum(200000)
        self.horizontalSlider_frequencystep.setPageStep(1000)
        self.horizontalSlider_frequencystep.setOrientation(Qt.Orientation.Horizontal)
        self.horizontalSlider_frequencystep.setInvertedAppearance(False)
        self.horizontalSlider_frequencystep.setTickPosition(QSlider.TickPosition.TicksBelow)
        self.label_frequencystep = QLabel(self.groupBox_freq_sweep)
        self.label_frequencystep.setObjectName(u"label_frequencystep")
        self.label_frequencystep.setGeometry(QRect(10, 80, 91, 21))
        self.label_frequencystep.setText(u"<html><head/><body><p>Frequency step: </p></body></html>")
        self.doubleSpinBox_freqstart = QDoubleSpinBox(self.groupBox_freq_sweep)
        self.doubleSpinBox_freqstart.setObjectName(u"doubleSpinBox_freqstart")
        self.doubleSpinBox_freqstart.setGeometry(QRect(10, 30, 91, 31))
        self.doubleSpinBox_freqstop = QDoubleSpinBox(self.groupBox_freq_sweep)
        self.doubleSpinBox_freqstop.setObjectName(u"doubleSpinBox_freqstop")
        self.doubleSpinBox_freqstop.setGeometry(QRect(150, 30, 91, 31))
        #if QT_CONFIG(tooltip)
        self.doubleSpinBox_freqstop.setToolTip(u"Stop frequency in MHz")
        #endif // QT_CONFIG(tooltip)
        self.doubleSpinBox_freqstop.setProperty("showGroupSeparator", False)
        self.doubleSpinBox_freqstop.setSuffix(u" MHz")
        self.doubleSpinBox_freqstop.setDecimals(0)
        self.doubleSpinBox_freqstop.setMinimum(300.000000000000000)
        self.doubleSpinBox_freqstop.setMaximum(14000.000000000000000)
```

```
        self.doubleSpinBox_freqstop.setSingleStep(0.100000000000000)
        self.doubleSpinBox_freqstop.setStepType(QAbstractSpinBox.StepType.DefaultStepType)
        self.doubleSpinBox_freqstop.setValue(6000.000000000000000)
        self.label_timestep = QLabel(self.groupBox_freq_sweep)
        self.label_timestep.setObjectName(u"label_timestep")
        self.label_timestep.setGeometry(QRect(10, 150, 91, 21))
        self.label_timestep.setText(u"<html><head/><body><p>Time step: </p></body></html>")
        self.horizontalSlider_timestep = QSlider(self.groupBox_freq_sweep)
        self.horizontalSlider_timestep.setObjectName(u"horizontalSlider_timestep")
        self.horizontalSlider_timestep.setGeometry(QRect(10, 180, 211, 22))
        self.horizontalSlider_timestep.setMinimum(1)
        self.horizontalSlider_timestep.setMaximum(3000)
        self.horizontalSlider_timestep.setPageStep(100)
        self.horizontalSlider_timestep.setValue(1)
        self.horizontalSlider_timestep.setOrientation(Qt.Orientation.Horizontal)
        self.horizontalSlider_timestep.setInvertedAppearance(False)
        self.horizontalSlider_timestep.setTickPosition(QSlider.TickPosition.TicksBelow)
        self.pushButton_frequency_direction = QPushButton(self.groupBox_freq_sweep)
        self.pushButton_frequency_direction.setObjectName(u"pushButton_frequency_direction")
        self.pushButton_frequency_direction.setGeometry(QRect(110, 30, 31, 31))
        self.pushButton_frequency_direction.setText(u"&rarr;")
        self.groupBox_magnitude_plot = QGroupBox(self.centralwidget)
        self.groupBox_magnitude_plot.setObjectName(u"groupBox_magnitude_plot")
        self.groupBox_magnitude_plot.setGeometry(QRect(10, 10, 481, 351))
        self.graphicsView_magnitude_plot = PlotWidget(self.groupBox_magnitude_plot)
        self.graphicsView_magnitude_plot.setObjectName(u"graphicsView_magnitude_plot")
        self.graphicsView_magnitude_plot.setGeometry(QRect(10, 20, 461, 321))
        MainWindow.setCentralWidget(self.centralwidget)
        self.menubar = QMenuBar(MainWindow)
        self.menubar.setObjectName(u"menubar")
        self.menubar.setGeometry(QRect(0, 0, 800, 22))
        MainWindow.setMenuBar(self.menubar)
        self.statusbar = QStatusBar(MainWindow)
        self.statusbar.setObjectName(u"statusbar")
        MainWindow.setStatusBar(self.statusbar)

        self.retranslateUi(MainWindow)

        QMetaObject.connectSlotsByName(MainWindow)

    # setupUi

    def retranslateUi(self, MainWindow):
        self.groupBox_freq_sweep.setTitle(QCoreApplication.translate("MainWindow", u"for-quickly-copy-pasting-widgets", None))
        self.doubleSpinBox_freqstart.setSuffix(QCoreApplication.translate("MainWindow", u" MHz", None))
        self.groupBox_magnitude_plot.setTitle(QCoreApplication.translate("MainWindow", u"Magnitude plot", None))
        pass

    # retranslateUi
```

A.2.11. Old PySide windowed application

```
# -*- coding: utf-8 -*-

################################################################################
## Form generated from reading UI file 'main.ui'
##
## Created by: Qt User Interface Compiler version 6.7.0
##
## WARNING! All changes made in this file will be lost when recompiling UI file!
################################################################################

from PySide6.QtCore import (
    QCoreApplication, QDate, QDateTime, QLocale, QMetaObject, QObject, QPoint, QRect, QSize,
        QTime, QUrl, Qt
)
from PySide6.QtGui import (
    QAction, QBrush, QColor, QConicalGradient, QCursor, QFont, QFontDatabase, QGradient,
```

```
        QIcon, QImage, QKeySequence,
    QLinearGradient, QPainter, QPalette, QPixmap, QRadialGradient, QTransform
)
from PySide6.QtWidgets import (
    QAbstractSpinBox, QApplication, QCheckBox, QDoubleSpinBox, QGroupBox, QLabel, QMainWindow, QMenu, QMenuBar, QPushButton,
    QSizePolicy, QStatusBar, QWidget
)

from pyqtgraph import PlotWidget


class Ui_MainWindow(object):

    def setupUi(self, MainWindow):
        if not MainWindow.objectName():
            MainWindow.setObjectName(u"MainWindow")
        MainWindow.resize(790, 420)
        MainWindow.setAcceptDrops(False)
        MainWindow.setWindowTitle(u"OpenVQA")
        MainWindow.setLocale(QLocale(QLocale.English, QLocale.UnitedKingdom))
        self.centralwidget = QWidget(MainWindow)
        self.centralwidget.setObjectName(u"centralwidget")
        self.groupBox = QGroupBox(self.centralwidget)
        self.groupBox.setObjectName(u"groupBox")
        self.groupBox.setGeometry(QRect(20, 0, 271, 351))
        self.label_frequencystep = QLabel(self.groupBox)
        self.label_frequencystep.setObjectName(u"label_frequencystep")
        self.label_frequencystep.setGeometry(QRect(20, 120, 101, 21))
        self.label_frequencystep.setText(u"<html><head/><body><p>Frequency␣step</p></body></html>")
        self.doubleSpinBox_freqstart = QDoubleSpinBox(self.groupBox)
        self.doubleSpinBox_freqstart.setObjectName(u"doubleSpinBox_freqstart")
        self.doubleSpinBox_freqstart.setGeometry(QRect(10, 50, 121, 31))
        self.doubleSpinBox_freqstart.setFrame(True)
        self.doubleSpinBox_freqstart.setDecimals(3)
        self.doubleSpinBox_freqstart.setMinimum(300.000000000000000)
        self.doubleSpinBox_freqstart.setMaximum(14000.000000000000000)
        self.doubleSpinBox_freqstart.setSingleStep(100.000000000000000)
        self.doubleSpinBox_freqstart.setValue(4000.000000000000000)
        self.doubleSpinBox_freqstop = QDoubleSpinBox(self.groupBox)
        self.doubleSpinBox_freqstop.setObjectName(u"doubleSpinBox_freqstop")
        self.doubleSpinBox_freqstop.setGeometry(QRect(140, 50, 121, 31))
        #if QT_CONFIG(tooltip)
        self.doubleSpinBox_freqstop.setToolTip(u"Stop␣frequency␣in␣MHz")
        #endif // QT_CONFIG(tooltip)
        self.doubleSpinBox_freqstop.setProperty("showGroupSeparator", False)
        self.doubleSpinBox_freqstop.setSuffix(u"␣MHz")
        self.doubleSpinBox_freqstop.setDecimals(3)
        self.doubleSpinBox_freqstop.setMinimum(300.000000000000000)
        self.doubleSpinBox_freqstop.setMaximum(14000.000000000000000)
        self.doubleSpinBox_freqstop.setSingleStep(100.000000000000000)
        self.doubleSpinBox_freqstop.setStepType(QAbstractSpinBox.StepType.DefaultStepType)
        self.doubleSpinBox_freqstop.setValue(6000.000000000000000)
        self.label_timestep = QLabel(self.groupBox)
        self.label_timestep.setObjectName(u"label_timestep")
        self.label_timestep.setGeometry(QRect(170, 120, 61, 21))
        self.label_timestep.setText(u"<html><head/><body><p>Time␣step</p></body></html>")
        self.label_freqstart = QLabel(self.groupBox)
        self.label_freqstart.setObjectName(u"label_freqstart")
        self.label_freqstart.setGeometry(QRect(20, 30, 101, 16))
        self.checkBox_steptrigger = QCheckBox(self.groupBox)
        self.checkBox_steptrigger.setObjectName(u"checkBox_steptrigger")
        self.checkBox_steptrigger.setGeometry(QRect(10, 90, 111, 20))
        self.label_freqstop = QLabel(self.groupBox)
        self.label_freqstop.setObjectName(u"label_freqstop")
        self.label_freqstop.setGeometry(QRect(150, 30, 101, 16))
        self.checkBox_reversedirection = QCheckBox(self.groupBox)
        self.checkBox_reversedirection.setObjectName(u"checkBox_reversedirection")
        self.checkBox_reversedirection.setGeometry(QRect(140, 90, 131, 20))
        self.doubleSpinBox_frequencystep = QDoubleSpinBox(self.groupBox)
```

```
        self.doubleSpinBox_frequencystep.setObjectName(u"doubleSpinBox_frequencystep")
        self.doubleSpinBox_frequencystep.setGeometry(QRect(10, 140, 121, 31))
        self.doubleSpinBox_frequencystep.setDecimals(3)
        self.doubleSpinBox_frequencystep.setMaximum(14000.000000000000000)
        self.doubleSpinBox_frequencystep.setValue(100.000000000000000)
        self.doubleSpinBox_timestep = QDoubleSpinBox(self.groupBox)
        self.doubleSpinBox_timestep.setObjectName(u"doubleSpinBox_timestep")
        self.doubleSpinBox_timestep.setGeometry(QRect(140, 140, 121, 31))
        self.doubleSpinBox_timestep.setDecimals(1)
        self.doubleSpinBox_timestep.setMinimum(0.300000000000000)
        self.doubleSpinBox_timestep.setMaximum(100000.000000000000000)
        self.doubleSpinBox_timestep.setValue(100.000000000000000)
        self.pushButton_startstop = QPushButton(self.groupBox)
        self.pushButton_startstop.setObjectName(u"pushButton_startstop")
        self.pushButton_startstop.setGeometry(QRect(90, 310, 100, 32))
        self.pushButton_startstop.setStyleSheet(u"")
        self.groupBox_magnitude_plot = QGroupBox(self.centralwidget)
        self.groupBox_magnitude_plot.setObjectName(u"groupBox_magnitude_plot")
        self.groupBox_magnitude_plot.setGeometry(QRect(290, 0, 481, 351))
        self.graphicsView = PlotWidget(self.groupBox_magnitude_plot)
        self.graphicsView.setObjectName(u"graphicsView")
        self.graphicsView.setGeometry(QRect(10, 30, 461, 311))
        MainWindow.setCentralWidget(self.centralwidget)
        self.menubar = QMenuBar(MainWindow)
        self.menubar.setObjectName(u"menubar")
        self.menubar.setGeometry(QRect(0, 0, 790, 24))
        self.menuSweep_menu = QMenu(self.menubar)
        self.menuSweep_menu.setObjectName(u"menuSweep_menu")
        MainWindow.setMenuBar(self.menubar)
        self.statusbar = QStatusBar(MainWindow)
        self.statusbar.setObjectName(u"statusbar")
        MainWindow.setStatusBar(self.statusbar)

        self.menubar.addAction(self.menuSweep_menu.menuAction())

        self.retranslateUi(MainWindow)

        QMetaObject.connectSlotsByName(MainWindow)

    # setupUi

    def retranslateUi(self, MainWindow):
        self.groupBox.setTitle(QCoreApplication.translate("MainWindow", u"Frequency sweep", None))
        self.doubleSpinBox_freqstart.setSuffix(QCoreApplication.translate("MainWindow", u" MHz", None))
        self.label_freqstart.setText(QCoreApplication.translate("MainWindow", u"Start frequency", None))
        self.checkBox_steptrigger.setText(QCoreApplication.translate("MainWindow", u"Step triggering", None))
        self.label_freqstop.setText(QCoreApplication.translate("MainWindow", u"Stop frequency", None))
        self.checkBox_reversedirection.setText(QCoreApplication.translate("MainWindow", u"Reverse direction", None))
        self.doubleSpinBox_frequencystep.setSuffix(QCoreApplication.translate("MainWindow", u" MHz", None))
        self.doubleSpinBox_timestep.setSuffix(QCoreApplication.translate("MainWindow", u" ms", None))
        self.pushButton_startstop.setText(QCoreApplication.translate("MainWindow", u"Turn on", None))
        self.groupBox_magnitude_plot.setTitle(QCoreApplication.translate("MainWindow", u"Magnitude plot", None))
        self.menuSweep_menu.setTitle(QCoreApplication.translate("MainWindow", u"Sweep menu", None))
        pass

    # retranslateUi
```