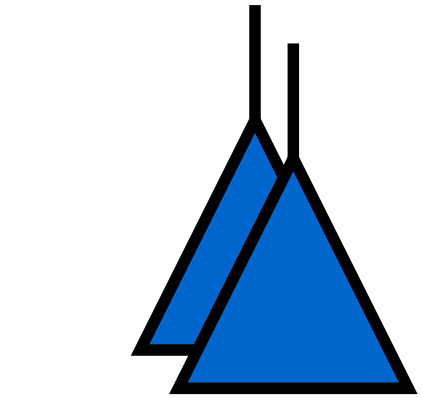


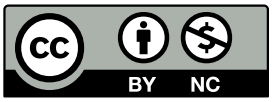
GPR-20

Software Guide

Grupo de Desminado Humanitario
Departamento de Ingeniería Eléctrica y Electrónica
Facultad de Ingeniería
Universidad de Los Andes
Bogotá D.C., Colombia

GPR-20 Software Guide

Proprietary Notice



Contact Information

Please communicate any comments to:

Name: Grupo de Desminado Humanitario

Email: desminadohumanitario@uniandes.edu.co

Dependency: Departamento de Ingeniería Eléctrica y Electrónica

University: Universidad de Los Andes

City: Bogotá D.C.

Country: Colombia

Versioning

Date	**Comments**
March 24th 2021	First major release
March 2nd 2022	Added hyperlinks to third-party software. Added links to software repositories. Extended information on software utilities.
March 25th 2022	Modified UML diagrams to incorporate the latest features from software.

Contents

1 Introduction

The GPR-20 robot requires a complete software suite in order to acquire data and move to different coordinates within the sampling area. The robot acquires data from the subsurface using a Ground Penetrating Radar (GPR). The GPR consists of a Vector Network Analyzer (VNA) and two antennae. The GPR is moved within the sampling area using stepper motors. The actions of both acquiring data and moving the GPR are executed using commands from the robot software suite. The robot also requires an interface for the users to command it.

When operating, the robot will execute a survey: sampling the ground on different coordinates. There are three coordinates that are considered: X coordinate, Y coordinate and Z rotation. The X and Y coordinates refer to the location of the GPR in each axis. On the other hand, the Z rotation refer to the rotation of the antennae. This rotation is included to sample different electromagnetic polarizations. The acquired data is saved on a persistent storage when each point is sampled. Additional operations carried out during a survey consists of providing feedback to the user and acquiring the height from the antennae to the ground. The height is acquired from an infrared (IR) sensor located in the GPR.

The software suite is responsible of executing the commands required to perform the aforementioned actions. Many approaches can be used to develop the software in which one stands out: a software suite based on the Robotic Operating System (ROS). ROS is an open-source framework that provides libraries and tools to develop robotic applications. Using ROS to develop the software allowed to have modular packages that are then integrated into a graph-like architecture. Each package is part of a peer-to-peer network of processes that are loosely coupled using the ROS communication infrastructure. A detailed explanation of the ROS framework is addressed in this document.

Each of the GPR-20 actions has different requirements that constrain the device in which the software must be executed. Some actions such as acquiring and storing the VNA data require a considerable amount of computing power. Other actions require driving physical signals within a defined timing constrains. These requirements led to distributing the software on two devices: a Raspberry Pi 4 single-board computer and a personal computer. The Raspberry Pi 4 is used as the on-board computer i.e. it is physically located in the robot while the personal computer is connected to the robot via a network interface.

This document presents the GPR-20 software suite. It addresses a detailed explanation of the software execution environment, the software suite architecture, and a detailed explanation of each of the software packages. The document is organized as follows: first, it explains the used operating system. Then, the ROS framework is introduced in detail. The third section of the documents corresponds to a general overview on the software architecture. Each of the software packages is presented on the fourth section. Finally, a section is devoted to keep a record on how to extend and improve the software suite.

2 Operating System: Canonical Ubuntu

The operating system (OS) in which the GPR-20 software runs is Canonical Ubuntu (link). Ubuntu is a Debian-based Linux distribution that is developed and maintained by Canonical. It is one of the most popular Linux distributions and it is currently used on personal computers, servers and internet-of-things (IoT) devices. Releases for Ubuntu occurs every six months with long-term support releases happening every two years. Because of its popularity, Ubuntu provides support and easy-to-install mechanisms for different libraries and utilities. Within the context of the GPR-20 robot, this means that it simplifies the management of the required libraries and framework thus improving the development process.

The OS is used on both the on-board computer and the personal computer. The installation on the personal computer is straightforward and does not require to customize the operative system to fit any constrain. This is not the case for the Raspberry Pi 4. The on-board computer requires the operating system to minimize the RAM usage, CPU usage, and disk usage. This is addressed by tailoring Ubuntu to fit those requirements. The first requirement is the architecture of the Raspberry Pi BCM58712 SoC: ARMv8-A. Using a different architecture from x86 might arise compatibility issues on the OS itself and the libraries. However, Ubuntu is available on ARMv8-A and most of its libraries work on both architectures. In order to prevent issues from compatibility, each library is checked to ensure that it works on ARMv8-A.

Another requirement that must be addressed is disk space. The OS should minimize the disk usage since the main storage of the Raspberry Pi is a 16GB SD card. In order to minimize disk usage, the server version of Ubuntu is used. The server version of Ubuntu does not include third-party software that is included in the desktop version of the OS thus reducing disk space usage. Using the server version also implies that no desktop environment is installed. A minimal desktop is installed since it is required to display the user interface of the robot. The installed desktop environment is Ubuntu MATE.

To ensure that the Raspberry Pi resources are used at their best, the Ubuntu 20.04 LTS version is used. It must be noted that the 18.04 LTS version was tested on the Raspberry Pi 4 but performance of the 20.04 LTS version was better. It also must be noted that Ubuntu is not the official OS for the Raspberry Pi but it was used since it simplifies the software development and improves compatibility with the personal computer as it runs Ubuntu.

3 Framework: Robotic Operating System

The main used framework is the Robotic Operating System (ROS, link). Contrary to what its name suggest, the framework is not an operating system but a set of tools and libraries to develop robotic applications. ROS software is organized in *packages*. Packages are the most atomic build and release item from ROS i.e. it is the most granular element that can be built and released. Packages contain ROS run-time processes that are defined as *nodes* under ROS terminology. Nodes are the elements that perform computation thus providing the required capabilities from software. A package can contain multiple nodes depending on the application needs.

Besides from organizing the software, ROS also defines concepts for communication among nodes. The basic element for ROS communication are *messages*. Messages are data structures that define typed fields. The data types that can be used depend on primitives that include numeric, Boolean, string and time types. The primitives can be arranged as arrays and nested structures if necessary. A message is defined in a *msg* file and may contain an arbitrary amount of data fields.

The most simple communication mechanism provided by ROS consists of *topics*. Topics are a transport system with a name and a message type in which any node can *publish* or *subscribe*. Publishing is the act of a topic sending out a message through the topic. On the other hand, subscribing refers to a topic retrieving data from the topic. Another communication mechanism provided by ROS are *services*. Services provide a request/reply communication between two agents. A service is defined by two messages that have the request and reply definition in a *.service* file. Services are similar to remote procedure calls and are blocking: the sender must wait until receiving the response.

A third communication mechanism are *actions*. Actions are not a primitive communication mechanism but an extension to ROS. Actions are similar to services in the sense that they provide a request/reply communication. Actions differ from services as they provide support for non-blocking calls. In other words, actions do not require the sender to wait for the reply as services do. An action is specified by three messages: *goal*, *feedback* and *result*. The goal defines the input data required to execute the procedure. The feedback is used to provide the sender with the progress of a goal. Finally, the result defines the output data of the procedure. The three required messages of an action are defined in a single *.action* file.

Applications using ROS can be represented as graph. Nodes in the graph are the processes i.e. the ROS nodes in which computation takes place. The graph edges are the communication mechanisms provided by ROS: topics, services and actions. This architecture provide benefits for robotics applications such as decoupling the processes to build complex systems. On the other hand, the decoupled processes have a more specific purpose thus facilitating its development. In general terms, a robotic application can be developed easily when using ROS, and this is the reason behind ROS popularity among the robot community.

4 Software Architecture Overview

The software suite for the GPR-20 robot is developed under the ROS framework thus generating a graph-like architecture. Nevertheless, the graph is not enough to describe the architecture. Specific details of each element must be addressed in order to correctly describe and justify the architecture. This section is devoted to describe the requirements of the software suite and how it is organized to fulfill them. First, the general requirements of the software are specified. Then, the requirements are linked to a ROS package and organized in a graph architecture. Finally, implementation details are addressed.

4.1 General Requirements

The GPR-20 robot purpose is to acquire Ground Penetrating Radar (GPR) data from deactivated mine-fields. In order to acquire the data, the robot must be able to get the data from a Vector Network Analyzer (VNA) and save it in a persistent storage. The VNA and the antennae must be moved within a sampling area to effectively acquire relevant data. The height between a fixed point of the antennae and the ground is also acquired and stored. Finally, an user interface must be provided in order to define the data-acquisition parameters. A summary of the global requirements for the GPR-20 software suite is presented on table 1.

Number	Requirement
1	Communicate with the VNA to acquire data.
2	Save the VNA data in a persistent storage.
3	Move the GPR within the sampling area
4	Acquiring height data.
5	Save height data in a persistent storage.
6	Provide an user interface to set the data-acquisition parameters.

Table 1: Global requirements for the GPR-20 software suite.

Table 1 presents a summary of the software requirements and associates them with a number. Requirement number one (1) defines that data must be retrieved from the VNA. In order to communicate with the VNA, the software can make use of the VXI-11 protocol. The VXI-11 protocol is supported by the VNA and since it relies on the TCP/IP protocol, it only requires a network interface. The user manual of the VNA defines the messages to configure the device and retrieve the data.

Saving data is specified in requirements two (2) and five (5) in which respectively, VNA and height data is saved in persistent storage. Going beyond the definition of persistent storage, two options arise: storage devices and cloud storage. However, since the robot might operate in a place in which no Internet is available,

cloud storage is discarded as a first-hand approach. Cloud storage may be used as backup and/or to share the data with the community. Storage devices include hard-drive drives (HDDs), solid-state drives (SSDs), SD cards and USB flash drives. Since the software will execute on either a personal computer (HDD and/or SSD) or the Raspberry Pi 4 (SD card), data will be stored in one of those devices. Since acquired data size can be in the order of gigabytes (GBs), either the HDDs or the SSDs will be preferred.

Requirement number three (3) indicates that the GPR will move within a sampling area. The movement of the GPR is handled using five stepper motors: three for moving along the X and Y axes and two for rotating the antennae. The motors are controlled using *drivers*, that in turn are controlled using a two-signal protocol. The protocol signals are required to have specific timing requirements in order for the motors to work. These signals are intended to be driven from the Raspberry Pi 4 single-board computer. Endstop sensors are used in each axis as a way to tell the position of an axis on startup.

Requirement number four (4) states that height data must be also acquired. The height data is acquired using an infrared (IR) sensor located in the GPR mount. The IR sensor output is a signal in which its voltage corresponds to a specific height. Since neither the Raspberry Pi 4 or the personal computer has an analog/digital converter (ADC), an external one is required. The ADC communicates using the SPI protocol thus the Raspberry Pi 4 is the best option to acquire the data.

Finally, requirement number six (6) indicates that an user interface should be provided. An user interface can be a command-line interface (CLI) or a graphical user interface (GUI). Since using a CLI requires the user to know how to use a command line in first place, it is discarded in favor of a GUI. The GUI should be designed for an inexperienced user to interact with the robot.

4.2 Architecture Definition

A software architecture can be defined keeping into account the requirements from table 1. The first defined set of elements are the processing elements of the architecture i.e. the ROS nodes. The ROS nodes are defined based on the general software requirements and constrains generated by the implementation itself. Table 2 presents the processing nodes that will be used in the architecture with a brief description of their purpose.

The `Axis Driver` node aims to fulfill requirement number three (3) by handling the axes movement. The

Name	Purpose
Axis Driver	Handles the movement of a single axis of the robot.
Height Sensor	Samples data from the height sensor ADC through the SPI protocol.
VNA Acquisition	Manages the communication with the VNA device. It can configure and retrieve data from the device.
Data Processing	Stores the VNA and height sensor data in a persistent storage.
User Interface	Provides a graphical user interface from which the user can interact with the robot.
Main Control	Provides synchronization mechanisms within the robot software.

Table 2: Defined nodes for the GPR-20 software suite.

`Axis Driver` node must be able to manage the movement of linear and rotational axes. Depending on the movement type, the node should sample the endstop sensor. The same `AxisDriver` node must be used for the three axes. The `Height Sensor` node fulfills requirement number four (4). It retrieves data from the ADC using the SPI protocol. The node must retrieve data from the ADC and then calculate its distance value. The `VNA Acquisition` node fulfills requirement number one (1). It is able to configure the device and acquire data via the VXI-11 protocol. The `Data Processing` node fulfills requirement number two (2) and five (5). The node processes and saves the data in the persistent storage. The processing consists of arranging the data in a specific format such as H5 or CSV. The `User Interface` fulfills requirement number six (6). This node provides a graphical user interface for an user to input the robot parameters to acquire data. Finally, the `Main Control` serves to provide synchronization mechanisms within the robot software. This node is in charge of executing specific tasks when they must take place. For example, the VNA data should be retrieved once the GPR is located in its target coordinates and not during the axes movement.

Communication mechanisms need to be addressed once nodes tasks' are defined. In other words, the communication mechanisms for each node must be defined based on its functionality and the ROS framework. The first definition consists of defining the communication requirements for each node. This corresponds to defining what data or request a node needs to either receive or provide. Table 3 presents the communication requirements for each node.

Node	Requirements
Axis Driver	*Target coordinate:* the node must receive a target coordinate and move to it.
	Current Coordinate: the node must provide the value of its current coordinate.
Continues on next page.	

Node	Requirements
	Homing: the node must receive a command to execute the homing sequence.
Height Sensor	*Height value:* the node must provide the height value.
VNA Acquisition	*Connection:* the node must receive the parameters to connect to the VNA.
	Configuration: the node must receive the VNA configuration parameters.
	Data: the node must provide the VNA data.
	Frequency: the node must provide the VNA frequencies array
Data Processing	*VNA data:* the node must receive the VNA data.
	Height data: the node must receive the height data
User Interface	*Coordinate limits:* the node must provide the coordinate limits for acquiring data.
	Step values: the node must provide the step values for each linear axis.
	VNA Connection: the node must provide the connection parameters for the VNA.
	VNA Configuration: the node must provide the VNA configuration parameters.
	Current coordinates: the node must receive the current coordinates of the axes.
	Height: the node must receive the current height of the antennae relative to the ground.
Main Control	*Coordinate limits:* the node must receive the coordinate limits for each axis.
	Step values: the node must receive the step values for each linear axis
	Target coordinates: the node must provide the target coordinates for each axis.
	Acquire VNA data: the node must request the VNA data acquisition.
	Acquire height data: the node must request the height data acquisition.

Table 3: Communication requirements for each node.

The second definition the communication mechanism type. The communication mechanism can be either a topic, a service, or an action. A topic will be used when a continuous stream of data is required. Both the service and the action will be used when a request to execute a procedure is required. However, they will defer based on the procedure execution time and blocking requirements. Since services are blocking, they will be used when short execution time is expected. Otherwise, a long-duration service could result in a non-optimal timing of the robot. Actions will be used on long-duration tasks as they are non blocking and allow different procedures to take place simultaneously. Table 4 presents the communication requirements

with their type.

Node	Requirement	Type
Axis Driver	Target coordinate	*Action:* in order to move more than one axis at the time. The target coordinate for an axis works as a request/response but using a service could result in moving a single axis at the time. That is the reason for the selection of an action.
	Current coordinate	*Topic:* in order to continuously provide information on the axis current coordinate.
	Homing	*Service:* in order to ensure that the axis has executed the homing sequence. Using an action is possible but would require an additional effort that does not translate into relevant benefits. Using a service implies that the homing sequences for the linear axes will be executed one at the time. This does not increment the robot timing as the homing sequence only take place once.
Height Sensor	Height Value	*Topic:* to provide real-time feedback on the antennae height.
	Height Value	*Service:* to provide a synchronous measurement on the antennae height.
VNA Acquisition	Connection	*Service:* as it only transfers information between two nodes. The node will only store the values thus no major execution time is expected.
	Configuration	*Service:* as it only transfers information between nodes. The service execution is to store the configuration parameters so no major execution time is expected.
	Data	*Service:* the node will ask the VNA device for the data. This action takes between one (1) and two (2) seconds. Nevertheless, since data has to be taken in specific coordinates, the service ensures that the execution is blocked until the response is sent.
	Frequency	*Service:* the node will retrieve the frequencies vector from the VNA. This procedure is expected to take place once.
Data Processing	VNA Data	*Service:* the data will be retrieved by sending a service request. The service is used to ensure that the data acquisition is synchronized.
	Height Data	*Service:* a service is used to ensure that the data acquisition is synchronized.
User Interface	Coordinate limits	*Service:* the user inputs the data and then it is sent as a request.
	Step values	*Service:* similarly to the coordinate limits, the data is sent as a request.
	VNA Connection	*Service:* since it is sent once and requires synchronization.
	VNA Configuration	*Service:* requires synchronization and is also sent once.
	Current coordinates	*Topic:* since its data is used to provide a real-time feedback to the user.
	Height	*Topic:* it is also used to provide feedback to the user.
Main Control	Coordinate limits	*Service:* the data is received from a service only once.
	Step values	*Service:* also received only once.

Continues on next page.

Node	Requirement	Type
	Target Coordinates	*Action:* it must be an action in order to move the axes at the same time. Using a service would imply that only one axis moves at the time so the measurement will take longer.
	Acquire VNA data	*Service:* used because of the synchronization requirements of acquiring data.
	Acquire height data	*Service:* also requires synchronization to ensure that data is acquired appropriately.

Table 4: Communication requirements for each node with their corresponding type.

As both the nodes and the communication requirements of them are specified, it is possible to implement them in code. It must be noted that the aforementioned definitions can be extended and/or changed depending on situations that arise during the implementation. Changes can arise from unforeseen requirements, library dependent considerations and optimizations. However, it is expected that the possible situations are minimized by defining the architecture beforehand.

4.3 Implementation Details

In order to improve the code implementation, some considerations are taken into account. These considerations will shape the way in which the code is developed to improve possible modifications and the code development process itself. These considerations refer to how the packages will be defined, how the code itself will be organized and additional requirements such as code comments and naming conventions.

4.3.1 Packages Definitions

ROS software is developed in packages. Packages are folders that include the code, configuration files and third-part elements that aid in the fulfilling of the package objectives. Packages can include code in Python, C++ or Lisp. The used programming language for the GPR-20 software is Python thus packages must be organized for this programming language. Listing 1 presents the package organization for the GPR-20 software.

```
gpr20_<package_name>/
|----- nodes/
|---------- node_name
|----- src/gpr20_<package_name>/
|---------- __init__.py
|---------- src_file_0.py
|---------- src_file_1.py
|---------- ...
|----- test/
```

```
├────────── ros/
├──────────── <ROS test Python files>
├────────── unit/
├──────────── <Unit test Python files>
├────────── <package_name>_ros.py
├────────── <package_name>_unit.py
├────────── <package_name>_ros_test.test
├────────── <package_name>_unit_test.test
├────── CMakeLists.txt
├────── README.md
├────── package.xml
├────── setup.py
```

Listing 1: Package organization for the GPR-20 software

The first sub folder of a package is `nodes`. The `nodes` sub folder is used to store the nodes files of a package. The node files are executable Python scripts that start a node execution. Despite being Python scripts, these files do not have an extension. In order to execute them as Python scripts, they must include a *shebang* (`#!`) at the beginning of the file. The used shebang is `#!/usr/bin/env python`.

The second sub folder of a package is the `src` folder. In order to follow the best practices of the ROS community, another folder is added inside the `src` folder with the name of the package. This folder will include the source files for the package. The source files will have the `.py` extension to indicate that they are Python files. A `__init__.py` file is used to define a Python module in the folder. This file is required for the node files to import the required features.

The third sub folder of a package is the `test` folder. This folder will have the test files for the package. Test files are divided in ROS and unit tests depending on their purpose. ROS tests are used to validate the ROS functionalities of a node. Unit tests will validate the ROS-independent code used in the package. Source files for the test will be organized in two additional sub folders: `ros` and `unit`. Additional files are required to execute the tests from the `rostes` functionality.

Additional files in the package include `CMakeLists.txt` to define how ROS will compile the package; `README.md` that will provide a brief description of the package; `package.xml` to define the ROS dependencies of the package; and `setup.py` to define the Python execution of the package. The `setup.py` file is required to execute the nodes as intended since this file provides the setup configuration of the package.

4.3.2 Code Organization

The code will be organized in up to three abstraction levels. By defining these abstraction levels, it is expected that the code will comply with the best practices in software development. Another benefit of abstracting the packages implementation is to allow updates and modifications to be performed easily. Finally, testing will be better focused by abstracting the code into different levels. The three abstraction levels are presented in table 5.

Level	Description	Use case
High	The high abstraction level is reserved for ROS functionalities. ROS functionalities refer to the definition of topics, services and actions, and features such as parameters and node initialization. It is expected for the high abstraction level to wrap the functionalities provided from the mid abstraction level. No functionalities are expected to be implemented in this level.	Since the GPR-20 software suite is completely developed over ROS, the high abstraction level is used in every package. It is not possible to develop a package without including the ROS interface.
Mid	The mid abstraction level provides the functionalities of the package. The functionalities are defined as complex procedures in which two or more low level functionalities are merged. It is also possible to use functionalities provided by third-party libraries at this level. In general terms, if a functionality can be split into simpler ones, they must be implemented in this level.	This level is expected to be included in every package since it is the actual implementation of them. The mid level can be distributed into different classes in order to group similar functionalities.
Low	This level is used to implement the basic functionalities of the package. This level differs from the mid level as its functionalities can not be split into simpler ones. Other criteria for defining the low level is the purpose of the functionalities. If functionalities depend on external elements such as hardware or remote APIs, they are required to be implemented in the low level.	The low level can not be included in a package if the basic functionalities are provided by a third-party library. Low-level implementations are expected to be defined in different classes.

Table 5: Abstraction levels for code organization.

4.3.3 Naming Conventions and Documentation

Naming conventions will be based on the ROS naming conventions. Additional conventions include starting the package name with the `gpr_20_` prefix. Naming of files will be in lowercase and will use an underscore to separate words. The documentation for the GPR-20 software suite is the Google docstring style.

5 Software Utilities

This section presents the software utilities for the GPR-20 robot. Elements of the software architecture for the GPR-20 robot are referred as utilities since their implementation is not directly bound to an specific framework. Nevertheless, ROS is used since it provides communication mechanisms and allow that utilities are independent from each other. The framework independence and the ROS integration are concealed by defining multiple abstraction levels that encapsulate the functionalities. This means that a software utility can be used with or without the ROS interface or can be easily ported to a different framework. Software utilities are accessible via their repositories. Table 6 presents the software repositories for the GPR-20 software stack.

Software	
Name	**Link**
Axis Driver	`https://github.com/gdh-uniandes/gpr20_axis_driver`
Height Sensor	`https://github.com/gdh-uniandes/gpr20_height_sensor`
VNA Acquisition	`https://github.com/gdh-uniandes/gpr20_vna_acquisition`
VNA Processing	`https://github.com/gdh-uniandes/gpr20_vna_processing`
Main Control	`https://github.com/gdh-uniandes/gpr20_main_control`
User Interface	`https://github.com/gdh-uniandes/gpr20_user_interface`

Table 6: Software repositories for the GPR-20 robot.

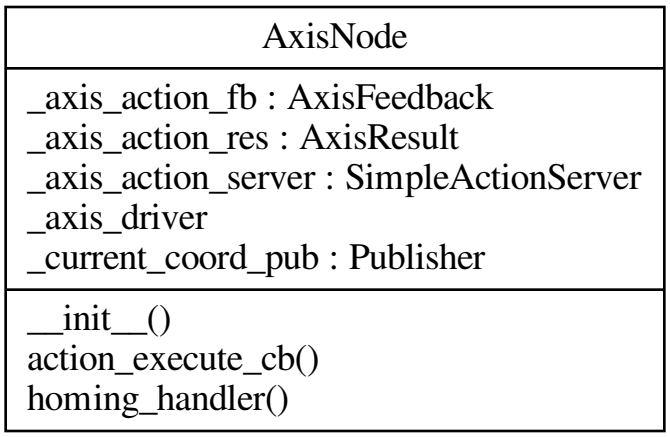
5.1 Axis Driver

The axis driver utility is in charge of handling the movement over a single GPR-20 robot axis. Three axes are handled trough this utility: the linear X and Y axes and the rotational Z axis. Linear axes depend on an endstop sensor and the motor while the rotational Z axis only depends on the motor. The software utility is defined on three abstraction levels (high, mid and low) as shown on table 7. High level is reserved for the ROS framework features like parameters, services, topics and actions. Mid level is reserved for handling the high level behavior of the axis i.e. calculating the required steps, executing the required steps to move the axis, performing the homing sequence and checking if the positioner lies within the axis range. Finally, low level is the interface from software to hardware. Low level handles the Raspberry Pi's GPIO pins as defined by the motor and sensors interface.

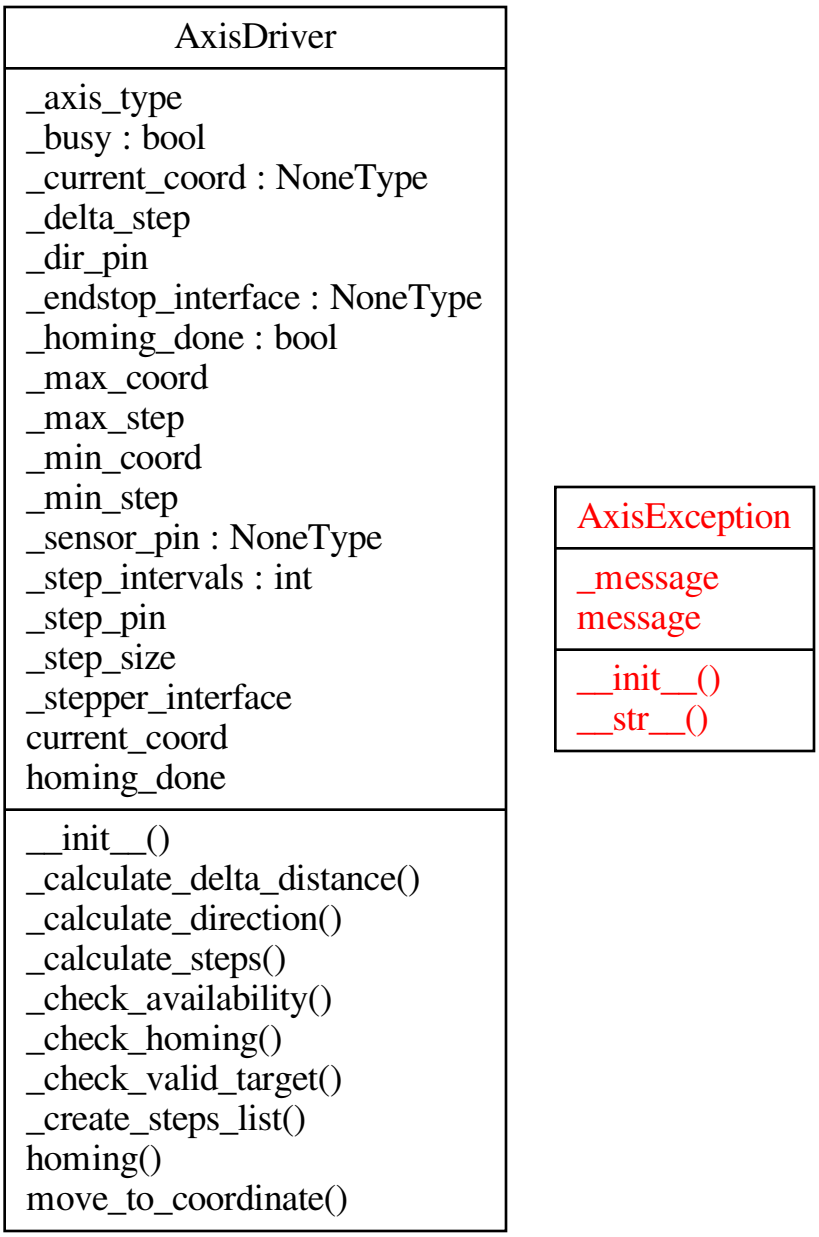
Abstraction Level	Features
High	- Top class for the axis driver utility. - Includes the ROS features such as parameters, topics, services and actions. - Makes use of the axis driver to execute requests.
Mid	- Driver for the axis. - Has two types of methods: major and utilities. Major methods are used to manage complex behaviors of the axis. Utility methods provide specific functionalities to the aforementioned complex behaviors. - Both major and utility methods use the interfaces to hardware.
Low	- Consist of the hardware interfaces for the motor and endstop sensor. - Data is driven or sampled from the Raspberry Pi's GPIO at this level.

Table 7: Abstraction levels for the axis driver utility.

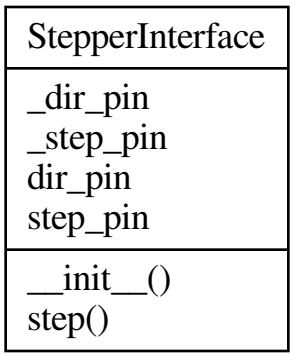
Each abstraction level can be mapped to different classes definitions that provide the features described on table 7. The high abstraction level is mapped to `AxisNode` class whose UML diagram is presented on figure 1a. The `AxisNode` class implements the ROS features for the axis driver i.e. the topics, services, actions and parameters for the node to work as intended. The mid abstraction level is mapped to `AxisDriver`, which implements the complex driver behaviors such as moving to a target coordinate and performing the homing routine. An UML diagram for the `AxisDriver` class is presented on figure 1b. Finally, the low abstraction level is implemented in two classes: `StepperInterface` and `EndstopInterface`. `StepperInterface` implements the required software routines to use the stepper motors via their drivers. `EndstopInterface` implements code to sample the endstop sensor. Figures 1c and 1d present the UML diagrams for the `StepperInterface` and `EndstopInterface` classes respectively.



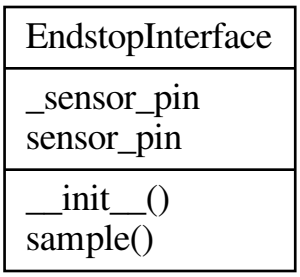
(a) UML diagram for the `AxisNode` class.



(b) UML diagram for the `AxisDriver` class.



(c) UML diagram for the `StepperInterface` class.



(d) UML diagram for the `EnstopInterface` class.

Figure 1: UML diagrams for the axis driver software utility.

The GPR-20 robot axis is intended to be used under the ROS framework even if the driver itself (`AxisDriver`) can work independently from the ROS interface. The ROS architecture for the GPR-20 axis driver is shown on figure 2. The ROS interface for the axis driver makes use of topics, services, actions and parameters. Parameters are used to setup the driver and define its behavior and are specified on table 8. The axis driver ROS interface provides a single service: `homing`. The `homing` service commands the axis to execute the homing sequence to allow further movement. A single topic is also available and consists of the `current_coord`. This topic publishes the axis current coordinate continuously. Finally, the ROS interface provides an action to move the axis. The action receives a goal and publishes the result depending on the execution status. There is no feedback from the action since the `current_coord` publishes the same information.

Parameter	Description
dir_pin	Pin number for the direction input of the stepper motor driver.
step_pin	Pin number for the step input of the stepper motor driver.
sensor_pin	Pin number for the sensor pin input. Not required for rotational axes.
step_size	Step size for axis. Consists of the minimum coordinate change made by a step from the motor.
min_step	Minimum pause value for a step in seconds. Defines the maximum speed of the axis movement.
max_step	Maximum pause value for a step in seconds. Defines the minimum speed of the axis movement.
delta_step	Step pause value minimum change in seconds. Used to define how many different speed values are available for the axis.
min_coord	Minimum coordinate value for axis. Defines the lower bound for the axis. It is also the initial value for the axis current coordinate.
max_coord	Maximum coordinate value for axis. Defines the upper bound for the axis.
type	Axis type. Can be either linear ('LIN') or rotational ('ROT').

Table 8: GPR-20 Axis Driver ROS parameters definition.

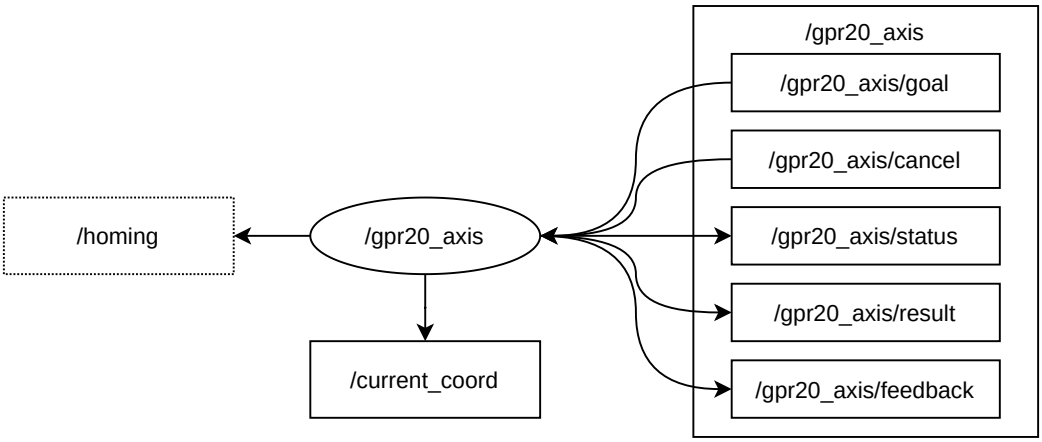


Figure 2: ROS graph diagram for the axis driver utility.

5.2 Height Sensor

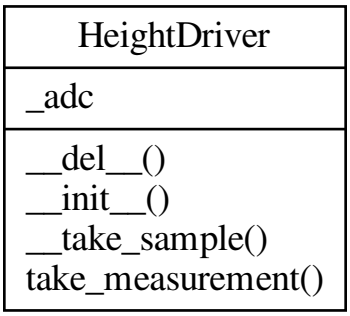
The GPR-20 height sensor driver utility is responsible of acquiring distance measurements from the antennae mount sensor. Two abstraction levels are defined for the sensor driver: high and mid. The abstraction layers features are shown on table 9. There is no low abstraction level since it would correspond to reading the analog-digital converter which is implemented in an open-source library. The mid abstraction level handles the sensor reading using the aforementioned library and the conversion to a distance value. Finally, the

high abstraction level defines the ROS interface for the utility.

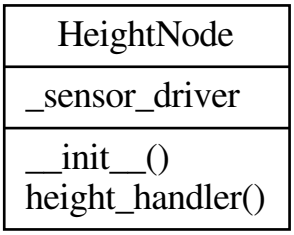
Abstraction Level	Features
High	- Makes use of the sensor driver class to acquire distance data. - Includes the ROS interface for the height sensor.
Mid	- Instances the analog/digital converter library. - Converts the sensor data into a distance value.

Table 9: Abstraction levels for the axis driver utility.

The two defined abstraction layer are mapped to Python classes that implement the features described on table 9. The mid abstraction layer is mapped to `HeightDriver` and implements a single public method that takes the distance measurement. The `HeightDriver` class makes use of the `MCP3008` library to read the MCP3008 analog/digital converter through the Raspberry Pi's SPI interface. Once the MCP3008 data is read, it is converted to a distance value using a formula derived from the sensor datasheet. The high abstraction level is mapped to the `HeightNode` class that implements the ROS interface for the height utility. A service is implemented in the `HeightNode` in order to provide the distance measurement using the ROS interface. The distance measurement is sent in the response for the `get_height` service. UML diagrams for the `HeightDriver` and `HeightNode` classes are shown on figures 3a and 3b. A graph diagram for the height sensor utility is shown on figure 4.



(a) UML diagram for the `HeightDriver` class.



(b) UML diagram for the `HeightNode` class.

Figure 3: UML diagrams for the height sensor utility.

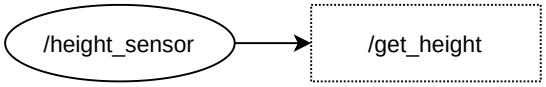


Figure 4: ROS graph diagram for the height sensor utility.

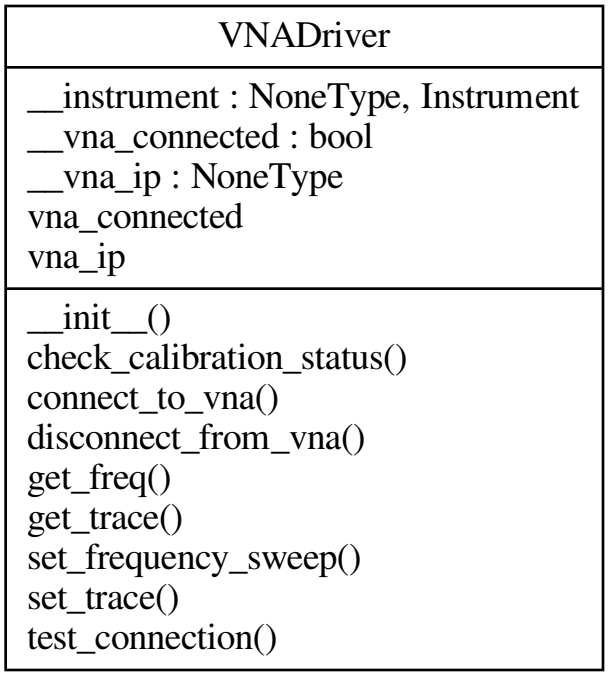
5.3 VNA Acquisition

The VNA acquisition utility is responsible of acquiring the VNA data from the device and make it available to the software architecture. Two abstraction levels are defined for the VNA acquisition utility: high and mid. The mid abstraction level corresponds to a driver that implements code to configure the VNA connection, frequency sweep parameters and get both frequency and trace data. The ROS interface for the utility is implemented in the `VNANode` class and define the service servers for the utility features. Table 10 presents the abstraction levels for the VNA acquisition utility.

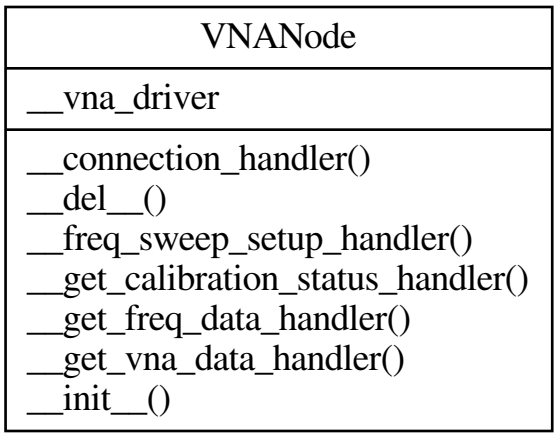
Abstraction Level	**Features**
High	- Provide four services to configure and acquire data from the VNA. - Instantiates the VNA driver class to execute the requested procedures.
Mid	- Uses the `vxi11` library to communicate with the VNA device. - Defines the processes required to connect/disconnect from the device, configure its parameters and request data.

Table 10: Abstraction levels for the VNA acquisition utility.

The abstraction levels presented in table 10 are implemented in two classes. The `VNADriver` is the implementation for the mid abstraction level and `VNANode` for the high abstraction level. UML diagrams for `VNADriver` and `VNANode` are presented in figures 5a and 5b. The `VNADriver` class communicates with the VNA device under the VXI11 protocol and uses the `vxi11` library to execute do so. The `VNANode` provide four service servers whose handlers call the respective methods from the driver implementation. The services are used for connecting and disconnecting the device, configure the frequency sweep parameters (start and stop frequencies and number of points), get the frequencies vector and get a trace. Figure 6 presents the graph diagram for the `vna_acquisition` node.



(a) UML diagram for the `VNADriver` class.



(b) UML diagram for the `VNANode` class.

Figure 5: UML diagrams for the height sensor utility.

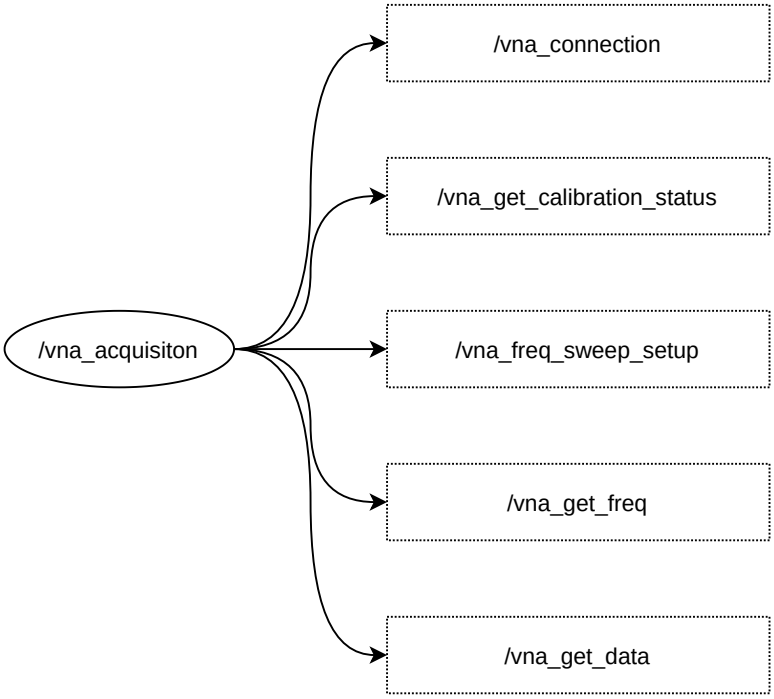


Figure 6: ROS graph diagram for the height sensor utility.

5.4 Data Processing

The Data Processing is the utility that stores the robot data in a persistent storage. The data processing utility receives the data from the ROS communication mechanisms and then stores it. Since data storage can be a time-consuming process, threads are used to avoid blocking the utility when multiple storage requests are received. It must be noted that the processing time depends on the used storage technology. Data is stored using the JSON format and includes information regarding the coordinates and date in which data was acquired, the height sensor data and the VNA trace. Three abstraction levels are defined for the Data Processing utility: high, mid and low. The high abstraction level is used to integrate the ROS functionality of the utility i.e. the service in which data is received. The mid abstraction level provides the mechanisms to execute the storage threads. Finally, the low level functionality consist of the processing stages for the data. Table 11 presents the abstraction levels of the Data Processing utility.

Abstraction Level	Features
High	- Provide a single service in which a request to storage data is received. - Instantiates the main processing class to execute the storage procedure.
Mid	- Executes the processing stages for the data from parsing to storage.
Low	- Defines the procedures for data processing. - Data does not solely consists of VNA traces, it also includes the sample coordinates, antennae height, timestamp and unique identifiers.

Table 11: Abstraction levels for the data processing utility.

The VNA processing utility is implemented in three classes: the `ProcessingNode` class for the high abstraction level, the `DataProcessing` class for the mid abstraction level and the `StorageThread` for the low abstraction level. The `ProcessingNode` class implements a single method to handle the storage request received via the `/store_data` service. The high-level class also instantiates the `DataProcessing` class. This class implements a single method that bypasses the data to the low-level thread. This approach is used to facilitate the inclusion of different processing stages for the VNA data. Finally, the `StorageThread` parses the data into the JSON format and stores the data.

5.5 Main Control

The main control is the utility that coordinates the features provided by the remaining utilities from the software architecture. The user configures the main control through the user interface in order to use the GPR-20 robot. Configuring the control unit is usually done from the user interface yet it can be configured from other interfaces such as command consoles and web applications.

The main control interfaces with the user through services and topics of two different types: configuration and feedback. Configuration interface corresponds to the mechanisms in which the survey parameters and commands are passed to the control. Feedback interfaces provides data from the control unit to the user. Feedback data is intended to be displayed on the user interface. Table 12 presents the ROS communication mechanism for the main control unit. The `set_survey_parameters`, `start_survey` and `stop_survey` correspond to the configuration mechanisms. In the other hand, `control_status`, `control_target_coordinate`, `control_current_coordinate` and `control_survey_data` correspond to the feedback mechanisms.

Name	Mechanism	Description
set_survey_parameters	Service	This service is sent by the user with the survey parameters. Survey parameters include the frequency start and stop frequencies, the sampled frequency points, start and stop values for both the X and Y axes, and the surveyed points per axis.
start_survey	Service	This service is used as the start command for the survey process. The service message does not carry any data in it.
stop_survey	Service	This service is used as the stop command for the survey process. It must be noted that this command is reserved for the situations in which the user might want to stop the surveying process. The survey stops itself when finished otherwise.
control_status	Topic	This topic broadcasts the general status of the GPR-20 robot software. The topic message contains a status code and a status message.
control_target_coordinate	Topic	This topic publishes the current target coordinate of the GPR-20 positioner. The topic message contains the three relevant coordinates of the GPR-20 robot. A negative value is used when no survey is taking place.
control_current_coordinate	Topic	This topic publishes the condensed current coordinate of the GPR-20 robot.
control_survey_data	Topic	This topic publishes statistical data of the current survey. This data includes the completion percentage, total points, and visited points.

Table 12: Communication mechanism between the GPR-20 main control and the user interface.

Additional communication mechanism are defined in order to command and exchange data between the main control and the different utilities. These communication allows the main control to effectively execute the required commands. The main control directly commands three (3) `Axis Driver` instances, the `Height Sensor` utility, the `VNA Acquisition` utility, and the `VNA Processing` utility. Table 13 presents the communication mechanisms between the main control and the different software utilities. It must be noted that since three (3) `Axis Driver` instances are used with the same interface, their communication mechanisms are presented only once.

Name	Utility	Mechanism	Description
axis	Axis Driver	Action	This action provides the mechanism to set the target coordinate of the axis and execute the required movement. The action does not provide feedback since there is a topic used for this purpose.
current_coord	Axis Driver	Topic	The topic publishes the current coordinate of the axis. An specific Boolean flag is also included in the message to indicate when the axis has not executed the homing sequence.
homing	Axis Driver	Service	Used to command an axis to execute the homing sequence of the driver. No response is expected from this service besides from changing the homing status flag.
get_height	Height Sensor	Service	This services provides the requester with the latest measurement from the height sensor. Using the service allows to synchronize the height measurement with the data acquisition process.
measured_height	Height Sensor	Topic	The topic continuously streams the measurement from the height sensor. This information is used for feedback purposes.
get_data	VNA Acquisition	Service	It is used to retrieve the data from the VNA instrument. The data corresponds to the S21 parameter acquired within the specified bandwidth.
get_freq	VNA Acquisition	Service	This service is used to retrieve the frequency values in which data is acquired. This service is used to determine which frequencies are sampled during the data acquisition process.
connection	VNA Acquisition	Service	This communication mechanism is used to execute the VNA connection in the VNA Acquisition node. The response consists of a Boolean flag indicating the result of the connection procedure.
freq_sweep_setup	VNA Acquisition	Service	This service is used to configure the VNA device with the frequency sweep parameters.

Continues on next page.

Name	Utility	Mechanism	Description
`store_data`	`VNA Processing`	Service	It is used to store data in a persistent location. Data does not only include the VNA responses but its coordinates, height measurement and frequency values.

Table 13: Communication mechanism between the GPR-20 main control and the software utilities.

The main control utility is implemented in four (4) classes along three abstraction levels. The used abstraction levels do not correspond to the defined abstraction levels for other utilities. In turn, the abstraction levels are defined to properly interface with the different software utilities. Table 14 presents the three abstraction levels of the main control software utility.

Abstraction Level	Description
High	The high abstraction level is reserved for the communication between the graphical user interface and the main control. This abstraction level can either receive or send information and/or commands from the control to the interface and vice versa. This is defined as the highest level as the information and commands are closer to the user.
Mid	This abstraction level organizes and commands the different actions in the robot in order to execute them as intended. The mid abstraction level implements the functionalities and code to execute the survey keeping into account the synchronization between the different actions. It must be noted that this abstraction level does not communicates directly with other software utilities.
Low	This abstraction level groups the functionalities to be accessed in different utilities. The idea of the low abstraction level is to implement the communication mechanisms between the control and other utilities.

Table 14: Abstraction levels for the main control.

5.6 User Interface

The user interface is the utility that provides the user with a graphical interface from which the robot can be commanded. The graphical interface is connected to ROS in order to send and receive both data and commands. The graphical interface is built using the PyQt5 framework. The graphical interface provides an user to execute the functionalities described on table 15.

The functionalities are implemented in such a way that the user modifies the data in the interface and sends the commands to the main control. The decoupling that exists between the user interface and the main

Functionality	**Description**
Input the VNA IP address	The VNA IP is used to connect the VNA acquisition utility with the device itself. It must be manually entered by the user based on the device configuration.
Input the VNA frequency points	The number of frequency points that will be sampled depend on the VNA calibration. Timing constrains and data quality also are taken into account to define the frequency points.
Input the VNA start/stop frequencies	The start/stop frequencies of the VNA sampling are defined by the user based on data quality. The frequencies are also defined based on the VNA calibration.
Input the sample start/stop coordinates	The start/stop coordinates will determine how large will be the survey area. The coordinates are referred to the linear axes of the GPR-20 robot. Each coordinate consists of a pair of values that range from 0 to 800. There is no constrain on which value (start or stop) needs to be higher than the other thus a *reverse* survey can be performed.
Input the sample points per axis	The number of sample points define the resolution in which data will be acquired. The user will input a number that defines the number of points in which the axis will stop considering the whole length of the axis.
Start/stop a data-acquisition	The user is able to either start and stop the data-acquisition via the user interface. If an error is raised when starting the data-acquisition, a pop-up will show the error. The start and stop buttons will be enabled or disabled based on the data-acquisition status.

Table 15: User interface functionalities.

control allows for some situations to arise and compromise the user experience. These situations consists of the main control being at certain state with the user interface not appropriately representing it. For example, the main control could be executing a data-acquisition survey while the interface allows for sending parameters and the start command. Two workarounds were implemented for this situations: providing a bidirectional communication between the user interface and the main control, and assigning responsibilities to the user interface.

The bidirectional communication between the user interface and the main control is critical for a consistent user experience. The main control provides the user interface with feedback in order to prevent congruent situations to arise. The user interface then changes its status to only allow for valid actions to take place. Following with the previous example, if the user interface is initialized after a survey is initialized, the main control provides feedback to the user interface in order to prevent sending new parameter values and/or the start survey command. This prevention is done by locking or unlocking the user interface input fields. It must be noted that although the user interface prevents such situations from happening, the main control

can handle such erroneous requests without compromising the data acquisition process.

The second workaround makes the user interface is responsible on checking the validity of the input data from the user. Checking the validity does not check that values are correct but that they can be properly used to configure the robot. For example, the user can input the VNA IP address with only three out of the four fields with a number. In this situation, the user interface will check that the IP address is valid thus warning the user that its value should be checked. However, checking that the entered IP address correspond to the VNA address is done in the main control. In this situation, the user interface will receive the feedback from the main control and then warn the user on the erroneous data. Table 16 presents the input fields for the user interface with the checks that are performed in the user interface.

Field	**Checks**
VNA IP Address	- Check that it has its four numbers set.
Start Frequency	- Check that is a value different from zero. - Check that the value is lower than the stop frequency.
Stop Frequency	- Check that is a value different from zero. - Check that the value is higher than the start frequency.
Frequency Points	- Check that is a value different from zero. - Check that the value is lower or equal than four thousand (4000).
X-Axis Start Coordinate	- Check that the value is positive. - Check that the value is lower than eight hundred (800).
X-Axis Stop Coordinate	- Check that the value is positive. - Check that the value is lower than eight hundred (800).
X-Axis Points	- Check that the value is greater than two (2).
Y-Axis Start Coordinate	- Check that the value is positive. - Check that the value is lower than eight hundred (800).
Y-Axis Stop Coordinate	- Check that the value is positive. - Check that the value is lower than eight hundred (800).
Y-Axis Points	- Check that the value is greater than two (2).

Table 16: Validity checks executed on the user interface.

The user interface is presented in figures 7 and 8. The user interface allow the user to input the VNA IP address, the frequency parameters and the coordinate parameters. The VNA address is entered via four input fields. This separation is done since the user interface is designed for a touchscreen where a keyboard might not be present. The remaining input fields receive the input values from the user via the up and down arrows. The arrows allow for value changes in steps. The value of the steps is configured by the user in order to allow for value changes more rapidly.

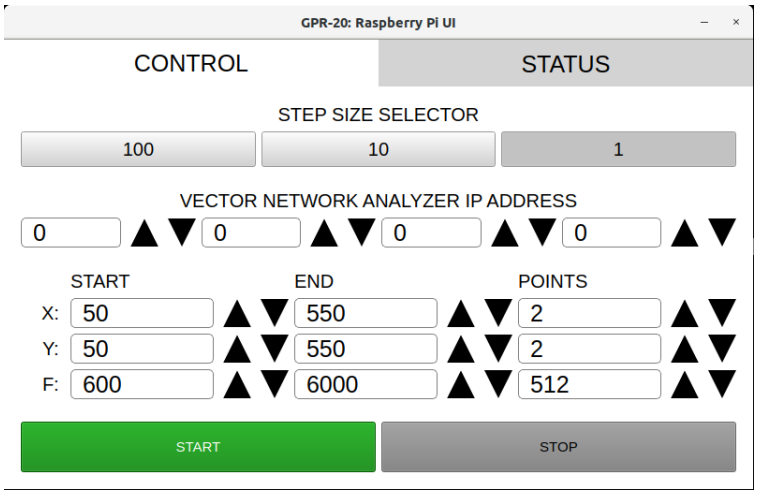


Figure 7: User interface (control) for the GPR-20 robot.

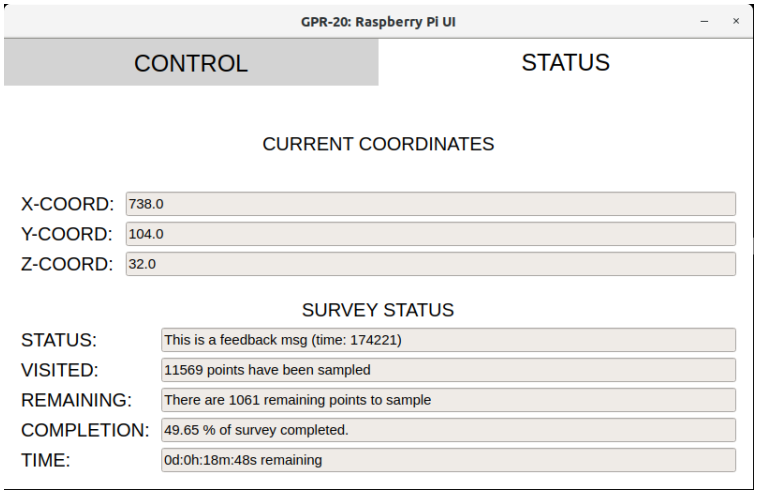


Figure 8: User interface (status) for the GPR-20 robot.