Universitetet i Stavanger

Faculty of Science and Technology

Bachelor's Thesis

Study Program/Specialization: Bachelor of Science / Datateknologi	Spring semester 2022 Open
Authors: Lars Sverre Levang, Vidar André Bø og Jonas Emanuel Gilje	
Faculty supervisor: Tomasz Wiktorski Thesis title: Digital Architecture for an Automated Drilling Rig Credits (ECTS): 20	
Key words: Drillbotics, GUI Design	Pages: 86 + enclosure: 18 Stavanger, 15 May 2022

University
of Stavanger

Faculty of Science and Technology
Department of Electrical Engineering and Computer Science

Digital Architecture for an Automated Drilling Rig

Bachelor's Thesis in Computer Science

Lars Sverre Levang[1]
Vidar André Bø[1]
Jonas Emanuel Gilje[1]

Faculty supervisor
Tomasz Wiktorski[1]

Affiliations
[1]University of Stavanger

May 14, 2022

Abstract

The objective of this thesis was to develop the digital architecture for a small-scale drill rig intended for use by the Drillbotics team at the University of Stavanger, for an international student competition by the same name. The main goals of the project has been to develop a robust software architecture, data acquisition system, data management system and graphical user interface. The main criteria are the guidelines given by competition organisers, criteria given in the thesis description, and criteria given by the Drillbotics team.

We created a system for communication between the computer, programmable logic controller and the drill rig such that we can communicate between platforms using the CAN protocol. With this communication in place, both the data acquisition logging and control system can operate without delay. Any data retrieved is stored in a data management system, as per competition guidelines. The database has been stress tested and has a 15x safety margin between operation- and top speed, ensuring the database will not be a bottleneck.

The main human machine interface for the drill rig, the graphical user interface, on the computer was developed using principles researched in advance to ensure an interface that was based on good industry practices. The big focus on researching proper methods of making the interface is due to the competition recently adding human machine interface as a major judging criteria in the competition.

A system has also been developed that covers models used for steering the directional drilling according to the industry standard minimum curvature method. The path given by the minimum curvature method is the ideal path that we try to follow. The path also has safety margins given to it to ensure the rig never strays too far from the path.

The systems created in this project have had a side-goal of being scalable and using good abstractions such that it is able to be used by future Drillbotics teams, for both future computer science bachelor groups, or the rest of the team as well.

Acknowledgments

We would like to thank our thesis supervisor prof. Tomasz Wiktorski for all our productive talks, always having valuable insight for what we needed to prioritise, and always answering our questions in a swift and clear manner. His impact is resounding throughout this thesis.

We also want to thank Drillbotics team member Magnus Wersland for his help in introducing us to the project, helping us develop the data acquisition system, and general questions about our project. From the team we also want to thank Mya Chordar for her valuable industry insight and giving us good ideas for our designs.

Lastly we want to thank both family and friends, and particularly our fellow students who always help us stay motivated to push forward together.

Contents

List of Tables

List of Figures

Code Blocks

Abbreviations

BHA Bottom Hole Assembly.

CAN Controller Area Network.

CPU Central Processing Unit.

CSV Comma-separated Values.

DBMS Database Management System.

DHS Downhole Sensor.

GUI Graphical User Interface.

HMI Human–Machine Interface.

PDM Positive Displacement Motor.

PLC Programmable Logic Controller.

RPM Revolutions per Minute.

RSS Rotary Steerable System.

SDK Software Development Kit.

SPE Society of Petroleum Engineers.

TVD True Vertical Depth.

UiS Universitetet i Stavanger (University of Stavanger).

WOB Weight on Bit.

Chapter 1

Introduction

Every year since 2017, the University of Stavanger has participated in a global student competition designed to have students worldwide develop a small-scale drill rig to drill a provided rock sample. The competition aims to teach students important industry methods, as well as hoping to find new interesting approaches to existing problems that exist in the petroleum industry.

The 2022 competition is the first competition that UiS participates in since 2020 due to covid-19 affecting the organisers ability to have the competition run in 2021. From the former competition to the 2022 competition there has been a complete redesign in the control-system architecture. It now uses a programmable logic controller (PLC) as opposed to a set of Arduino microcontrollers to communicate with the rig and control the drilling process. The project's goals is to develop the systems and code to steer the rig, creating the interface and data acquisition systems.

1.1 Thesis

The objective of this project was to produce the components to control the rig as desired, where following the competition requirements was our main priority. This thesis describes a complete set of software components developed to enable a drill rig to drill in of sample of rock or concrete. The main objectives were to develop a robust data collection system between machine and drill rig, design a graphical user interface (GUI) for live drilling, create a proper software architecture, and database management for all the data that gets handled.

In prior years of UiS participating in the competition, code has been made with similar goals to what was given to our project. Therefore we also reviewed the codebase that already existed, and gauged whether this was valuable for the this project. As the Drillbotics competition is held annually there was also a side goal that was deemed important and was given some priority, which was to make the created code and eventual thesis approachable and usable by the rest of the Drillbotics team, for both present and future groups.

The Drillbotics competition is set to be held in the summer, roughly two months after the deadline of this thesis. The physical rig is scheduled to be completed, and tested in the weeks following the thesis submission deadline, and testing of the code on the physical rig can therefore not be fully described in this thesis. This situation necessitates certain precautions and principles to be used in the development of the logical components which this thesis focuses on. There was an emphasis that the system was to be developed as robust as possible, and is able to swiftly accommodate modules that are removed or added depending on the requirements of the team as the competition deadline draws nearer. The other necessity is that we have a healthy and robust testing methodology such that physical testing goes without issue.

At the beginning of the project it was decided together with UiS Drillbotics and our supervisor that the tasks for this thesis would be extended such that a larger focus could be put on the highest priority features. The reason for the extension was due to there not being a group for the electrical engineering side of the project as originally intended, and our group was most qualified to handle the more important parts that was supposed to go to the electrical group. This means that the *data acquisition system*, and partially the *sensor installation and calibration* would fall under this thesis. Mainly it meant that this thesis also entails having to develop most of the data acquisition system as well, both on the PLC and computer side. As mentioned in sub-chapter 1.1 the main focus was software architecture, GUI development, database management and data acquisition, as these were all the largest priorities to get a working system.

1.2 The Competition

To better understand the motivation for our work, here is some background on the competition. Drillbotics is an international annual competition organised by the Society of Petroleum Engineers (SPE). The objective of the competition is to design and create an autonomous lab-scale drilling rig that leverages sensors and control algorithms to drill into a rock sample provided by the competition.

The aim for the competition is for students to "*Design a drilling rig and related equipment to autonomously drill a vertical well as quickly as possible while maintaining borehole quality and integrity of the drilling rig and drill string.*" [4]. Points are granted based on how well the aforementioned points are done. This year is the 8th year the competition is held, and the 6th year that the University of Stavanger participates. The participating teams have to build the rig from the ground up and implement all the software needed to operate it themselves. Up until now the only necessary function for the competition has in theory been a to start/stop the autonomous part of the drilling.

Competition guidelines frequently change to better reflect current industry problems, giving students practical insight into how things work in the industry. This year the newly added change is that the drill operators should be able have more interaction in the drilling interface, to better be able to respond to unforeseen situations, and human machine interfaces (HMI) has a big focus. These guidelines and recently new editions to it will be one of the primary factors leading our design choices.

Chapter 2

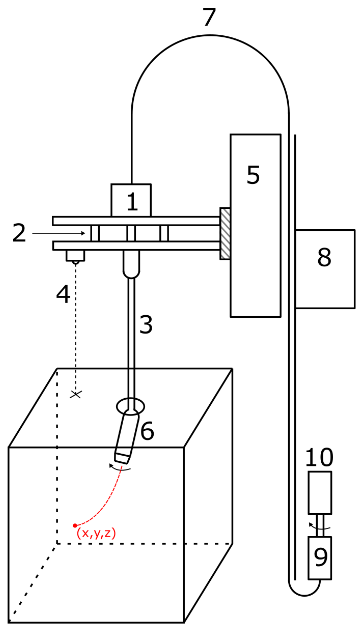
The Drilling Rig

The objective for this thesis as mentioned in chapter 1.1 is to develop the logical architecture around a small-scale autonomous drilling rig for a international student competition named Drillbotics. While these tasks mainly lie in creating code and designing interface elements, it's helpful to have a good overview on the piece of physical equipment that the code is developed for. The assignment text described the goals and objectives:

The following is part of the assignment text:

A small-scale autonomous drilling rig has been constructed and tested from May 2017 at UiS. The work is based on a multi-disciplinary approach to develop, optimize, and capitalize on a variety of research areas such as mathematical modelling, mechatronics, information technology, cybernetics, software, and petroleum engineering. An important objective is to strengthen the understanding of the control system design for drilling systems in terms of performance optimization, drilling problems management, data analysis, and model calibration and validation, etc. The goal has been to use this fall and spring to construct a new drilling robot, as the second generation of our UiS drilling robot. This new robot will be used for an international university competition organized by SPE.

See image and descriptions below to get a better view into how the rig looks, and its individual pieces:



- 1. Top drive
- 2. Load cells
- 3. Drill string
- 4. Laser sensor (distance)
- 5. Electric hoist/elevator
- 6. Bottom hole assembly
- 7. Water and electric cables
- 8. Electric box
- 9. Pump
- 10. Engine for pump

Figure 2.1: Drawn illustration of the drill rig.

The above illustration is a good pointer for how the rig looks and the components that go into making it work. See below for a bit more detail into what each part individually is responsible for.

2.1 Top Drive

Formerly the *top drive* has been used to make a "pilot" hole in the block we are drilling into, making the entire drill string rotate with a set RPM (revolutions per minute). A change from former designs to the current one is the use of a positive displacement motor (PDM) to control RPM on the bit, meaning the pilot hole cannot be driller with the top drive. The top drive is now then used primarily together with the rotary steerable system (RSS) to rotate the drill string to decide downhole drilling direction.

2.2 Load Cells

The *load cells* are there to be able to determine the weight on bit (WOB), which is the amount of downward force exerted on the drill bit. Normally the weight is expressed in thousands of pounds, but for our case we use Newtons.

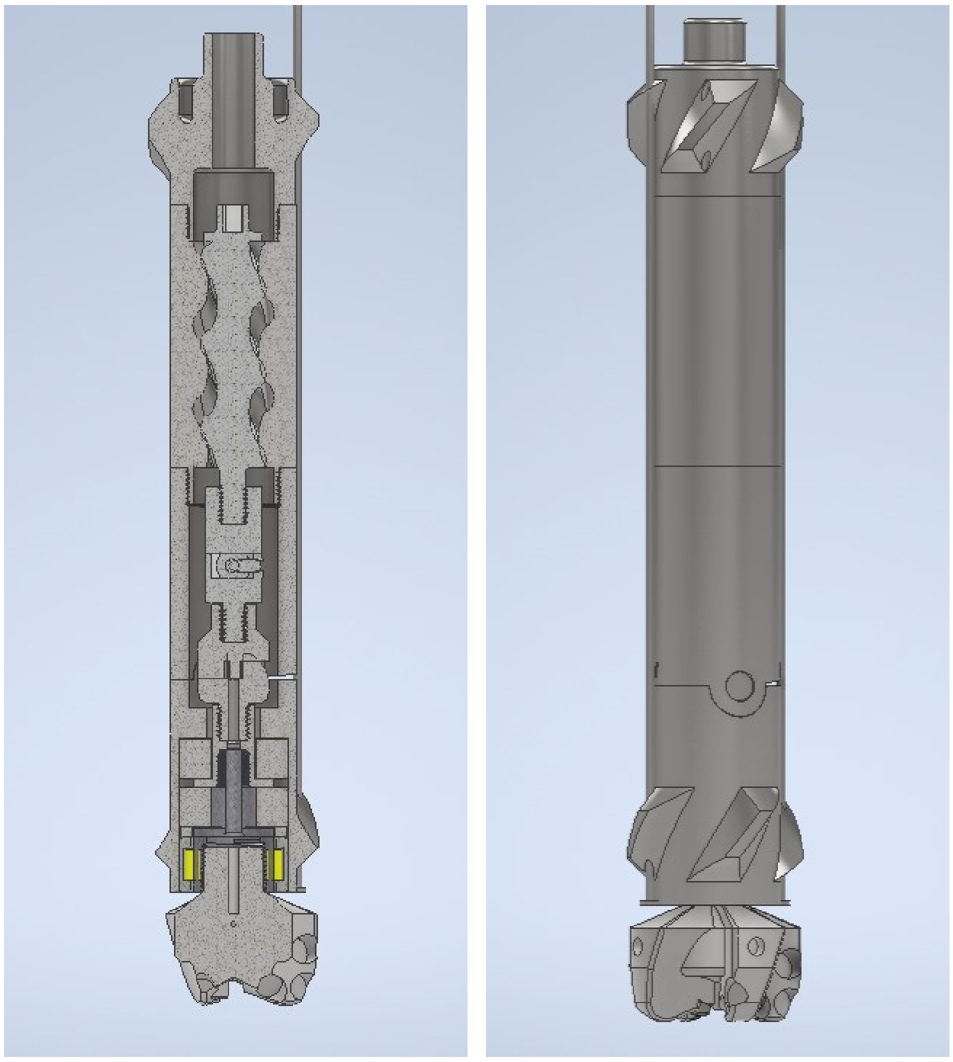
2.3 Hoisting System

The rig uses a Bosch Rexroth R036050000 to raise and lower the drilling assembly. It's speed will be adjusted to apply the required weight on bit.

2.4 Bottom Hole Assembly

The *bottom hole assembly (BHA)* is the main component that allows for drilling. The responsibility of the BHA is to provide RPM to the drill bit, or more precisely the positive displacement motor (PDM) that's a part of the BHA. The positive displacement motor works by pushing some form of drilling fluid through it, converting the hydraulic power of the fluid into mechanical power to make the drill bit rotate. In this specific case, water will be pumped with sufficient force through the PDM to make the bit rotate.

The BHA also has a rotary steerable system (RSS) that is connected between the PDM and the drilling bit. It controls steering and goes between straight and deviated steering, meaning it either goes straight down, or is bent in one direction. The direction is determined by 2 steel cables that gets pulled to drag the directional part of the RSS back and forth. It has a drilling cable through the middle that controls rotation between the PDM and the bit.



(a) Interior view (b) Exterior view

Figure 2.2: Exterior and Interior view of the Bottom Hole Assembly used. Permission given by Drillbotics for use in thesis.

Figure 2.2 is a illustration of the insides of the BHA used at UiS. The upper threaded section is the PDM, it functions by pushing water into it with sufficient pressure to make it turn around, converting hydraulic power into mechanical power. The PDM is connected to a drilling cable that goes all the way down through the RSS and to the bit, and is the part that makes the bit rotate. The RSS is located between the PDM and bit and is there to steer the direction that the bit is pointed, either being positioned straight down, or deviated in one direction. The best curve to reach any given point in a block is the curve between 2 points on a circle essentially, so it only needs to be able to go straight or in that one direction. See below for an image of the exterior of the BHA.

2.5 Pump

The pump is powered by a 400V electric motor and will supply pressure to the circulation system. The pump is capable of operating at up to 200 bar, but in this application it will be supply pressure in the 50 bar range. The high pressure water is used to supply hydraulic energy to the PDM which will rotate the drill bit.

2.6 Other Parts

The *electrical box* contains the connections to the programmable logic controller (PLC), and handles sending and receiving signals to/from the PLC. The *Water and electric cables* supply water and electricity to the PDM and top drive respectively. The *drill string* holds the BHA, and is hollow to allow water to flow through it. The *laser sensor* is used to control how far down the drill is, and is used both while drilling, and while setting up drilling. Connected to the rig are also a number of limit switches, two of which are used to tell if the hosting system is in the top or bottom position.

Chapter 3

Software Architecture

3.1 Introduction

The automated drill needs some form of communication to operate. This communication needs to have relatively high throughput and most importantly, low latency. Data will have to travel from sensors to the PLC and PC fast enough that the drilling process is able to be automated. Creating a system like this proposes a number of challenges, most of which simply boil down to a producer-consumer problem.

The architecture of the existing code is based around a microservice based approach. This might have been a good solution for a setup with a lot of Arduino microcontrollers, but for the current setup with a single PLC it is much too complicated. The codebase also has a complete lack of unit and integration tests, so it will be very difficult to port over to a new solution. Therefore we have elected to build a new codebase from scratch.

The project will as before use the Python programming language, and the codebase will be relying on object-oriented programming techniques such as encapsulation, abstractions, inheritance, etc. As a large priority of this thesis will be to make a proper user interface, Python was convenient due to its PyQt5 library making it fairly trivial to make a graphical user interface according to our needs. While it is not necessarily a perfect language, since techniques like information hiding are enforced only by naming conventions [7], it does have the features that are needed for the project and is fairly simple to write.

3.2 The Producers and The Consumers

On the drill itself, there are two independent systems that produce data, the PLC and the DHS (downhole sensor). These two will produce data at different rates, so acquiring data from these sources cannot be done effectively with a synchronous approach. On the PC, one producer of data is the user inputs from the GUI. The GUI must be able to effectively collect user input while the data acquisition from the drill rig is still active. The data which is collected from the GUI inputs will be passed on to the controller which will then be passed on to the PLC to control the drilling process. As well as being a producer, the GUI is also a consumer as it picks up the data that will be displayed on screen.

3.3 Why It Cannot Be Done Synchronously

When building a basic implementation of the GUI it became obvious that there were performance implications with updating the GUI inline with a data intensive system. Simply running a `while True:` loop in Python made user interface unusable. This is caused by the fact that if a loop is run in the GUI process, it will not allow the PyQt5 event loop to function properly, thereby grinding the UI to a halt.

On the communication side there were also problems. When reading data from the CAN bus, the read function will block execution until it has received a message. The library did not allow for the program to just continue if there were no messages, and no message could be skipped either. This meant that if there was a delay, for instance when updating the GUI, there would be a huge backlog of unread messages on the bus.

3.4 The Tools Used

To solve the producer–consumer problems, the system relies heavily on the `multiprocessing` module in Python. This module can be used to spawn separate processes which will allow for parallelism in the program. It also provides classes which which can handle thread safe communication between the processes such as the `Queue` class.

The `PyQt5` library also has classes and functions for handling long-running processes. One of these classes are for instance used for creating the loop that acquires data from the main process. These solve the aforementioned problem with the event loop not working.

The PLC and DHS is connected to the PC over a CAN interface, the `python-can` module together with the *Kvaser* driver (a dependency needed to operate with the add-in card) is used to allow reading from these interfaces in Python.

3.5 Architectural Decisions

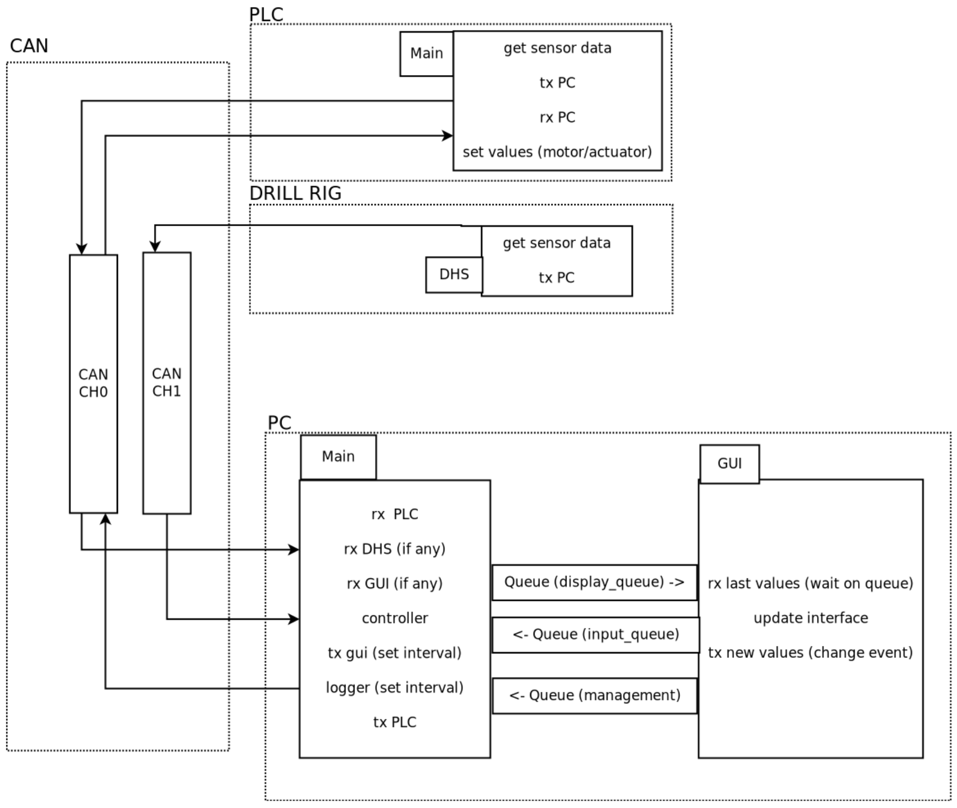


Figure 3.1: Architecture

Figure 3.1 shows a general overview of the information flow from the software perspective.

3.5.1 General Project Structure

At the core of program is the `Main` class. This is the class that contains all the processes, the queues for inter-process communication, the controller and the highest level abstraction of the CAN communication. This class also contains the main loop of the program, which is responsible for collecting the data from the various processes and passing them on to where it will be processed.

3.5.2 GUI

The first part of the program that was separated out was the graphical user interface. This is done by creating a new process and giving it several queues that it shares with the main part of the program. One of these queues is the display queue, the queue used for sending data that will be displayed on screen. The GUI process will sit and wait for data to arrive on this queue and update the data on screen when it arrives. Data will not be sent onto this queue on every iteration of the main loop however. As the system receives thousands of messages every second it would be impossible for the GUI to keep up. The display data is therefore sent onto the queue every x iterations of the main loop, where x is adjusted automatically based on the how fast the GUI is able to handle the data. This is explained in detail in chapter 7.2.5.

When receiving input from the user, this data will also be put onto a queue. This will happen as often as the GUI inputs are updated by setting up events in the GUI. The input data will be picked up from the queue by the main function, but rather than waiting for it, the main function will just check if the queue is not empty and pick up whatever is there. This is done to avoid a potential deadlock. Using this event base approach in the GUI also made sure that user inputs are working, albeit with a slight but consistent delay even if the display logic if overloaded. This was tested by trying to update a plot much faster than the GUI was capable of, and then turning the hoisting mechanism on the drill rig on and off repeatedly.

Separating the user interface and the control logic parts of the program has a number of benefits. The most obvious one is modularity. It would be fairly simple to write a different frontend for the application as they are only connected by a couple of queues in Python's multiprocessing module. Similarly, the user interface could also be connected to a different controller since this naturally is also just dependent on a couple of queues to function. Another benefit is the speed increase that comes along with running parts of the program in different processes. Separate processes will allow Python to utilise multiple CPU cores, but it will also prevent having to wait for the display logic to collect more data from the acquisition system.

3.5.3 Class Design

When designing the classes for the classes, it was put in effort to adhere to the single responsibility principle. Abstraction layers are created to simplify both the development process, but also the maintainability in the long term. An example of this is that the `Main` class “knows” of the existence of a controller and it’s methods, but not the implementation. The `Main` class can give data to it and receive data from it, but the processing that is done by the controller is not it’s concern. All of this means that as long as the same methods in the controller can be called, the controller implementation can be changed without affecting the `Main` class.

For classes that will have similar functionality, but will contain different data fields and structure, a base class is created. This base class will contain the methods that will be the same regardless of the other data fields, while the child classes will contain the methods and fields that are unique in implementation.

3.5.4 Main Loop

The main loop (code block 3.1) of the program is located in the `Main` class and is sort of the centre hub of the application. The loop runs until the program is exited, and it consists mostly of high level method calls to the main class’ objects.

3.5.5 Drill Controller

The controller part of the system is responsible for collecting and routing the different types of data that is acquired by the acquisition system. An example of this is the collecting of data that will be displayed in the GUI from the different sources. The controller is also intended to process the data that is collected from the downhole sensor and from there estimate the bit position. Code block 3.2 shows the method that is called from the main loop to handle the incoming data from the PLC. The incoming object gets stored in the controller and added to the relevant fields in the display data

```
while running:
   # Read data from PLC and pass to controller
   plc_data = self.PLC_CAN_conn.read()
   self._logger.log_PLC(plc_data)
   self.drill_controller.plc_sensor_input(plc_data)

   self.read_DHS()
   self.read_user_input()

   # Read from management queue for closing program and
      for controlling update interval
   if self.GUI_management_queue.qsize() > 0:
      msg = self.GUI_management_queue.get()
      if msg == -1:
         break
      if msg == 1:
         self.GUI_flow_controller.slow_down()

   if self.GUI_flow_controller.run():
      # Update the GUI when the flow controller allows it
      self.display_queue.put(
         self.drill_controller.gui_output()
      )

   # Send control data to plc
   self.PLC_CAN_conn.send(
      self.drill_controller.generate_control_output()
      .get_can_output()
   )
```

Code Block 3.1: Main loop

object. The display data object can then be extracted and sent to the GUI at any time as the controller will always have a copy. Similar functions to this can be written for the DHS and for other data sources in the future. This makes it so that new incoming data also can be added at any time from different sources. The relevant data from these sources can then be sent to the GUI in one go, rather than having one for each source. This reduces complexity on both sides of the communication between the GUI and the controller.

```
def plc_sensor_input(self, sensor_data: PLCSensorData):
    self._display_data.refresh_rate.PLC =
        self.PLC_refresh_rate_calc.update()
    self._plc_sensor_data = sensor_data
    self._display_data.temp_circulation_system =
        sensor_data.temp_circulation_system
    self._display_data.pressure_circulation_system =
        sensor_data.pressure_circulation_system
    self._display_data.hoisting_height =
        sensor_data.hoisting_height
    self._display_data.interlock = sensor_data.interlock
    self._display_data.wob = sensor_data.wob
```

Code Block 3.2: Receive data from PLC

3.5.6 CAN Connections

Communication over CAN from the PLC and DHS to the PC will happen over two different channels running at different bandwidths. Because of this, data will arrive asynchronously and the acquisition of data from the different channels are therefore separated. Channel 0 which receives data from the PLC will run inline with the main loop, but Channel 1 will run in a separate process. This process will have a queue that it can send data over similar to the GUI. The `Main` class' loop will read data from this queue whenever there is data on it. The CAN communication is described in depth in chapter 4.

Chapter 4

Data Acquisition

4.1 CAN

The data acquisition system uses CAN as the main protocol. Pythons `python-can` module and Kvaser's SDK is used to communicate with the CAN interfaces. The system uses two different CAN channels, channel 0 which runs at 1000 kbps and channel 1 which runs at 500 kbps. The communication between the PC and PLC runs at channel 0 while channel 1 is used by the DHS (downhole sensor) and the PC. Channel 1 is run at a lower bandwidth because it uses a much longer cable and also has to run through a slip ring.

Each message is assigned an ID and this is used to categorise the message. To make the ID assignments more clear, the messages sent from the PLC to the PC have IDs in the range 100 - 199, the messages from PC to PLC 200 - 299 and the ones from the DHS to the PC have IDs in the range 300 - 399. As an example, the pressure and temperature of the circulation system gets sent from the PLC in a message with ID 130. The full list can be found in the GitHub repository, see appendix A.

4.1.1 Conversion of Data Types

The CAN messages have a max size of 8 bytes. The values sent will vary in size, some will only need 1 byte, while some will use 4. To simplify the send and receive process, all values utilise 4 bytes of the message.

Both the CAN implementation on the PLC and in Python requires that the messages are formatted as an array of bytes. This required conversion of floating point numbers and integers to make them fit in a byte array. On the PC this is handled by Python's `struct` package, while the PLC required a lot more low level trickery. The conversion of REAL (32 bit floating point) to 4 bytes of a byte array when sending from the PLC is accomplished by creating a pointer to an array of 4 bytes and pointing it to the REAL variable. This is demonstrated in code block 4.1. The 4 bytes in the CAN message are then assigned to the value of this pointer, thus writing a REAL to 4 bytes. When receiving a REAL in form of a byte-array on the PLC the same procedure is performed, only in reverse. Conversion from byte-array to DINT (32 bit INT) and reverse on the PLC is done by shifting the byte values by a multiple of 8 bits according to its position in the array. On the PC a combination of the `struct` package and `int` type's methods to accomplish the same.

4.2 Receiving

Code block 4.2 shows how the CAN bus is read from. The `self._bus.recv` function call is set to a timeout of two seconds so that the program can continue even if no message is received. If so, the CAN connection is set to a timed out state where the it will not attempt to receive data unless it is reactivated by the operator.

To avoid enormous classes and functions for receiving and parsing messages, Python's duck typing feature is used. This enables the functions for receiving a message to be written only once, while the methods defining how the message is structured is defined in the message's class. The ideal approach for this would be to use interfaces, but Python does not allow for this. Interfaces would make it so that an object in one of the class's

```
FUNCTION RealToByteArray
VAR_OUTPUT
   msg : ARRAY[0..7] OF USINT;
END_VAR
VAR_INPUT
   value1 : REAL;
   value2 : REAL;
END_VAR
VAR
   pt1 : POINTER TO ARRAY[0..3] OF USINT;
   pt2 : POINTER TO ARRAY[0..3] OF USINT;
   i : INT;
END_VAR

pt1 := ADR(value1);
pt2 := ADR(value2);
FOR i:=0 TO 3 DO
   msg[i] := pt1^[i];
   msg[i+4] := pt2^[i];
END_FOR;
```

Code Block 4.1: Write REAL to byte-array

```python
def read_from_bus(self) -> can.Message:
    if not self._timed_out and self._bus:
        m = self._bus.recv(timeout=2.0)
        if m is None:
            logging.error(f"Reading from CAN channel
                {self._channel} timed out, reading disabled")
            self._timed_out = True
        return m
    if self._timed_out or not self._bus:
        time.sleep(0.01)
    return None
```

Code Block 4.2: Read from bus

fields could be swapped with an object of another class as long as this class implemented the same interface. Code block 4.3 contains the function in the CANConnection class that is used to pass a received message on to the appropriate mapper.

```python
def read(self):
    self.last = False
    # reads until the last message in the
    #iteration is received
    while not self.last:
        msg: can.Message = self.read_from_bus()
        if msg is not None:
            # the message_mapper returns true if it
            # receives the last message
            self.last = self.sensor_data.message_matcher(msg)
        else:
            self.last = True

    self.sensor_data.timestamp = time.time()
    self.sensor_data.timed_out = self._timed_out
    return self.sensor_data
```

Code Block 4.3: Read CAN

The message mapper for the data that is received from the PLC is displayed in code block 4.4. The mapper places the received values in the correct field in the object based on the id of the message.

```python
def message_matcher(self, msg: Message) -> bool:
    match msg.arbitration_id:
        case 100:
            self.plc_state = byte_to_int(msg.data[0:4])
        case 110:
            self.wob.sensor_1 = byte_to_float(msg.data[0:4])
            self.wob.sensor_2 = byte_to_float(msg.data[4:8])
        case 120:
            self.wob.sensor_3 = byte_to_float(msg.data[0:4])
            self.wob.avg = byte_to_float(msg.data[4:8])
        case 130:
            self.pressure_circulation_system =
                byte_to_float(msg.data[0:4])
            self.temp_circulation_system =
                byte_to_float(msg.data[4:8])
        case 140:
            self.interlock.switch_mapper(byte_to_int(
               msg.data[0:4]))
        case 150:
            self.pump_set_point = byte_to_float(msg.data[0:4])
            self.hoisting_height =
                byte_to_float(msg.data[4:8])
        case 160:
            self.top_drive_rpm = byte_to_float(msg.data[0:4])
            self.top_drive_torque =
                byte_to_float(msg.data[4:8])
        case 170:
            self.top_drive_rpm_set_point =
                byte_to_float(msg.data[0:4])
            self.top_drive_torque_set_point =
                byte_to_float(msg.data[4:8])
        case 199:
            return True
    return False
```

Code Block 4.4: Message mapper

4.3 Sending

Preparing messages for sending is handled in much the same way as receiving. The class for the CAN connection has no idea how to structure the message for sending, but the object that is to be sent does. The CAN connection will call on this function to structure the new message.

```python
# Base class for data that is sent to the plc
class BaseControlData:
    def __init__(self):
        self._control_mode: int = 1
        self._can_list = []

    @property
    def control_mode(self):
        return self._control_mode

    @control_mode.setter
    def control_mode(self, value):
        if self._control_mode != value:
            self._control_mode = value
            a = bytearray(8)
            self.write_int_to_byte_array(a,
                self.control_mode, 0)
            self._can_list.append((200, a))

    # Creates a new message list and returns the old one
    def to_can(self) -> list[tuple[int, bytearray]]:
        values = self._can_list
        self._can_list: list[(int, bytearray)] = []
        return values
```

Code Block 4.5: Control Data

In code block 4.5 the updated values gets added to the `can_list` when they are different from the previous value. The list gets cleared when the `to_can` function is called. This makes sure that when the drill rig is controlled manually, the commands only gets sent once. Use of bandwidth is therefore reduced, and this behaviour is also required when starting processes that are solely controlled by the PLC, such as activating the stabiliser (a mechanism holding the BHA in the initial phase of the drilling). It would not make sense to send a message for running the stabiliser on every iteration. As shown in code block 4.5 when the `control_mode` gets set, the setter functions gets called. The setter checks the new value against the old value and stages a message for sending in the `can_list`.

4.4 Interlocks and Limit Switches

The drill rig has a number of switches that will indicate if something is wrong in the drilling process, for example that a door is open. These boolean values need to get transmitted from the PLC on each iteration, but it would be quite wasteful to send each of them in a separate message. All of these values are therefore put into a single integer by shifting the bit into the appropriate place in the integer. The code for converting an integer back to multiple boolean values is displayed in code block 4.6. Logical AND is used for separating out the correct bit, and is shifted right to make the result `0" or1`". The conversion to a boolean type happens if needed at a later stage.

```
class Interlock:
   def __init__(self):
      self.front_door: int = DoorSensor.CLOSED
      self.rear_door: int = DoorSensor.CLOSED
      self.limit_hoisting_bottom: int =
         LimitSwitch.INACTIVE
      self.limit_hoisting_top: int = LimitSwitch.INACTIVE
      self.limit_stabilizer_hoist_side: int =
         LimitSwitch.INACTIVE
      self.limit_stabilizer_opposite_hoist: int =
         LimitSwitch.INACTIVE
      self.hoisting: int = DeviceState.DISABLED
      self.pump: int = DeviceState.DISABLED

   def switch_mapper(self, switch_values: int):
      self.front_door = switch_values & 1
      self.rear_door = (switch_values & 2) >> 1
      self.limit_hoisting_bottom = (switch_values & 4) >> 2
      self.limit_hoisting_top = (switch_values & 8) >> 3
      self.limit_stabilizer_hoist_side = (switch_values &
         16) >> 4
      self.limit_stabilizer_opposite_hoist =
         (switch_values & 32) >> 5
      self.hoisting = (switch_values & 64) >> 6
      self.pump = (switch_values & 128) >> 7
```

Code Block 4.6: Interlock class

The process for writing boolean values to a single integer is almost the exact opposite of the process described in code block 4.6. Bits get shifted left into their corresponding place in the integer.

4.5 Downhole Sensor

The DHS is able to produce new sensor values much faster than they can be processed by the PC. The values that are received from the DHS are therefore the average of the 25 last measurements. This is shown in code block 4.7 and 4.8.

```
if (icm20948.quatDataIsReady() &&
    icm20948.accelDataIsReady()) {
      icm20948.readAccelData(&now_accel.x, &now_accel.y,
          &now_accel.z);
      avg_accel.x += now_accel.x;
      avg_accel.y += now_accel.y;
      avg_accel.z += now_accel.z;

      icm20948.readQuatData(&now_quat.w, &now_quat.x,
          &now_quat.y, &now_quat.z);
      avg_quat.w += now_quat.w;
      avg_quat.x += now_quat.x;
      avg_quat.y += now_quat.y;
      avg_quat.z += now_quat.z;
      numQuatAccel++;

      if (icm20948.magDataIsReady()) {
         icm20948.readMagData(&now_mag.x, &now_mag.y,
             &now_mag.z);
         avg_mag.x += now_mag.x;
         avg_mag.y += now_mag.y;
         avg_mag.z += now_mag.z;
         numMag++;
      }

   }
```

Code Block 4.7: DHS read current values

```
    //Write to CAN every 25 iterations
    if (numQuatAccel >= numberOfValues) {
       send_float_can(300, avg_accel.x /
           (float)numQuatAccel);
       send_float_can(301, avg_accel.y /
           (float)numQuatAccel);
       send_float_can(302, avg_accel.z /
           (float)numQuatAccel);

       send_float_can(310, avg_quat.w /
           (float)numQuatAccel);
       send_float_can(311, avg_quat.x /
           (float)numQuatAccel);
       send_float_can(312, avg_quat.y /
           (float)numQuatAccel);
       send_float_can(313, avg_quat.z /
           (float)numQuatAccel);

       send_float_can(320, avg_mag.x / (float)numMag);
       send_float_can(321, avg_mag.y / (float)numMag);
       send_float_can(322, avg_mag.z / (float)numMag);
       send_float_can(399, 0.0);
       numMag = 0;
       numQuatAccel = 0;
       avg_accel.clear();
       avg_mag.clear();
       avg_quat.clear();

    }
```

Code Block 4.8: DHS sending of average values

Taking the average of the last 25 values when sending resulted in 100 messages being sent per second. This means that a complete set of values is processed by the PC approximately every 100 ms.

Chapter 5

Logging

Logging of data from the DHS is a requirement for the competition and is done using an SQLite database. This chapter will go through the implementation of a logger in Python, including design choices and considerations. It will cover how it interacts with the rest of the software architecture, as well as how it was stress tested to ensure that it can cope with logging intensities at or exceeding the needs of the application.

5.1 Implementation

5.1.1 Technology Considerations

The logger uses SQLite as its database management system (DBMS). We chose this DBMS due to its ease of use, and because our application does not need a custom-tailored engine with pre-eminent performance. The results from the stress test, which are found in chapter 5.2.2, show that SQLite also is suitable for our logging purposes.

5.1.2 Python Implementation

The logging functionality of the application is implemented in its own class, of which a single object is instantiated inside the main application. This logger object has methods corresponding to each data object we want to log in the database. Using an SQL schema, we create the three tables in which the database log entries are stored. The information that is logged in the database is the values from either a PLC sensor data object, a downhole sensor object, or a user input data object. The data objects' value fields, stored as instance variables, correspond to their own column in the SQLite database.

Logging is done through the main application object. Downhole sensor and user input logging is done inside the `read_DHS()` and `read_user_input()` methods respectively, through a call to the `Logger` object that is initialised when an object of the `Main` class is instantiated. PLC data logging is done immediately following the reading of PLC data from the CAN bus. Each of the data objects has its own log method in the `Logger` class.

Interacting with the SQLite database is not instantaneous, and we therefore optimise the methods responsible for logging data by adding a buffer. This is done because calling the SQLite library every time the logger is called, combined with the frequency at which we receive data, would make time spent on logging longer. This would each event loop run cycle take longer, which runs the risk of the system becoming unresponsive. Each of the methods has its own buffer, and when the buffer count exceeds a set maximum buffer size, all buffer entries are stored on disk by inserting them into the database.

The main methods for the `Logger` class are shown in code snippet 5.1. The logger is run by setting the boolean `run_logger` instance variable to `True`. The logging speed and log buffer size are set in the config file. Logging speed is a way to limit the number of entries that are put in the database. For instance, if the logging speed is 10, it will only log every 10 data sets. The log method flushes the buffer when the buffer exceeds the log buffer size. The method that flushes the buffer stores the buffer data to disk by spawning the static method `write_to_db()` in its own thread.

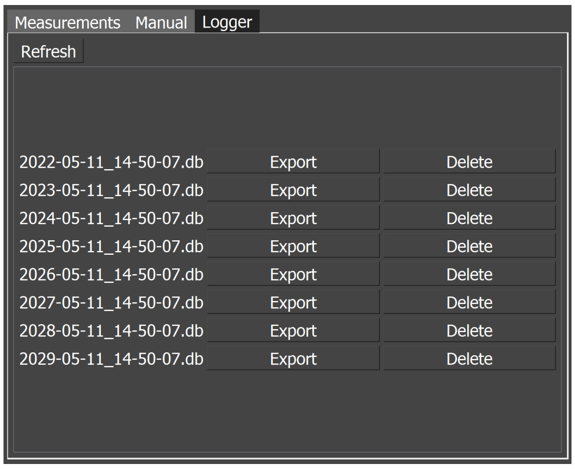
```python
class Logger:
   def log_PLC(self, item: PLCSensorData):
      if not self._run_logger:
         return
      if self._PLC_counter >= self._logging_speed:
         self._PLC_counter = 0
         self._PLC_buffer.append(copy.copy(item))
         if len(self._PLC_buffer) >= self.buff_size:
            self.flush_PLC_buffer()

      self._PLC_counter += 1
   def flush_DHS_buffer(self):
      values = self._DHS_buffer
      self._DHS_buffer = []
      t = threading.Thread(target=self.write_to_db,
          args=(values, self.filename))
      t.start()

   @staticmethod
   def write_to_db(values: list, filename: str):
      try:
         conn = connect(filename)
         for value in values:
            value.write_to_db(conn)
         conn.commit()
         conn.close()
      except OperationalError as e:
         logging.error("Database cannot sustain logging
             speed")
         logging.error(e)
```

Code Block 5.1: Code for logging.

5.1.3 Log File Management and Export

The application has start/stop functionality, and each time the application is started, it creates a new database file. This causes the data from the different drilling runs to be stored separately. The name of the file is the time at which the run commenced.

The log file management tab is the application's interface with these log files. The user interacts with this interface through a tab in the main window, as shown in figure 5.1. A record of all the database log files is displayed, and the user can also delete or export files. Export functionality is handled by a custom SQLite-to-CSV converter, as shown in code snippet 5.2.



2022-05-11_14-50-07.db	Export	Delete
2023-05-11_14-50-07.db	Export	Delete
2024-05-11_14-50-07.db	Export	Delete
2025-05-11_14-50-07.db	Export	Delete
2026-05-11_14-50-07.db	Export	Delete
2027-05-11_14-50-07.db	Export	Delete
2028-05-11_14-50-07.db	Export	Delete
2029-05-11_14-50-07.db	Export	Delete

Figure 5.1: GUI log management tab

```python
def write_db_rows(filename: str, column_names: list[str], rows: list[sqlite3.Row]):
    with open(filename, "w", newline='') as f:
        writer = csv.writer(f, delimiter=';')
        writer.writerow(column_names)
        writer.writerows(rows)
```

Code Block 5.2: Export to CSV file.

5.2 Stress Testing the Logger

As with any logging solution, it was required that the logger was stress tested to ensure that there is not any data loss that can end up obscuring the data. The tests were run under different conditions, including with high speed, various durations, and different-sized databases.

5.2.1 Stress Testing Methodology

For inserting data into the database, the logger can expect to insert 10 data sets from the downhole sensor, and 10 data sets from the the rest of the sensors per second. For the stress testing, it is necessary to simulate as close to actual production load for the computer simulating, and we will want to try a set of various amount of data sets per second, over a various different time ranges. While it is not expected that the rig will be running for very long at a time, and will not be getting more than a total of 20 values per second, the logger will be tested far beyond its required operating speed to ensure that we have good margins.

5.2.2 Results

Below you can see table 5.1 detailing the results of the stress testing. The logger was run at frequencies between 10Hz and 300Hz, and duration between 10s and 12h to test a wide array of conditions to figure out where the bottleneck would be when run beyond its operating speed. In reality it will end up running for 10–30 minutes at around 20Hz under normal operating conditions.

	10s	60s	10m	1h	3h	12h
10Hz	Success	Success	Success	Success	Success	Success
20Hz	Success	Success	Success	Success	Success	Success
50Hz	Success	Success	Success	Success	Success	Success
100Hz	Success	Success	Success	Success	Success	Success
200Hz	Success	Success	Success	Success	Success	Partial
300Hz	Success	Success	Success	Partial	Fail	Fail

Table 5.1: Results from stress testing the logger.

The only conditions under which the test was unable to finish as intended was when it attempted to run the test for 300 Hz over 3 and 12 hours, where it then failed after 97 % completion. 2.6 % of logging entries were dropped. When running the database with a 200 Hz load for 12 hours, the database managed 99.8 % completion, missing .18 % of entries after 12 hours. Similar results occurred when attempting to run at 300Hz for 1

```
with open(os.path.join("logger", "schema.sql")) as file:
    conn.executescript(file.read())
    conn.commit()

s, = conn.execute("SELECT COUNT(did) FROM
    dh_sensor").fetchone()
messages_count = frequency * seconds

start_time = time.time()
wanted_end_time = time.time() + seconds
for i in range(messages_count):
    lg.log_DHS(generate_dh_data())
    time.sleep(max(0, (wanted_end_time -
        time.time())/(messages_count - i)))
end_time = time.time()
time.sleep(0.5)

n, = conn.execute("SELECT COUNT(did) FROM
    dh_sensor").fetchone()

print(f"logged {n-s} entries in
    {end_time-start_time:.2f} seconds.")
```

Code Block 5.3: Test query & method

hour where it was missing a small amount of data, and finished less than a minute later. These tests were also run with databases that were empty, and databases that had 1 GB or more data in them, and we got similar results between both, concluding that the sizes that we are dealing with will not be a problem for the logging.

From this, it can be concluded that the bottlenecks start appearing at 15 times the speed that is going to be expected when run for more than one hour. The test running at 200Hz for 12 hours was also slightly too slow, albeit it ended up logging everything a short time after the 12 hours had gone. As the margins between what the stress test concluded and the actual operating speed are this large, it can be assumed that there will not be any bottlenecks with the logging during operating conditions.

See code block 5.3 for how we ran our tests.

Chapter 6

Control System

A major part of making the rig operate is to design and implement a control system, both automatic and manual. As the rig remains unfinished by this thesis deadline, automatic drilling is yet to be implemented, but all the pieces are in place for it. Sending and receiving data is working, and there is a manual steering page in place for all available values.

6.1 The Control Process

Figure 6.1 shows the interface for controlling some parts that can be controlled manually. It is created to allow for testing of both the installed equipment and the control system itself, but is not needed for actually running the drilling process. See chapter 9.1 for plans to implement automatic control.

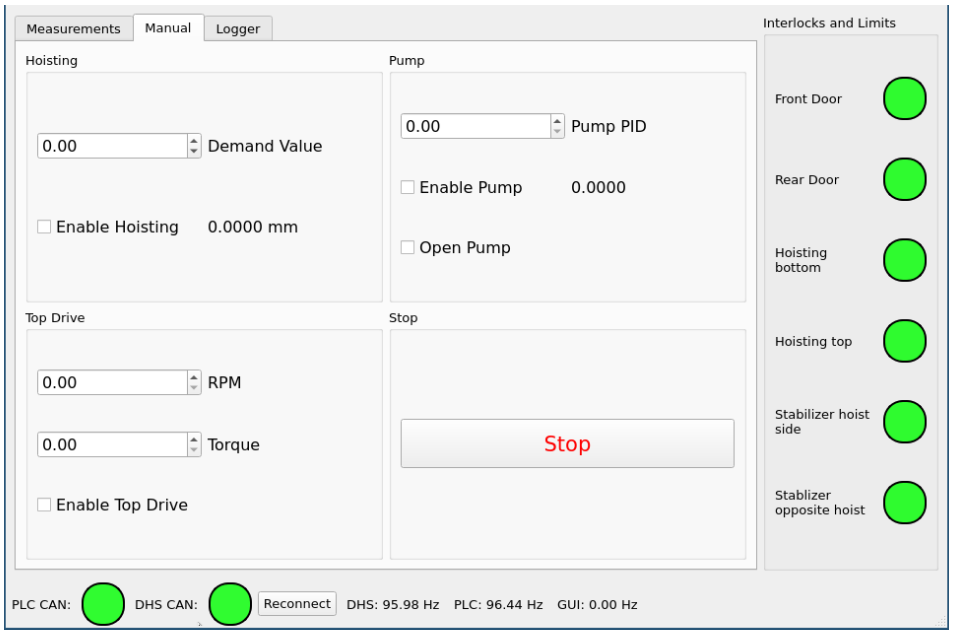


Figure 6.1: Manual control

When a value is adjusted in the GUI, it triggers an event that will pass the message on to the controller part of the system. Code block 6.1 shows how the different values in the GUI are collected and put into an object and then onto a `Queue`. All of the different inputs on the manual page are connected to this method so that if one input changes, all of them get read. This makes for more code to execute for a single button press, but it prevents having to write separate logic for each button.

At the other end of the `Queue`, the object is picked up and placed into the controller. When called for, the controller will place this object's values into a new object of the class `ManualControlData`. This class inherits from `BaseControlData` shown in code block 4.5 and is extended with the functionality required to generate CAN messages containing the data for manual control. When the CAN message has been generated, the message is handled by the `CANPLCConnection` and sent to the PLC.

```python
def get_inputs(self):
  self.inputs.hoisting_enable =
     self.hoisting_enable.isChecked()
  self.inputs.hoisting_demand_value =
     self.hoisting_demand_value.value()

  self.inputs.pump_open = self.pump_open.isChecked()
  self.inputs.pump_enable = self.pump_enable.isChecked()
  self.inputs.pump_PID = self.pump_PID.value()

  self.inputs.top_drive_enable =
     self.top_drive_enable.isChecked()
  self.inputs.top_drive_rpm_set_point =
     self.top_drive_rpm_set_point.value()
  self.inputs.top_drive_torque_set_point =
     self.top_drive_torque_set_point.value()

  self.input_queue.put(self.inputs)
```

Code Block 6.1: Read inputs in GUI

6.2 PLC

When the PLC picks up the data that is received from the CAN interface, it first gets converted back to the more useful data types like integers and floating-point numbers. This process is described in chapter 4.1.1. The values are then assigned to the appropriate variables.

Control of the different motors and actuators is accomplished by linking variables in the PLC code to the outputs that are connected to these devices.

```
Outputs.DO_PumpEnable := DINT_TO_BOOL(Process.PumpEnable);
Outputs.AO_PUMP :=
   REAL_TO_INT(Process.PumpPID/PumpScaling*IntMax);
```

Code Block 6.2: Set output variable

Code block 6.2 shows how the motor for the pump is controlled. The variable `Process.PumpEnable` is the value that has been received from the PC and thereafter validated. `Outputs.AO_PUMP` is the variable that is linked to the electric output on the PLC that is connected to the motor controller. To set the voltage of the analog output the `Process.PumpPID` value is first divided by the maximum value that can be received from the PC. This value is then multiplied the maximum value of an integer. The resulting value is then used by the PLC to determine what voltage to send to the output.

```
//Only downwards when top switch is pressed
IF (Inputs.DI_HOISTING_LIM_TOP = FALSE) AND
    (Process.HoistingDemandValue > 0) THEN
   Outputs.AO_HOISTING := 0;
//Only upwards when bottom switch is pressed
ELSIF (Inputs.DI_HOISTING_LIM_BOTTOM = FALSE) AND
    (Process.HoistingDemandValue < 0) THEN
   Outputs.AO_HOISTING := 0;
ELSE
   Outputs.AO_HOISTING :=
       REAL_TO_INT(Process.HoistingDemandValue /
       HoistingScaling * IntMax);
END_IF
```

Code Block 6.3: Set hoisting speed

To prevent damage to equipment and/or people it is necessary to implement certain safety mechanisms. Code block 6.3 shows how the PLC only allows the hoisting mechanism to travel upwards if the bottom limit switch is pressed and opposite. It is impossible for an operator using the GUI to override this.

Chapter 7

Graphical User Interface

The GUI was a central component of our thesis and something we wanted to focus quite a bit of effort on. It was important from the beginning to be able to design a proper GUI built with the operator in mind, and being as efficient as possible. Much of the motivation to do a good GUI where we follow industry principles is due to the competition organiser adding human machine interfaces (HMI) as a judging criteria, motivating students to learn about this, and creating good interfaces. To further this goal, we conducted a literature review in which we reviewed both companies in the industry, but also Drillbotics' own guidelines, as well as various other sources for both design and alarm management. In this chapter we'll begin with doing a structured literature review, before designing a GUI using the principles and information learnt from the review.

7.1 Literature Review: GUI

The drill for which the graphical user interface (GUI) will be developed for is a scaled-down version of a type of machine used in the industry, namely a drill rig. Therefore this demands a need to consider how GUIs are made properly in the industry. The drill and the data from it will come continuously while it's running, and it is required that all of the relevant information is displayed properly to the operator. This includes how to display any and all incoming values, figuring out which values are more relevant, designing a way for the system to alert the operator in unforeseeable situations and more. Another consideration in the design will be the “human vs automation” factor, e.g. how much automation there should be as opposed to systems where human intervention should be possible.

Looking at the guidelines for the competition it states the following: “Visualisation of the processes (automation, optimisation, drilling state, etc.) should be intuitive and easily understood by the judges, who will view this from the perspective of the driller operating a rig equipped with automated controls.” [5] This means figuring out how this is done in the industry for drilling operators is important, and the GUI will take inspiration from that. We’ll also conduct some short surveys with the petroleum students at the University of Stavanger on how they’d like to final product to look like.

In the past, the only method for drilling has been with automated control. A new element for the competition this time around is the human factor component. Marcin Nazaruk, chair of the SPE Human Factors Technical section had this to say about the inclusion of the human factor: “Asking students to not only read about human factors but actually to integrate HF requirements into their designs is an important milestone for the drilling industry. When things go wrong on a rig, too often we see examples of the operator getting blamed for what HF practitioners may consider poor system design that leads to the equipment operator not having the needed information to diagnose a problem and intervene effectively” [12]. Finding the correct balance between automation and human involvement means fewer problems and better long-term operation of any system. While automation and automated systems excel at finely tuned tasks, they have no way to tackle unforeseen issues. You can program your system to have exceptions for as many issues as possible, but often there will be problems that nobody foresees, especially with mechanical hardware.

7.1.1 Defining Research Questions

In particular, the focus in this review will be these three research questions (RQ):

- RQ 1: How do existing GUIs look, and what could be learnt from them?
- RQ 2: What design principles should be used when designing the GUI?
- RQ 3: How much automation as opposed to human interaction should be implemented?

Methodology and Review Protocol

For identifying sources for this literature review, a few different sources of information are used. Mainly we will review the recommended resources given in the Drillbotics Guidelines [5], which will be our primary studies. Other relevant searches and studies we'll review are: on recommendation from our thesis supervisor we will review a former student's bachelor thesis on Human Machine Interfaces, and lastly, we will also be reviewing various articles and studies found from Google and Google Scholar on the topic. As we have a primary source of information through Drillbotics, the rest of the sources we'll review will be secondary. Therefore in choosing which sources to use, we'll try to find information that strengthens or builds upon what we learn through our primary studies. Along with this, we'll also review most sources that seem specifically relevant to our RQs through searches using keywords and formulations taken directly from our RQs. For full disclosure, due to time constraints, long sources (e.g. 25+ pages long will be "scanned" for relevant content, and not reviewed in their entirety. We will systematically go through each source and take out any relevant info, and consolidate what was learned in the conclusion. See some of the sources used below.

Automation vs Manual Operation

Taking a look at Fitts' list from 1951 there are a few items that are relevant to consider for the GUI to be best utilised in actual scenarios from a human–machine interface perspective. In particular, there are three main points from Fitts' list that one could argue would make long-time performance better, and make it worthwhile having an operator. These three points are *Ability to improvise and use flexible procedures*, *Ability to reason inductively* and lastly *Ability to exercise judgement.* The other items on Fitts' list have become more outdated as the list was originally made in 1951. [17] During normal operation, most industrial systems are most efficient when automated. However what the three previously mentioned points elude to is the benefit of having someone to rapidly tackle unexpected scenarios that slow down, or halt operation completely. The human also exists in the operation to monitor and improve upon the operation. While machines are increasingly able to do these things better due to machine learning and AI, humans still have an advantage in these areas for most systems. See the complete Fitts' list from 1951 below.

Humans appear to surpass present-day machines in respect to the following:	Present-day machines appear to surpass humans in respect to the following:
1. Ability to detect a small amount of visual or acoustic energy	7. Ability to respond quickly to control signals and to apply great force smoothly and precisely
2. Ability to perceive patterns of light or sound	8. Ability to perform repetitive, routine tasks
3. Ability to improvise and use flexible procedures	9. Ability to store information briefly and then to erase it completely
4. Ability to store very large amounts of information for long periods and to recall relevant facts at the appropriate time	10. Ability to reason deductively, including computational ability
5. Ability to reason inductively	11. Ability to handle highly complex operations, i.e. to do many different things at once
6. Ability to exercise judgement	

Table 7.1: Fitts' List

Designing for humans with automation principles is not easy as it's not an either/or distribution of work between the two. While pure automation has been the desired configuration for the Drillbotics competition for the previous eight years, switching over now to a hybrid human automation format, requires thorough thinking and review. The design should be based on convention and researched principles. In 2017, John D. Lee et al, laid out fifteen principles of human automation in their book, “Designing for People: An Introduction to Human Factors Engineering”. The principles laid out here give a good foundation of knowledge into the essentials for human factor engineering.

The list is as follows [11]:

1. Define and communicate the purpose of automation.
2. Define and communicate the operating domain.
3. Design the role of the person and automation.
4. Simplify the mode structure.
5. Make trustable and polite.
6. Signal inability to satisfy role.
7. Transparency - Keep the person informed.
8. Avoid accidental activation and deactivation.
9. Keep the person in the loop.
10. Support smooth re-entry into the loop.
11. Make automation directable.
12. Make the automation flexible and adaptable.
13. Consider adaptive automation.
14. Keep people trained
15. Consider organizational consequences.

Many of these make intuitive sense, while others seem intended for more specific cases. However, they all give valuable insight into principles that should followed for the design. What is clear after going through many sources is that one of the main driving factors of our design should be the importance of having a user-centred design. As stated in “Automation and autonomous systems: Human-centred design in drilling and well” on commission by the Petroleum Safety Authority Norway [15], “Experiences from the industry and the investigations strengthen the need for user-centred development and a focus on HF”. Another vital part they point out is that “IEA/ILO should be a key reference concerning the introduction of new technology such as automation and digitalisation.”

IEA/ILO (2020) refers to a report called “Principles and guidelines for human factors/ergonomics (HFE) design and management of work systems” and is made by the International Labour Organization (ILO) and the International Ergonomics Association (IEA). [10] The report also lists five principles for HFE design and management of work systems:

1. Ensure worker safety, health, and wellbeing in the optimization of work systems as a top priority.
2. Design and manage work systems to ensure organizational and worker alignment, continuous evaluation and learning, and sustainability.
3. Create a safe, healthy, and sustainable work environment from a holistic perspective, understanding and providing for human needs.
4. Account for individual differences and organizational contingencies in the design of work.
5. Make use of collective, trans-disciplinary knowledge and full participation of workers for designing systems, detecting problems, and creating solutions for HFE in work systems.

The focus of these principles and what should always be the main focus of the design is as mentioned the paramount importance of the human being in the centre of any design, and safety always being the first priority. While the system might be autonomous, the human should be aware of anything going on and be able to intervene and interact with the system to help lessen the impact of eventual issues.

There are of course always drawbacks and eventualities and *ironies* to consider. In the book *Ironies of Automation*, by L.D. Bainbridge [1] lists some ironies in the design of automated systems. While most are worried about human error and use automation to mitigate this, we need to consider the fact that the system designer(s) are human as well, which is something called "human-induced errors". What this essentially means is that human error can occur at the design stage, making the skills of the operator irrelevant, and making the need for robust testing of the design an important factor.

Another important thing to note is that if you automate too much, the remaining work left for the human operator may be so unsatisfying, and seemingly uninteresting that the operator's skill set would diminish over time, as well as the lack of enthusiasm to want to do the job well. There are also many factors to consider when considering that the human operator would unexpectedly have to take over. As mentioned if the system is designed to be automated, with human intervention only with unexpected error, the skills of the human operator will deteriorate over time, this is especially important when realising that the situation where the automation fails will likely be difficult to handle and requires a sharp skill set. Any operator no matter how skilled will always need some time to "switch over" from monitoring "mode" to active control "mode". We also know that it is difficult to monitor effectively over an extended amount of time, which is the same reason air traffic controllers generally work short amounts of time, with many breaks during their shifts, to keep efficiency up, and errors down. Depending on how critical a rig is to run without failure, one should consider the cost-efficiency and whether a similar system should be deployed, even if it ends up costing more in salary, although strictly not a factor for our purposes.

Alarm Management

Proper alarm management, the art of making sure that an operator is immediately aware of unexpected operations is also of utmost importance. Ensuring that we facilitate accurate and timely fault prompting and diagnosis to operators is vital to any actively running system, especially industrial [9]. Poor design is a large cause of many major incidents. The Health and Safety Executive from the UK points out five principles to ensure proper alarm management and workplace safety.

- Alarms should direct the operator’s attention towards plant conditions requiring timely assessment or action;
- Alarms should alert, inform and guide required operator action;
- Every alarm should be useful and relevant to the operator, and have a defined response;
- Alarm levels should be set such that the operators have sufficient time to carry out their defined response before the plant condition escalates;
- The alarm system to accommodate human capabilities and limitations;

Alarm management is one of the primary concerns when designing an industrial system. Any time you have a system with moving parts, high amounts of pressure, running engines, etc. you need to be very sure that you’re properly warned about unintentional operation as it can potentially be very dangerous. Depending on the system running autonomously it can lead to anything from improper operation to potentially being fatal.

The principles listed above also go quite hand-in-hand with more general design principles. Keeping an as aesthetic and as minimal design as possible ensures that there is less clutter for an operator to get lost in while trying to get an accurate idea of the situation. The implication here is that while you may have a lot of information available to display as a designer, more is not necessarily better, and you need to carefully consider what is more important. An alternative is to have a lot of information in front of you, but then your alarm management system needs to be of such quality that you'll easily be able to swiftly navigate your interface towards what is important, as suggested above. Alarms should be as infrequent as possible so that when important alarms actually do come in, it gets reacted on more swiftly.

General Design Heuristics

There is also the general design to consider for creating a good interface. Over 25 years ago, Jakob Nielsen at Nielsen Norman Group listed ten usability heuristics for user interface design. While the world has advanced rapidly and at an unprecedented rate since then, most of the principles he made back then are still as valid and relevant as they were then. While the heuristics he made were intended for general design, many of them are relevant for industrial usage on a drilling rig. The ten Usability Heuristics for User Interface Design are as follows: [13]

1. Visibility of system status:
    - Perhaps particularly relevant for anyone monitoring mechanical equipment, visibility of status, and eventually visibility of *alarms* is of particular importance as seconds can matter a lot.
2. Match between system and the real world
    - The design should use typical conventions and stick to words, phrases and concepts known to any operator, and not difficult to understand internal wordings. Most relevant for us here is the keep a normal layout with easily discernible sections.
3. User control and freedom
    - For actions done by mistake, users need a clearly marked "exit" like undo/redo.
4. Consistency and standards

- Refer to point number 2. Follow industry and platform standards.

5. Error prevention
   - Good error messages are nice, but error prevention is nicer. Warnings when things start going wrong, but before it has gone wrong is desired. Stuff like yellow warning lights could be an example of something we can implement.
6. Recognition rather than recall
   - The user should not need to recall something from one part of the interface to another. Necessary information should be readily available and easily visible.
7. Flexibility and efficiency of use
   - Eludes to having shortcuts that can make an experienced user/-expert more efficient. Not very relevant here.
8. Aesthetic and minimalist design
   - Keep irrelevant information out of the GUI design. "Extra" information competes with the relevant information in drawing the users attention.
9. Help users recognise, diagnose, and recover from errors
   - Express error messages in easily discernible, plain language such that no one has to search for an error code. Offer a solution.
10. Help and documentation
    - No explanation needed.

Some are more important than others, and some we have already discussed. But use conventions, make things as easily readable as possible, such that someone inexperienced could use it.

Real World Example

There are not a lot of examples of real-world drilling interfaces available online, however, some companies release more specific specs and interface designs. Schlumberger through their daughter company Cameron has a brochure showcasing their digital drilling control system. While a Drillbotics project is nowhere near as sophisticated and complicated as a real control system used by one of the oil giants, there is still knowledge to take from them. What is immediately noticeable when looking at their design is how much everything pops out, as long as the operator is paying attention, they will immediately notice the change in a light coming on, i.e. turning from green to red to indicate an error, due to contrast. [2]

More sophisticated control systems like the ones used in the industry usually have control systems similar to the one above which is not realistic to have for a small-scale project like Drillbotics. Their principles are still present as we can see throughout their design which we can take inspiration from. Mainly we want to give the operator maximum freedom and a good user experience. Improved user experience leads to happier workers, and better production and helps ease down the number of mistakes in a system. We can see these principles clearly implemented in the design used by Schlumberger.

Conclusion

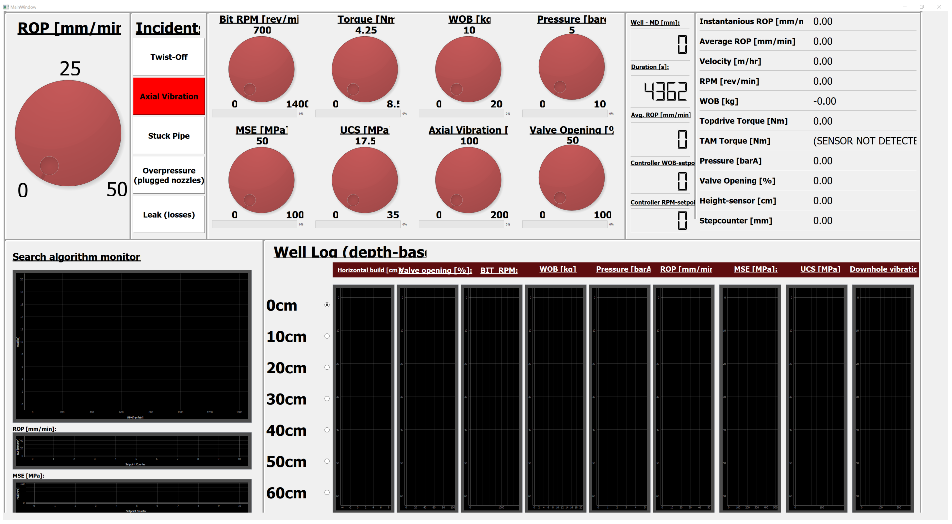
There are a lot of important principles, heuristics, and observations to consider when attempting to design the ideal interface. Generally, there are a lot of good rules to follow to create a good design, if the interface can check off all the boxes when finally implemented, it should in theory be good. Of particular importance is the idea that the operator/worker is in the centre of the design, and their wishes, granted they are realistic, are paramount. Therefore keeping the worker in the design loop so we can ensure good design both from a designer's and an operator's point of view.

The most important system in the design will be how we communicate things running in non-ideal conditions, and having proper alarm management to manage it.

7.2 Designing and Implementing The GUI

7.2.1 Former University of Stavanger Drillbotics GUI

The large focus and motivation on doing this part will originate from the former GUI designed for University of Stavanger's Drillbotics rig in previous years. The former GUI was cluttered, non-intuitive and confusing for the user. On running the drilling application/GUI it would open four separate windows for managing from the operators perspective, and 22 terminal windows to run everything, giving the user a sign that it's poorly designed. Due to the shortcomings of the former codebase and GUI design, the group desided to redesign everything in a way we felt was more appropriate, and using proper principles for good design. The main window of the old GUI is shown in 7.1

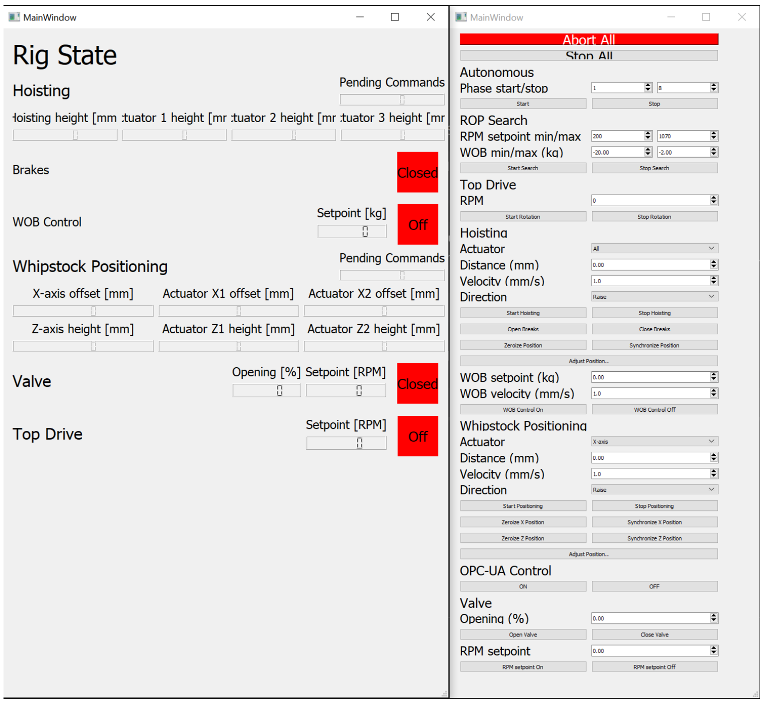


Well - MD [mm]:	0
Duration [s]:	4362
Avg. ROP [mm/min]	0
Controller WOB-setpo	0
Controller RPM-setpoi	0

Instantanious ROP [mm/n	0.00
Average ROP [mm/min]	0.00
Velocity [m/hr]	0.00
RPM [rev/min]	0.00
WOB [kg]	-0.00
Topdrive Torque [Nm]	0.00
TAM Torque [Nm]	(SENSOR NOT DETECTE
Pressure [barA]	0.00
Valve Opening [%]	0.00
Height-sensor [cm]	0.00
Stepcounter [mm]	0.00

Figure 7.1: This was the main page of the application.

After discussing with other members of the Drillbotics team, as well as talking to other petroleum students, we found that this design was suboptimal. It has some important features, but it is clear that the operator is not the first consideration when looking at this design, as nothing is optimised towards a human being able to see things fast and respond quickly.



MainWindow

Rig State

Hoisting — Pending Commands

ɧoisting height [mm ːtuator 1 height [mr ːtuator 2 height [mr ːtuator 3 height [mr

Brakes: Closed

WOB Control — Setpoint [kg]: Off

Whipstock Positioning — Pending Commands

X-axis offset [mm] | Actuator X1 offset [mm] | Actuator X2 offset [mm]

Z-axis height [mm] | Actuator Z1 height [mm] | Actuator Z2 height [mm]

Valve — Opening [%] Setpoint [RPM]: Closed

Top Drive — Setpoint [RPM]: Off

MainWindow

Abort All

Ston All

Autonomous
Phase start/stop: 1, 8

ROP Search
RPM setpoint min/max: 200, 1070
WOB min/max (kg): -20.00, -2.00

Top Drive
RPM: 0

Hoisting
Actuator: All
Distance (mm): 0.00
Velocity (mm/s): 1.0
Direction: Raise

WOB setpoint (kg): 0.00
WOB velocity (mm/s): 1.0

Whipstock Positioning
Actuator: X-axis
Distance (mm): 0.00
Velocity (mm/s): 1.0
Direction: Raise

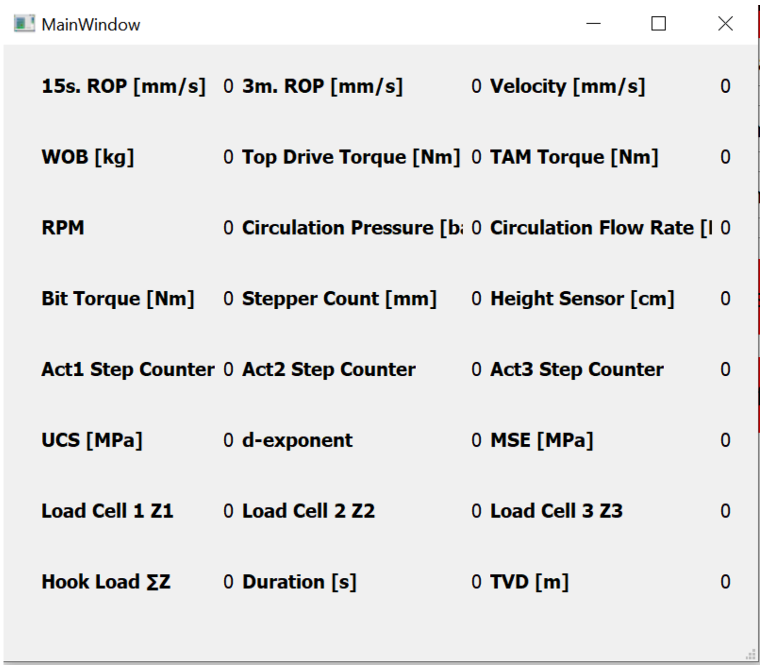
OPC-UA Control

Valve
Opening (%): 0.00

RPM setpoint: 0.00

Figure 7.2: Additional windows for manual control

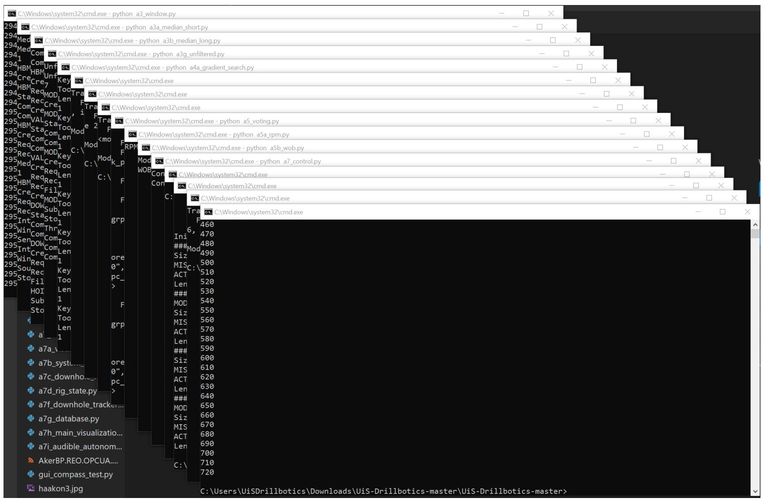
The "Rig State" window has text that gets cut off, and it is difficult to understand what its purpose is for a drill operator.



15s. ROP [mm/s]	0	3m. ROP [mm/s]	0	Velocity [mm/s]	0
WOB [kg]	0	Top Drive Torque [Nm]	0	TAM Torque [Nm]	0
RPM	0	Circulation Pressure [b	0	Circulation Flow Rate [l	0
Bit Torque [Nm]	0	Stepper Count [mm]	0	Height Sensor [cm]	0
Act1 Step Counter	0	Act2 Step Counter	0	Act3 Step Counter	0
UCS [MPa]	0	d-exponent	0	MSE [MPa]	0
Load Cell 1 Z1	0	Load Cell 2 Z2	0	Load Cell 3 Z3	0
Hook Load ΣZ	0	Duration [s]	0	TVD [m]	0

Figure 7.3: Drilling values

This 3rd window contains all the current values for each of the sensors on the drilling rig. While it's nice to have all the numbers available there is still lots of room for improvement here. As a rig operator you have a limited amount of time to detect when a value goes too far. While some of this could be detected in the main window, the lack of contrast, and lack of design makes it difficult for an operator to keep focused for longer lengths of time.


C:\Windows\system32\cmd.exe - python a3_window.py
C:\Windows\system32\cmd.exe - python a3a_median_short.py
C:\Windows\system32\cmd.exe - python a3b_median_long.py
C:\Windows\system32\cmd.exe - python a3g_unfiltered.py
C:\Windows\system32\cmd.exe - python a4a_gradient_search.py
C:\Windows\system32\cmd.exe
C:\Windows\system32\cmd.exe
C:\Windows\system32\cmd.exe
C:\Windows\system32\cmd.exe - python a5_voting.py
C:\Windows\system32\cmd.exe - python a5a_rpm.py
C:\Windows\system32\cmd.exe - python a5b_wob.py
C:\Windows\system32\cmd.exe - python a7_control.py
C:\Windows\system32\cmd.exe
C:\Windows\system32\cmd.exe
C:\Windows\system32\cmd.exe
C:\Windows\system32\cmd.exe
a7a_
a7b_syst
a7c_downho
a7d_rig_state.py
a7f_downhole_tracker...
a7g_database.py
a7h_main_visualizatio...
a7i_audible_autonom...
AkerBP.REO.OPCUA....
gui_compass_test.py
haakon3.jpg

```
460
470
480
490
500
510
520
530
540
550
560
570
580
590
600
610
620
630
640
650
660
670
680
690
700
710
720

C:\Users\UiSDrillbotics\Downloads\UiS-Drillbotics-master\UiS-Drillbotics-master>
```

Figure 7.4: CMD windows opening on running the Python program

Although it is not necessarily problematic to have all these windows opening, it does not give the user a lot of confidence in the program either. It is a lot of processes running at once, and all of them are running in separate windows. The preferred method for doing this would just be having the GUI open alone, and if you need to have more information through logs, a single window should suffice.

As mentioned after discussing the former interface with the other Drillbotics members, other petroleum students and our supervisor we all agreed there was tons of room for improvement here and thus we decided to rework the entire interface from scratch.

7.2.2 Design Process

As we had already completed a literature review on Graphical User Interface design (GUI), and had gotten some inspiration from the former GUI, we already had a rough idea of what we wanted to accomplish with the GUI, and how we wanted it to look. We reviewed the relevant sources given to us by Drillbotics in detail, and researched how major actors in the oil industry does it. We also reviewed the automation vs manual control problem on how to best design a system with an operator in its centre. The design process continued directly following the review, and we started working on a rough draft and basic ideas for what to include, and where to put it. The first *very* rough draft looked like this:

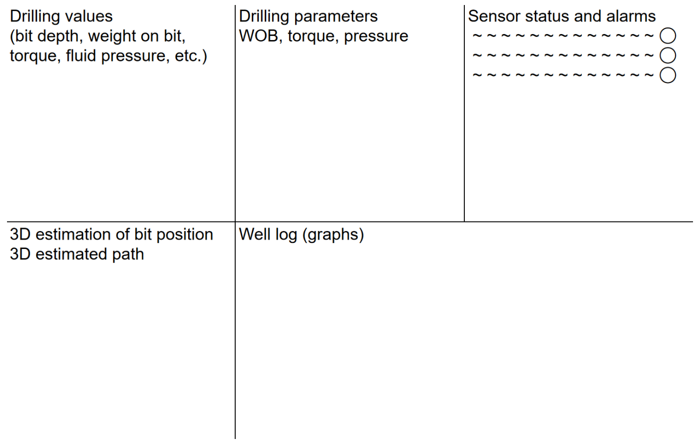


Figure 7.5: A rough first draft for how we would like the GUI to look

Figure 7.5 is our initial idea for how we wanted the GUI to look. We tried to consider the principles we researched in our literature review in chapter 7.1, to create a principled application with all the fundamental principles, and with the operator in the forefront. Essential to our GUI is the ease of use for the operator. This for us means giving the right amount of info in the correct way. It also means have proper alarm management so that the operator can respond quickly to unforeseen operations. This means having alarms (e.g. red lights flashing) and using colors to better warn if something is wrong (e.g. having the color of the numbers change given certain thresholds are exceeded).

Thanks to some help from Drillbotics member and petroleum engineer Mya Chordar we were able to take our right first draft to further develop a more complete idea. As she has experience operating industrial drill rigs she had valuable input on the design of our interface through actual experience. Together with Chordar we iterated over our sketch to provide a 2$^{\text{nd}}$ version of our draft, and together we proposed the design below:

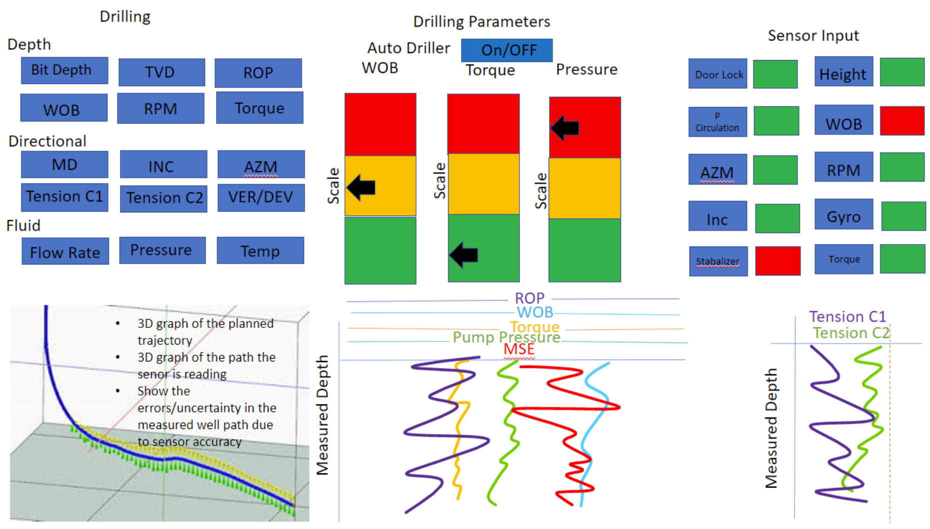


Figure 7.6: Second draft for how we would like the GUI to look made with help from Mya Chordar

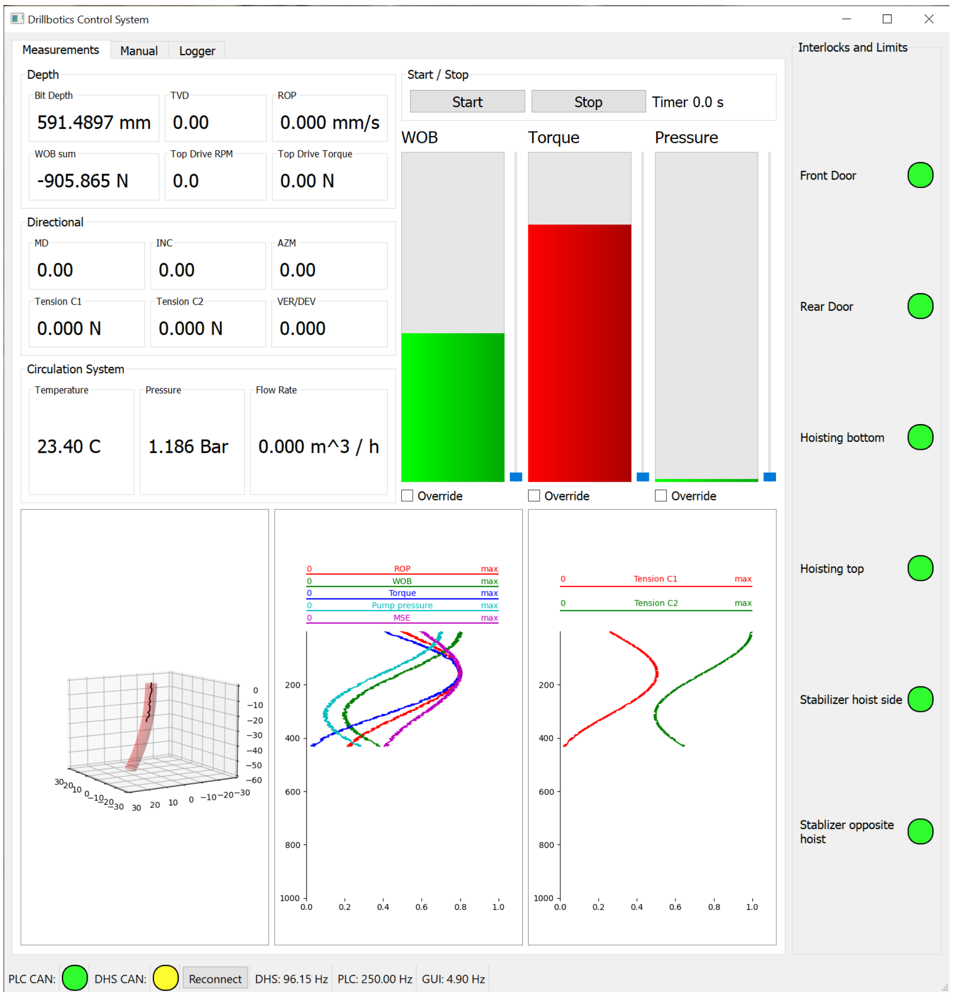
This was largely the design we ended up using when starting to put the pieces together in a PyQT5 application. More explanation into each section:

- Upper left: This *Drilling* section will contain values for all the sensors that are relevant for drilling. The initial idea was to have each of these values have certain thresholds and change colour based on whether they exceed these thresholds to easily warn the operator if some values are getting too high.
- Upper middle: These are the *drilling parameters* we have. The "auto driller" is to stop/start the autonomous process of finding coordinate X,Y,Z. Each of the gauges/meters below (WOB, Torque and Pressure) were there so that the operator could set their own levels using a meter next to them, and the colours would be our own pre-defined thresholds suggesting warning- and critical levels of operation.
- Upper right: These are our *sensor status* lights. Each sensor would have a light associated, and will turn red if we lose them, and green if they are active.
- Lower left: This is our *3D Plot* where we will first place our minimum curvature path, which is the industry standard method to find the best path to a given 3-dimensional coordinate. Layered on top of this

preliminary path, will be the current position and actual path the drill has used.

- Lower middle: This is the *well log* containing all the information a drill rig operator would want to have available history for. As the Y-axis here is the depth into the ground/block, it serves as a history for what has happened to get to the current position, and is important for a drill rig operator to know.
- Lower right: This is a 2D graph of the tension for the cables that we pull to steer the RSS.

Through discussing this with other members of the Drillbotics team and our supervisor we agreed that this would be a good design for the main window of our application. The remaining challenge here was to design this in a way that is aesthetically pleasing, while applying the knowledge from the literature review in chapter 7.1. As was discussed in the review, the interface is made for the operator, and is supposed to be as easy to use as possible. It was opted to have a dark interface to make contrasting colours easier to see for any changes in status. See the end result in figure 7.7.



Drillbotics Control System

Measurements | Manual | Logger

Depth

Bit Depth: 591.4897 mm
TVD: 0.00
ROP: 0.000 mm/s
WOB sum: -905.865 N
Top Drive RPM: 0.0
Top Drive Torque: 0.00 N

Directional

MD: 0.00
INC: 0.00
AZM: 0.00
Tension C1: 0.000 N
Tension C2: 0.000 N
VER/DEV: 0.000

Circulation System

Temperature: 23.40 C
Pressure: 1.186 Bar
Flow Rate: 0.000 m^3 / h

Start / Stop

Start | Stop | Timer 0.0 s

WOB | Torque | Pressure

☐ Override ☐ Override ☐ Override

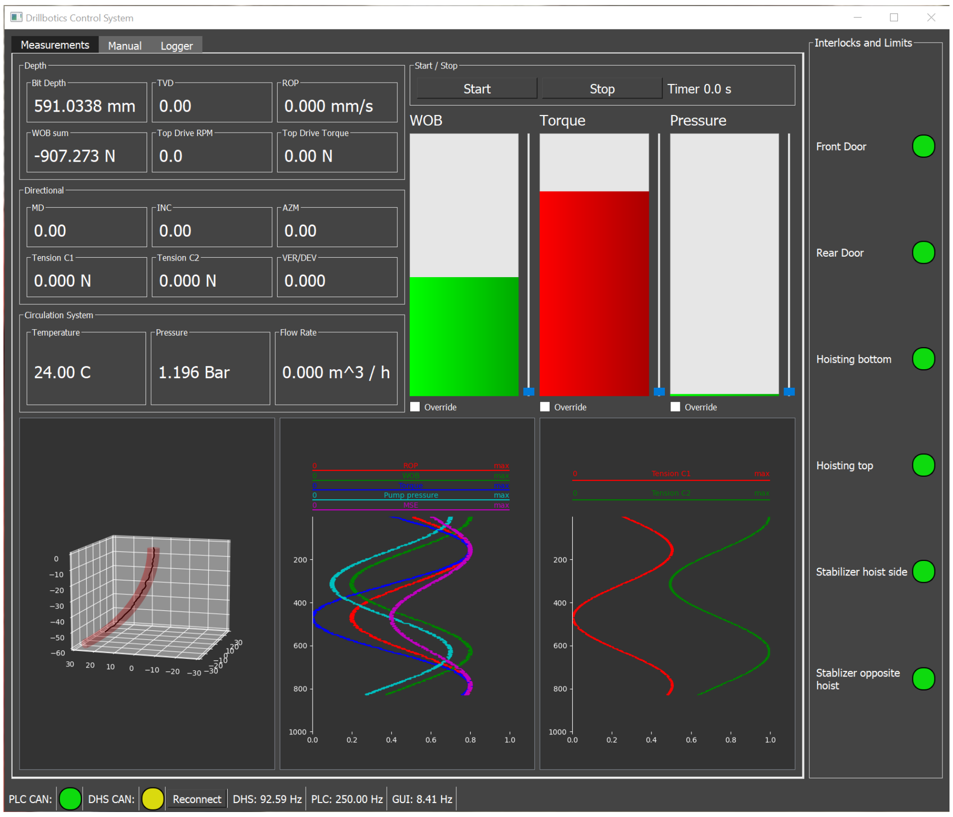
Interlocks and Limits

Front Door ●
Rear Door ●
Hoisting bottom ●
Hoisting top ●
Stabilizer hoist side ●
Stablizer opposite hoist ●

PLC CAN: ● DHS CAN: ● Reconnect DHS: 96.15 Hz PLC: 250.00 Hz GUI: 4.90 Hz

Figure 7.7: Final result

This interface accomplishes its goals, considering what was learned in chapter 7.1, and what has been discussed together with the rest of the team. Here the operator has been in the centre of the development of the GUI, and the GUI has been made so an operator can easily spot any errors or if the system is operation outside set parameters. We also allow the operator to decide whether they want dark or light mode, although this currently has to be changed by making a change in a config. The result looks can be seen in figure 7.8



Depth
Bit Depth: 591.0338 mm
TVD: 0.00
ROP: 0.000 mm/s
WOB sum: -907.273 N
Top Drive RPM: 0.0
Top Drive Torque: 0.00 N

Directional
MD: 0.00
INC: 0.00
AZM: 0.00
Tension C1: 0.000 N
Tension C2: 0.000 N
VER/DEV: 0.000

Circulation System
Temperature: 24.00 C
Pressure: 1.196 Bar
Flow Rate: 0.000 m^3 / h

Start / Stop
Timer 0.0 s

Front Door ●
Rear Door ●
Hoisting bottom ●
Hoisting top ●
Stabilizer hoist side ●
Stablizer opposite hoist ●

PLC CAN: ● | DHS CAN: ● | DHS: 92.59 Hz | PLC: 250.00 Hz | GUI: 8.41 Hz

Figure 7.8: Final result: Dark mode

Other general design decisions have also been largely decided through information learned in the review. This is points such as system status visibility, use of known conventions, consistency, error prevention and others. While a lot of what was reviewed is implemented, it is also important to remember which parts are not implemented and keep those in mind. In the case of this small-scale rig, most of the considerations not included are based on length of operation. A good example as was mentioned in chapter 7.1.1, air traffic controllers need tons of breaks during their shifts as keeping focus is not really feasible. In this case however, each time the rig is run for operation it will not be running for more than 20–30 minutes, meaning some of what was learnt in chapter 7.1 is not necessary for this project, due to other time-constraints.

7.2.3 PyQt5

PyQt5 is the GUI framework that is used for the application. Its ease of use in creating the kind of interface that was needed was one of the primary reasons for choosing this Python library in the project.

QT Designer

QT Designer is a graphical tool to set up QT applications. It is used to setup the layout and most of the display elements in the user interface. It allows us to set up grids and list so that the interface scales automatically to the size of the window. When the design process is completed, the result gets transpiled into a Python class which our application will inherit from.

Custom Elements

When the integrated elements in QT are not sufficient for the application we designed our own. These elements are created by inheriting from QT elements that are similar enough to the end result that they will be easy to work with. They are then extended with the required functionality or design and added to the GUI.

CSS

When styling custom QT elements we use QT style sheets. These style sheets are very similar to CSS [3]. This functionality is used for designing the indicator LEDs.

7.1 shows how the colour of the indicator is updated if it is different from the previous colour. The `__generate_css` method will update the style sheet of the element according to the objects properties.

```
@property
def color(self):
    return self._color

@color.setter
def color(self, value):
    if self._color != value:
        self._color = value
        self.__generate_css()

def __generate_css(self):
    self.setStyleSheet(
        f"max-width: {self.size}px;\n"
        f"max-height: {self.size}px;\n"
        f"min-width: {self.size}px;\n"
        f"min-height: {self.size}px;\n"
        f"border-radius: {self.border_radius}px;\n"
        f"border-style: solid;\n"
        f"border-color: black;\n"
        f"border-width: 2px;\n"
        f"background-color: {self._color.value}\n"
    )
```

Code Block 7.1: Indicator LED

7.2.4 Well Log Plots

The well log plots in the application is made using `pyplot` in the `matplotlib` library for Python. When the values that will be plotted are transmitted to the GUI, they are added to a field in an object of the `PlotPath2d` class shown in 7.2

The class is used to store the x and y axis of the plot. When new values are added to the NumPy arrays, they are automatically expanded if needed. If expanded, the arrays double in size.

When a new point is added, the lines in the plot are updated using the `__update_plot_lines` method in the plotter's class. This is displayed in code block 7.3. This method updates the individual plot lines, instead of calling the `plot` function each time. This makes for a dramatic perfor-

```python
class PlotPath2d:
    def __init__(self, buffsize=2):
        self.x = numpy.empty(buffsize)
        self.x[:] = numpy.nan
        self.y = numpy.empty(buffsize)
        self.y[:] = numpy.nan
        self.index = 0

    def add_point(self,x,y):
        self.x[self.index] = x
        self.y[self.index] = y

        self.index += 1

        if self.index == len(self.x):
            self.__expand_arrays()

    def __expand_arrays(self):
        old_size = len(self.x)
        new_size = old_size * 2
        self.x = numpy.resize(self.x, new_size)
        self.y = numpy.resize(self.y, new_size)
        self.x[old_size:] = numpy.nan
        self.y[old_size:] = numpy.nan
        logging.info(f"Expanded array to {new_size}
            elements")
```

Code Block 7.2: PlotPath2d class

mance improvement, especially when the plot contains many points.

```python
def __update_plot_lines(self):
    for i, line in enumerate(self.line_list):
        line.set_xdata(self.plot_paths[i].x)
        line.set_ydata(self.plot_paths[i].y)
    self.figure.canvas.update()
    self.figure.canvas.flush_events()
```

Code Block 7.3: Update plot lines method

3D Plot

Execution

The 3D plot of the drill path is plotted using using built-in functions from the pyplot package in the matplotlib library for Python. The 3D plotter object is instantiated inside the main window class, and the class Plotter3D inherits GraphicsScene, making it behave like a Qt widget. The plotted points are stored inside a PlotPath3d object, which is identical to PlotPath2d described in 7.2, only being extended to work with three dimensions. The code for the 3D plotter class is shown in code block 7.4.

```python
class Plotter3D(QGraphicsScene):
    def __init__(self):
        super().__init__()
        self.figure = plt.figure()
        self.ax = self.figure.add_subplot(111,
            projection="3d")
        self.ax.axes.set_xlim3d(-30, 30)
        self.ax.axes.set_ylim3d(-30, 30)
        self.ax.axes.set_zlim3d(-60, 0)
        canvas = FigureCanvasQTAgg(self.figure)
        self.proxy_widget = self.addWidget(canvas)
        self.drill_path = PlotPath3d()
        self.line, = self.ax.plot(self.drill_path.x,
            self.drill_path.y, self.drill_path.z)

    def __update_plot_line(self):
        self.line.set_xdata(self.drill_path.x)
        self.line.set_ydata(self.drill_path.y)
        self.line.set_3d_properties(self.drill_path.z)
        self.figure.canvas.update()
        self.figure.canvas.flush_events()

    def update_plot(self, x, y, z):
        self.drill_path.add_point(x, y, z)
        self.__update_plot_line()
```

Code Block 7.4: Code for the 3D plotter

Testing

Since it is established that we will not be able to test the 3D plot using real bit position data, we had to make up our own data for testing the 3D visualisation. Since we have a start point and a point inside the block (drill targets) that we need to drill towards, the simplest would be a straight line path connecting the three points in question. However, this is not a suitable realistic drill path for our drill bit. Therefore, we decided on creating a path based on a minimum curvature path. This would not only be ideal for our testing purposes, but is also a requirement for the competition, to have an ideal path before we start drilling. This will be more thoroughly explained in chapter 8 about drill path modelling.

Basis for model

We consider an idealistic version of the concrete block which is to be drilled as a perfect cube with each edge of the cube measuring 60 cm. The start point for drilling is in the centre of the top face of the cube, and will be chosen as the origin $(0, 0, 0)$ in accordance with Competition Guidelines. Further, a target point is given. These are chosen randomly with deltas given by constraints found in the Drillbotics Competition Guidelines.

The Guidelines specify that the maximum accumulated inclination angle with the vertical should not exceed 30°, and that the displacement from the block's centre vertical should not exceed 10 inches, or 25.4 centimetres. Given these specifications, the domain of adequate target values can be worked out as follows: To get x and y values, a set of polar coordinates (r, θ) is chosen where $r \in (0, 25.4)$ and $\theta \in (0, 2\pi)$, and convert to Cartesian values. We can work out the minimum displacement in the z direction by using the maximum values of r and θ,

$$|(\Delta z)_{min}| = r_{max} \cdot \cot \theta_{max} = 25.4 \cdot \cot 30^\circ \approx 44.0.$$

We then choose a value $z \in (-60, -44)$. The sign is negative because in the plot, 0 on the coordinate system's z axis corresponds to the the top face of the cube (i.e. the origin). x and y are found using the formulae $x = r \cos \theta$ and $y = r \sin \theta$. Further descriptions of the model which is plotted will be given in the subsequent chapter about drill path modelling.

7.2.5 Runtime Efficiency and Optimisation

One of the problems that occurred during the development phase was that the user interface crashed when the plots were updated too often. This was caused by the way PyQt5 handles the events for updating the user interface. A new event would be handled before the previous one had returned, and this caused the plots to crash. The solution to this problem ended up being to ignore the incoming data if the previous update of the GUI was yet to be completed. This approach alone is quite inefficient since all of the data that gets sent to the GUI process, but not displayed is wasted. The processing that is needed to transfer the data is therefore wasted.

A solution to this problem is to use some form of flow control and the implementation in this program draws inspiration from how congestion control is handled in TCP. The speed of the data transmission is increased for every iteration until some of the traffic has to be dropped. The GUI process will inform the acquisition system of this over a management queue. The acquisition system will then drop the transmission speed by a percentage of the current speed plus a fixed amount. The fixed amount is needed to ensure the desired speed decrease even at very high speeds. If the transmission speed is plotted over time, it would resemble a sawtooth wave, an example of which is shown in figure 7.9.

Code block 7.5 shows how the flow control is implemented on the data acquisition side of the program. The instance variable `_interval` refers to how many sets of data is collected from the PLC before one of them is sent to the GUI, while the `run` method decides whether new values should be transmitted to the GUI. If a code 1 is received from `GUI_management_queue` as shown in code block 7.6, this means that the GUI has dropped one of the data objects that were transmitted, and `_interval` is increased by calling the `slow_down` method. The speed will therefore decrease. For every time a data object is put onto the `display_queue`, the `_interval` is decreased and transmission speed is therefore increased. This is handled in the aforementioned `run` method.

```python
class FlowController:
   def __init__(self, max_value=300, min_value=25):
      self._max = max_value
      self._min = min_value
      self._n = 0
      self._interval = math.ceil((max_value + min_value) / 2)

   def run(self) -> bool:
      if self._n >= self._interval - 1:
         logging.debug(f"GUI interval: {self._interval}")
         self._interval = int(self._interval * 0.95)
         if self._interval > self._max:
            self._interval = self._max
         elif self._interval < self._min:
            self._interval = self._min
         self._n = 0
         return True

      self._n += 1
      return False

   def slow_down(self):
      prev_interval = self._interval
      self._interval = int(self._interval * 1.05) + 10
      logging.info(f"GUI interval changed from {prev_interval} to {self._interval}")
```

Code Block 7.5: Flow control

```python
  # Read from management queue for closing program and for
    controlling update interval
if self.GUI_management_queue.qsize() > 0:
   msg = self.GUI_management_queue.get()
   if msg == -1:
      break
   if msg == 1:
      self.GUI_flow_controller.slow_down()

if self.GUI_flow_controller.run():
   # Update the GUI when the flow controller allows it
   self.display_queue.put(self.drill_controller.gui_output())
```

Code Block 7.6: Use of flow control in the main loop

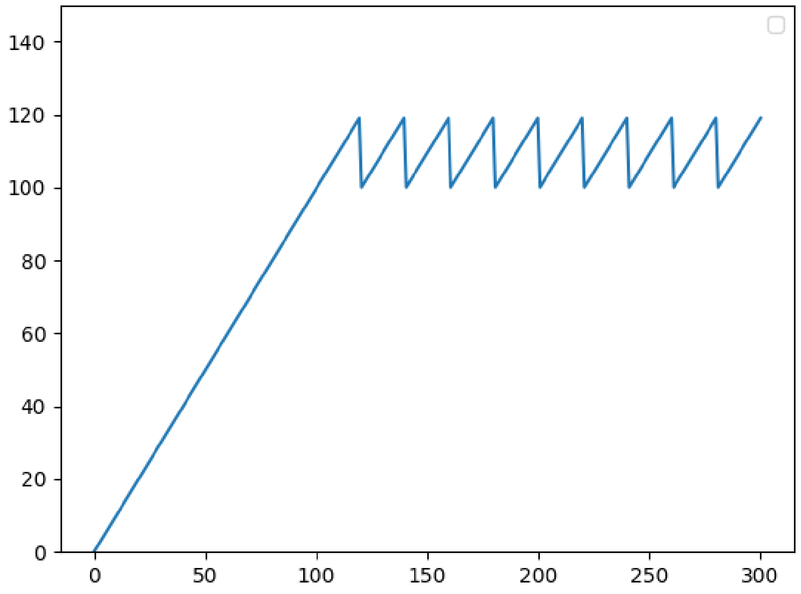


Figure 7.9: Example of sawtooth wave

The automatic speed adjustments mean that the software is able to be run on a wide range of hardware. It will slow down when run on a slower machine or using a display with higher screen resolution, and will speed up when run on fast hardware and a low screen resolution.

Chapter 8

Drill Path Modelling

Being able to suggest a path for the drilling is a required part of the competition. The standard way, and way that is recommended by the competition is the *minimum curvature method.* It is stated in the competition guidelines that "minimum curvature calculated trajectory (to represent the drilled wellbore position)" is a minimum requirement for the competition. To best do this we figured representing the trajectory in 3D and overlying the actual position and path on top of the minimum curved trajectory (best path) would be a good way to do this. The minimum curvature method is widely accepted as the industry standard for 3D directional surveys/estimations. [14]

The following is directly from the Drillbotics Guidelines:

All teams are required to provide plan vs. actual plots containing the following minimum requirements:

- *As-drilled trajectory and original planned trajectory shown on same TVD vs. Vertical Section plot*
  - *Vertical section direction to be determined by well center-to-target bearing*
- *As-drilled trajectory and original planned trajectory shown on same X/Y plot*
  - *Grid north reference to block north*

   – *[0,0] at well center*

Interpreting the above requirements, it can be concluded that all these requirements can be satisfied by creating a 3-dimensional plot.

8.1 Theory

In the preliminary phase of the GUI design process, a path based on a sine curve was sufficient for testing purposes. However, as described in the previous subchapter, this is not representative of the actual path which will be used for drilling. For drilling purposes, the basis for the path model needs to be calculated according to the minimum curvature method.

8.1.1 Minimum Curvature Path

A sketch for a typical minimum curvature path is given in figure 8.1. The starting point is the origin $A(0, 0, 0)$, and the target point is $B(x, y, z)$. The given constraint is that B is no further than 10 inches from the centre z axis, i.e. $\Delta xy = \sqrt{x^2 + y^2} < 25.4$. Further information on the selection of the target point $B(x, y, z)$ is given in chapter 7.2.4 about the basis for the 3D plot model.

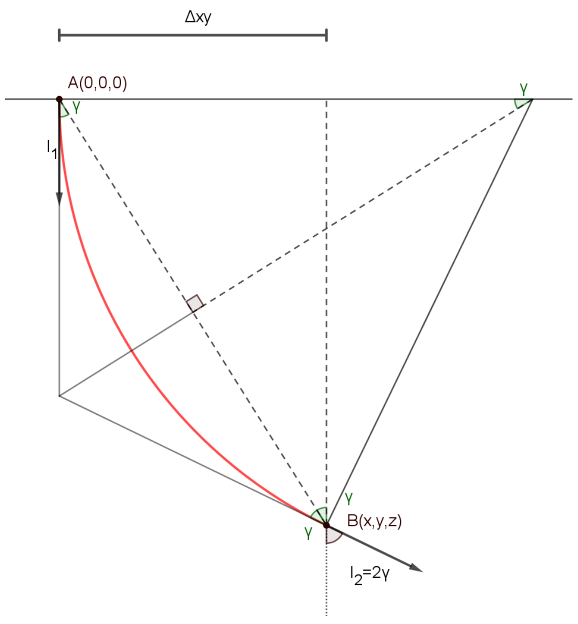


Figure 8.1: An illustration of the minimum curve path

To calculate the minimum curvature path, the radius of the circle on which the arc lies must first be computed. Using the top right-angled triangle in figure 8.1, the circle radius CR has a relationship given by

$$\sin\gamma = \frac{\overline{AB}/2}{CR} \Leftrightarrow CR = \frac{\overline{AB}}{2\sin\gamma}$$

Using the other right-angled triangle, whose smallest cathetus is Δxy and with hypotenuse $\overline{AB}$, the angle γ is worked out:

$$\tan\gamma = \frac{\Delta xy}{|z|} \Leftrightarrow \gamma = \tan^{-1}\frac{\Delta xy}{|z|}$$

Substituting this into the formula for CR, as well as using $\overline{AB} = \sqrt{x^2+y^2+z^2}$ and $\Delta xy = \sqrt{x^2+y^2}$, the following is found:

$$CR = \frac{\overline{AB}}{2\sin\left(\tan^{-1}\frac{\Delta xy}{|z|}\right)} = \frac{\sqrt{x^2+y^2+z^2}}{2\frac{\frac{\sqrt{x^2+y^2}}{|z|}}{\sqrt{1+\left(\frac{\sqrt{x^2+y^2}}{|z|}\right)^2}}} = \frac{1}{2\sqrt{x^2+y^2}}\left(x^2+y^2+z^2\right)$$

When the value CR is given, the circle's midpoint can worked in relation to the path's starting point by first finding the angle α between A and B in the xy-plane, and then multiplying by the radius:

$$\begin{aligned}\alpha &= \operatorname{atan2}(y, x)\\ \text{circle}_x &= A_x + CR\cos\alpha\\ \text{circle}_y &= A_y + CR\sin\alpha\end{aligned}$$

In order to make a paramterisation of the minimum curvature path, the angle ϕ is varied along the path. This will be varied in the range $[0, \phi_{max}]$. The value ϕ_{max} is calculated using trigonometry along with the displacement in the xy-plane and z direction between the circle midpoint and the path's endpoint (B):

$$\phi_{max} = \tan^{-1}\frac{|z|}{\sqrt{(\text{circle}_x - B_x)^2 + (\text{circle}_y - B_y)^2}}$$

A parameterisation for $t \in [0, 1]$ can now be given by

$$\begin{aligned}x(t) &= \text{circle}_x - CR\cos\alpha\cos(\phi_{max}t)\\ y(t) &= \text{circle}_y - CR\sin\alpha\cos(\phi_{max}t)\\ z(t) &= A_z - CR\sin(\phi_{max}t)\end{aligned}$$

A more general version of the minimum curvature path used in industry is given in [6]. However, given our case where $I_1 = 0$, $I_2 = 2\gamma$ and $A_1 = A_2 = \operatorname{atan2}(y, x)$, confer figure 8.1, the formulae in [6] turn out to be identical to the ones that have been derived out from scratch.

8.1.2 Random Walk

After having discussed discussed the 3D plot and path model with our supervisor, it was decided to implement a random walk algorithm in order to simulate real data coming from the rig. [16]

The idea behind a random walk algorithm is to introduce some uncertainty in the ideal path that otherwise would be plotted. This would be similar to a phenomenon known as tortuosity in real-life drilling, which is defined as “the obvious deviation of the wellbore trajectory in drilling direction”. In figure 8.2, an example is given as to how this deviation might look like, looking like a so-called dogleg. The random walk algorithm aims to recreate this kind om randomness.

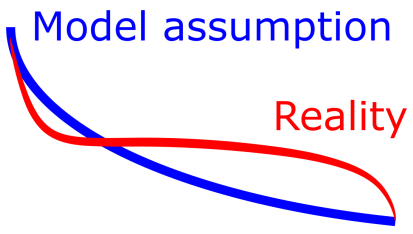


Figure 8.2: An example illustration of the randomness in the drill path caused by tortuosity in the medium.

The random walk model aims to introduce deviations in the x and y values of the model. This is done by finding some points evenly spaced in the z direction which the path will go through. The points are chosen a random angle from the path, while the radii are chosen from a range of predefined allowed values. The reasoning behind this, is so that the points should not lie too close to the path, while also not too far away. After the points have been chosen randomly, interpolation is used to string them together. The points only give deltas which are to be added to the path's positional arrays.

8.1.3 Safety Margin Surface

The safety margin surface is shaped like a curved cylinder around the ideal minimum curvature drill path. This is so that it can be easily identified when the drill path ends up outside of this predefined margin. The main idea for this visualisation was to plot an opaque cylinder around the ideal minimum curvature path. Given a Minimum Curvature Path parameterisation, the points (x, y, z) points needed to plot the cylinder can be worked out as described in this subchapter.

The general idea of the implementation is based vectors and vector rotations. A vector perpendicular to the path is found and rotated all the way around the path. Given lists P_x, P_y, P_z of points (x, y, z), they are shifted by one in order to find a list of tangential vectors T_x, T_y, T_z:

$$\begin{aligned}\mathrm{tang}_{y,n} &= P_{y,n+1} - P_{y,n}\\ \mathrm{tang}_{y,n} &= P_{y,n+1} - P_{y,n}\\ \mathrm{tang}_{z,n} &= P_{z,n+1} - P_{z,n}\end{aligned}$$

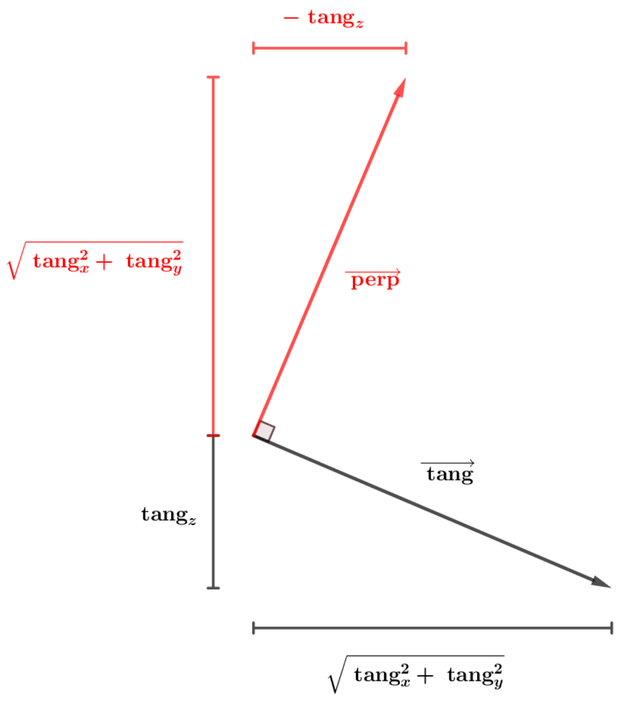

$-\mathbf{tang}_z$
$\sqrt{\mathbf{tang}_x^2 + \mathbf{tang}_y^2}$
$\overrightarrow{\mathbf{perp}}$
$\overrightarrow{\mathbf{tang}}$
$\mathbf{tang}_z$
$\sqrt{\mathbf{tang}_x^2 + \mathbf{tang}_y^2}$

Figure 8.3: An illustration of the perpendicular and tangent vector calculation.

When the resolution of the lists P is sufficiently high, it can be argued that T is a good enough approximation for the derivative of the path function, i.e. the true tangential vector. From figure 8.3, it can be seen that the perpendicular vector will be found by multiplying the x and y values of the tangential vector by the factor $-\mathrm{tang}_z/\mathrm{tang}_{xy}$, while the z value is simply tang_{xy}. Further, the list of vectors perpendicular to the path is found by using the following formulae:

$$\text{perp}_x = -\frac{\text{tang}_x\text{tang}_z}{\sqrt{\text{tang}_x^2 + \text{tang}_y^2}}$$
$$\text{perp}_y = -\frac{\text{tang}_y\text{tang}_z}{\sqrt{\text{tang}_x^2 + \text{tang}_y^2}}$$
$$\text{perp}_z = \sqrt{\text{tang}_x^2 + \text{tang}_y^2}$$

From equation 20 in *Three-Dimensional Rotation Matrices* [8], a general rotation matrix for rotation an angle θ about a normal vector $\hat{n}$ is given by

$$R(\hat{n}, \theta) =$$
$$\begin{pmatrix} \cos\theta + n_1^2(1-\cos\theta) & n_1n_2(1-\cos\theta) - n_3\sin\theta & n_1n_3(1-\cos\theta) + n_2\sin\theta \\ n_1n_2(1-\cos\theta) + n_3\sin\theta & \cos\theta + n_2^2(1-\cos\theta) & n_2n_3(1-\cos\theta) - n_1\sin\theta \\ n_1n_3(1-\cos\theta) + n_3\sin\theta & n_2n_3(1-\cos\theta) + n_1\sin\theta & \cos\theta + n_3^2(1-\cos\theta) \end{pmatrix}$$

This is used in order to rotate $\overrightarrow{\text{perp}}$ in order to get a vector $\vec{v}$ rotated by an angle θ:

$$\vec{v} = R(\overrightarrow{\text{tang}}, \theta) \cdot \overrightarrow{\text{perp}}$$

These vectors, rotating theta a set number of points to cover the region $\theta \in (0, 2\pi)$, along with the position vectors of the minimum curvature gives the points that form the basis of the safety margin surface in the 3D plot. An interpolation is used to cut down the resolution of the safety marigin surface, as the surface doesn't need to be as detailed as the minimum curvature path itself. We then use a method in matplotlib to plot the surface as a mesh of the (x, y, z) points, and this will be described in greater detail in subchapter 8.2.3.

8.2 Python Implementation

This subchapter will describe how the mathematics found in subchapter 8.1 has been implemented into Python code. This will be done using code snippets and screenshots from the GUI.

8.2.1 Minimum Curvature Path

The minimum curvature path is implemented as its own class object, with the result being the parametersation made up by the instance variables `self.par_x`, `self.par_y` and `self.par_z`. These are NumPy arrays of floats. The code for the minimum curvature path is given in code block 8.1.

```python
class MinimumCurvaturePath:
    def __init__(self, startpoint: tuple[float, ...],
        endpoint: tuple[float, ...], resolution: int =
        1000) -> None:
        self.startpoint = startpoint
        self.endpoint = endpoint
        self.resolution = resolution
        self.dx = self.endpoint[0] - self.startpoint[0]
        self.dy = self.endpoint[1] - self.startpoint[1]
        self.dz = self.endpoint[2] - self.startpoint[2]

        circle_radius = 0.5 * (self.dx**2 + self.dy**2 +
            self.dz**2) / numpy.sqrt(self.dx**2 + self.dy**2)
        azimuth_angle = numpy.arctan2(self.dy, self.dx)
        circle_x = circle_radius * numpy.cos(azimuth_angle)
        circle_y = circle_radius * numpy.sin(azimuth_angle)
        center = (circle_x + self.startpoint[0], circle_y +
            self.startpoint[1], self.startpoint[2])
        t_values = numpy.linspace(0, 1, self.resolution)
        phi_max = numpy.arctan(numpy.abs(self.dz)/
            numpy.sqrt((center[0]-self.endpoint[0])**2 +
            (center[1]-self.endpoint[1])**2))
        phi = phi_max * t_values

        self.par_x = center[0] - circle_radius *
            numpy.cos(azimuth_angle) * numpy.cos(phi)
        self.par_y = center[1] - circle_radius *
            numpy.sin(azimuth_angle) * numpy.cos(phi)
        self.par_z = center[2] - circle_radius *
            numpy.sin(phi)
```

Code Block 8.1: Python code for minimum curvature path

The drill controller will give the GUI front end one data set at a time, with each data set corresponding to one (x, y, z). As such, the class needs to be adapted to be able to give one point at a time, and also keep track of where it is along the path. This is done with an extra instance variable `index` and a method `get_next_point`, as shown in code block 8.2.

```python
class MinimumCurvaturePath:
    def __init__(self, (...)) -> None:
        (...)

        self.index = 0

    def get_next_point(self) -> tuple[float, ...]:
        point = (self.par_x[self.index],
            self.par_y[self.index], self.par_z[self.index])
        self.index += 1
        self.index %= self.resolution
        return point
```

Code Block 8.2: Python code for the get next point functionality.

8.2.2 Random Walk

The random walk is implemented in Python as described in subchapter 8.1.2. As with the minimum curvature path, it is added as its own class object, see code block 8.3. The `__init__` method of the `RandomWalk` class takes in a resolution, which is the length of the array of points that `generate_arrays` method returns. This length should correspond with the length of the path that the random walk is applied to. It also takes in the number of points that the random walk path should go through. When generating points, the number is subtracted by two, because the random path should always begin and end at $(0, 0)$ deviation in the x and y axes.

A range of radii is also specified, forming a band from which the random point can be chosen. Using random angles, the points are chosen and converted from polar to Cartesian coordinates. Lastly, the random points are stringed together to match the resolution argument.

In order to add random walk capability to the minimum curvature path, the `__init__` method will be extended slightly, see code block 8.3. The

```python
class RandomWalk:
   def __init__(self, resolution: int, radius_range:
      tuple[float, float], num_of_points: int) -> None:
      (...)

   def generate_arrays(self) -> tuple[numpy.ndarray,
      numpy.ndarray]:
      radii = numpy.random.uniform(*self.radius_range,
         self.num_of_points - 2)
      thetas = numpy.random.uniform(0, 2*numpy.pi,
         self.num_of_points - 2)
      x_points = numpy.r_[0, radii * numpy.cos(thetas), 0]
      y_points = numpy.r_[0, radii * numpy.sin(thetas), 0]
      x_interpolated = numpy.interp(numpy.linspace(1,
         self.num_of_points, self.resolution),
         numpy.linspace(1, self.num_of_points,
         self.num_of_points), x_points)
      y_interpolated = numpy.interp(numpy.linspace(1,
         self.num_of_points, self.resolution),
         numpy.linspace(1, self.num_of_points,
         self.num_of_points), y_points)
      return x_interpolated, y_interpolated

class MinimumCurvaturePath:
   def __init__(self, (...), random_walk_radius_range:
      tuple[float, float] = (0.3, 0.6),
      random_walk_points: int = 50) -> None:
      (...)

      random_walk_x, random_walk_y =
         RandomWalk(self.resolution,
         random_walk_radius_range,
         random_walk_points).arrays
      self.par_x += random_walk_x
      self.par_y += random_walk_y
```

Code Block 8.3: Python code for random walk

`__init__` method of `MinimumCurvaturePath` is extended to also take in arguments `radius_range` and `num_of_points` which it uses for instantiating the `RandomWalk` object. The default parameters of $r \in (0.3, 0.6)$ and $n = 50$ points work well with the scale that the application uses, but can be changed as required. For instance, using a random walk radius range of $(0, 0)$ will give a perfect minimum curvature path without any randomness.

8.2.3 Safety Margin Surface

The safety margin surface used in the 3D plot is generated as specified in subchapter 8.1.3. The Python code for this is given in the following code snippets. The first code block 8.4 shows the `__init__` method and the arguments that it takes. it is based on a `MinimumCurvaturePath`, which should use `random_walk_radius_range=(0,0)`. It also takes an argument `margin`, which specifies how far from the path the surface should lie. The resolution can also be adjusted by passing a corresponding argument. `THETA_RESOLUTION` is an internal constant value for the safety margin surface class, and specifies how many angles around the path should be used. This corresponds to the angle θ in the rotation matrix $R(\hat{n}, \theta)$ described in subchapter 8.1.3.

```python
class SafetyMarginSurface:
    def __init__(self, mcp: MinimumCurvaturePath, margin=1,
        resolution=50) -> None:
        THETA_RESOLUTION = 24
        # shift all values by one and calculate tangent
            vectors
        dx = mcp.par_x[1:] - mcp.par_x[:-1]
        dy = mcp.par_y[1:] - mcp.par_y[:-1]
        dz = mcp.par_z[1:] - mcp.par_z[:-1]
        dxy = numpy.sqrt(dx ** 2 + dy ** 2)

        #calculate perpendicular vectors
        perp_x = -dx * dz / dxy
        perp_y = -dy * dz / dxy
        perp_z = dxy
```

Code Block 8.4: Part 1 of the python code for the safety margin surface

With the tangential vectors calculated in code block 8.4, the perpendicular vectors can be calculated as shown in code block. The perpendicular vectors and tangent vectors are normalised so that they can be used in later calculations. We also initialise separate x, y and z matrices which will be used to plot the surface.

Next, in code block 8.5, the calculation of all the points is done. This is done by creating an appropriate rotation matrix for the point where the path along the curve for every angle `theta`, and calculating the dot product with a vector. After this is done, the points are added to the respective matrix that was initialised earlier, see code block 8.6. The three matrices are stored in the instance variable tuple `self.to_plot` which will be given to the matplotlib's surface plot function.

```python
for k, theta in enumerate(numpy.linspace(-numpy.pi/2,
    3*numpy.pi/2, THETA_RESOLUTION)):
    for i in range(length):
        n1, n2, n3 = tang_x[i], tang_y[i], tang_z[i]
        p1, p2, p3 = perp_x[i], perp_y[i], perp_z[i]
        rot_matrix = numpy.array((
            ( numpy.cos(theta) + n1**2 * (1-numpy.cos(theta))
                ,
                n1*n2*(1-numpy.cos(theta))-n3*numpy.sin(theta),
                n1*n2*(1-numpy.cos(theta))+n2*numpy.sin(theta)
                ),
            ( n1*n2*(1-numpy.cos(theta))+n3*numpy.sin(theta),
                numpy.cos(theta) + n2**2 *
                (1-numpy.cos(theta)) ,
                n2*n3*(1-numpy.cos(theta))-n1*numpy.sin(theta)
                ),
            ( n1*n3*(1-numpy.cos(theta))-n2*numpy.sin(theta),
                n2*n3*(1-numpy.cos(theta))+n1*numpy.sin(theta),
                numpy.cos(theta) + n3**2 *
                (1-numpy.cos(theta)) ),
        ))
        r1, r2, r3 = rot_matrix.dot(numpy.array((p1, p2,
            p3)))

        # append to lists
        (...)

    # normalise and multiply by length ("margin" argument)
    (...)
```

Code Block 8.5: Part 2 of the python code for the safety margin surface

```
    point_x = numpy.interp(numpy.linspace(1, length,
        resolution), numpy.linspace(1, length, length),
        rot_x + px)
    point_y = numpy.interp(numpy.linspace(1, length,
        resolution), numpy.linspace(1, length, length),
        rot_y + py)
    point_z = numpy.interp(numpy.linspace(1, length,
        resolution), numpy.linspace(1, length, length),
        rot_z + pz)

    # add points to matrices x_mat, y_mat and z_mat
    (...)

self.to_plot = x_mat, y_mat, z_mat
```

Code Block 8.6: Part 3, interpolation to get wanted number of points.

Lastly, to make the safety margin surface, the `Plotter3D` class is extended with the method `plot_safety_margin_surface()`. This is shown in code block 8.7.

```
class Plotter3D(QGraphicsScene):
    (...)
    def plot_safety_margin_surface(self, sms:
        SafetyMarginSurface):
        self.ax.plot_surface(*sms.to_plot, alpha=0.5)
```

Code Block 8.7: Part 4, adding the point matrices to the 3D plotter.

Chapter 9

Conclusion & Future Work

When the University needed a computer science group to help with this years Drillbotics competition, this was seen by our group as a brilliant project to do our thesis on. With a complete architectural redesign it meant we got to design things with our own standards and got to dictate quality. While certainly interesting, it has also given us a lot of challenges that have all been overcome. All the parts described in chapter 1 have all been implemented to varying degrees. These parts include *data acquisition system*, *data management system*, *drill control system*, *graphical user interface*, a well-designed *software architecture* and a system for using industry standard methods for drilling deciding paths. Most of the yet-to-be-added features are all related to the unusually late testing of the rig, although they are eventually going to be needed for this to be a complete product.

The first major section that was worked on was to complete a good design for the *software architecture* of UiS' Drillbotics drill rig. This was never intended to just have implications for this project, but the intention was from the very beginning to create everything we did so it would be scalable and could be used in future editions of Drillbotics by UiS. Therefore a lot of time was spent into creating good abstractions and class design so it would be relatively straight forward for any future Drillbotics teams at this university.

As was mentioned above, our priorities had to be re-aligned when we learned we had no accompanying bachelor group to handle the sections for automation and electronics-design, thus the *data acquisition system* had to be started as a major priority by our group instead. Many of the other features relied on us having data from the rig through this system, and thus it had to be robust. Thanks to Drillbotics member Magnus Wersland the development for the data acquisition system went seamlessly both on computer and PLC side of things. He filled in most of the blanks and helped us where we were inexperienced due to PLC programming being something that was taught in the automation and electronics-design bachelor. The system itself is done, but there are quite a few sensors that have yet to be added due to the rig being unfinished. More about this in 9.1. This system is responsible for both sending data between rig/PLC to the computer both ways.

Another competition requirement and feature that is nice to have is the *data management system*, or *logger*. Our database is a SQLite3 database. SQLite3 was chosen as it was the ideal option for our specific use as it didn't require to be very data-intensive. The requirement for our system was to be able to insert roughly 20 data sets per second, and the running time being at maximum 30 minutes. With this configuration in mind we also stress-tested the database to ensure we had comfortable margin between required speed and top speed. As can be read in chapter 5.2, the implemented logging system has potential to work at 15x faster than is required when running for 12 hours straight, giving us comfortable margins. Our choice of database depended on comfortable margins, so having margins as wide as they are was very important.

The control system is deciding how inputs from the user are turned into actions performed by the rig. It controls what is happening in the program before the data is picked up by the data acquisition system for sending over the CAN channels, essentially just making the messages ready to be sent from the computer, via the PLC and to the equipment on the rig. The system for this is finished, and we have manual control over the equipment that has been added. However, as the rig is planned to be finished in the weeks following the deadline, there are still pieces of the rig missing and therefore there is still equipment that is yet to be added. Using good abstractions has made it easy to add control over new pieces as they are added. Automatic drilling is yet to be added, more about this in chapter 9.1.

The most time-intensive section of our project was the literature review-and subsequent implementation of the GUI. A primary reason for us spending much time with the GUI, was the organiser implementing human machine interface (HMI) as another central judging criterion that we wanted to do well. It was recommended that we perform a literature review prior to creating the GUI such that we knew which principles were important when designing our system, especially when keeping HMI in mind. We reviewed material provided by the Drillbotics hosts, formed UiS interfaces, along with many separate sources to get some baseline ideas that could be further developed on. After the review was finished and we had gotten some conclusions from it we started the development of the GUI. The design process of the GUI, from first draft to implementation can be seen in chapter 7.2.2. The final result of the GUI is satisfactory when comparing to our goals, however there are more features we would like to add to it as soon as the rig has been tested, and we are receiving real data. The reason we need real data is because we'd need to be able to set thresholds to certain sensor output such that if these thresholds were exceeded it'd give alarms. Read more about our proposed future work in chapter 9.1.1.

Last major feature we worked on was related to being able to estimate what the best possible path would be from any given position given 2 points. The major idea here is to put the assembly on the block, set coordinates to (0, 0, 0) and from there getting to any given (x, y, z). The industry standard method, and the method we use, is called minimum curvature path, which is just essentially the arc of a circle to have as little curving as possible. In addition to adding minimum curvature path, with some random walk for simulation purposes, we also give the path a certain safety margin around the entire curved path, both visually and in the program, so it can warn the operator if it steps outside the pre-defined boundaries.

When we learned early on that one of the intended groups for the project could not be assigned, i.e. the automation and electronics-design group, our priorities were swiftly altered such that a larger focus could be on critical components, as opposed to features that are not critical for the system to work. While tough, this reorganisation reflects what happens every day in the industry, both on the computer science side, but also in most fields. Attempting to have good project management has helped a ton with getting decent results where we needed them.

As has been outlined in this chapter, and that we'll look at in chapter 9.1, there are still features to be added. Looking at the competition guidelines we've also been able to complete the requirements which were relevant to this project. See chapter 9.1 for more information into what features should- and can be added in the future.

9.1 Future Work

A project like this one is the type of project where features can be added and the code improved almost indefinitely. However, it is clear what the rig could use the most out of the things that are not implemented. As was mentioned in chapter 1.1, much of the physical testing has been planned for the weeks following the thesis deadline, after this eventual physical testing there are a few things that take priority.

9.1.1 GUI

The highest priority would be to adjoin the yet-to-be-added sensor-data into the GUI as soon as they are in place physically. Several of the displayed values are currently not directly tied into the drill rig itself, although the architecture is in place to allow then to be swiftly added as soon as they're in place. For the data coming into the GUI we'd also like to add thresholds so we can define safe operating ranges and have the numbers, or background of the numbers change colours representing certain thresholds.

There are also some lower priority features for the GUI that were considered, and could be future work. The first of which is to add a more proper design to the manual steering page. It is strictly not necessary for the competition, nor for normal use (as automatic control is the de-facto standard), but would be a good addition. Another feature that was discussed was to have a *drilling finished* screen after we stopped drilling. This screen would contain information for actual vs planned drill path and highlighting how far off it was, average values for ROP, WOB, Torque and Pump Pressure, display total time, final coordinate and lastly a button to export the logs for the session.

Eventually the GUI will also need to have a field where the operator can enter in the target (x, y, z) coordinate we want to hit with the drill, and redraw the canvas to include the proposed path to it.

9.1.2 Downhole Sensor

The data from the downhole sensor needs to be converted to a format that is easy to use (could be a direction vector) for further processing and plotting.

9.1.3 Automatic Steering

Automatic steering is one of the features missing that has the highest priority to be implemented. However automatic steering can't be implemented before all of the control system is made, and all the components are added to the rig physically. As soon as this is done, to implement automatic control you'd have to make an algorithm to control each of the *states* of the rig.

9.1.4 Estimation of Bit Position

After automatic drilling has been implemented and testing can start, implementing estimation of bit position would be next on the list of things to prioritise. This estimation has to be actively calculated based on the vectors mentioned in 9.1.2.

9.1.5 Alarm Management and Fault Prevention

Our alarm management is already functional and alerts properly if certain thresholds are exceeded in the form of typical alarm colours. However we can improve upon the current system by having more describing text alarms, e.g. "pump pressure at critical value" and similar so it becomes marginally easier for the operator to know immediately what error has occurred. This could also be taken a step further where you could detect and inform if certain errors occur, such as “stuck pipe” and “water leakage” are examples of alerts that could be derived just with our data, if we have the alarms trigger if certain sets of alarms exceed the thresholds. Also implementing an alarm if the drill path escapes the safety margins given would be preferable.

In the future there could also be time put into improving on these aspects by using machine learning to help with both fault detection and trend prediction.

Bibliography

[1] Martin Anderson. *The Ironies of Automation.* 1983. URL: https://humanfactors101.com/2020/05/24/the-ironies-of-automation/ (visited on 05/11/2022).

[2] Cameron. *Digital Drilling Control System.* 2014. URL: https://www.slb.com/-/media/files/cam-drlg-re/brochure/digital-drilling-control-system-br.ashx (visited on 05/11/2022).

[3] The QT Company. *Qt Style Sheets.* URL: https://doc.qt.io/qt-5/stylesheet.html (visited on 03/31/2022).

[4] Drillbotics. *Drillbotics: Autonomous Drilling with a Miniature Drilling Rig.* 2022. URL: https://drillbotics.com/ (visited on 02/02/2022).

[5] Drillbotics. *Drillboticső Guidelines Group B.* 2022. URL: https://drillbotics.com/wp-content/uploads/simple-file-list/Guidelines/Guidelines-2022/2022-Drillbotics-Guidelines-Rev-2-Group-B.pdf (visited on 03/07/2022).

[6] DrillingFormulas.com. *Minimum Curvature Method.* 2010. URL: https://www.drillingformulas.com/minimum-curvature-method/ (visited on 05/01/2022).

[7] Python Software Foundation. *Private Variables.* 2021. URL: https://docs.python.org/3/tutorial/classes.html#private-variables (visited on 03/31/2022).

[8] Howard Haber. *Three-Dimensional Rotation Matrices.* 2012. URL: http://scipp.ucsc.edu/~haber/ph216/rotation_12.pdf (visited on 05/01/2022).

[9] Health and Safety Executive. *Human factors: Alarm management.* URL: https://www.hse.gov.uk/humanfactors/topics/alarm-management.htm (visited on 03/29/2022).

[10] IEA/ILO. *Principles and Guidelines for Human Factors/Ergonomics (HF/E) Design and Management of Work Systems.* May 1, 2020. URL: https://www.sintef.no/globalassets/project/hfc/documents/principles-and-guidelines_may2020-1-1_ilo_iea.pdf (visited on 05/14/2022).

[11] John D. Lee et al. *Designing for People: An introduction to human factors engineering.* 2017. URL: https://www.researchgate.net/publication/319402797_Designing_for_People_An_introduction_to_human_factors_engineering (visited on 03/22/2022).

[12] Marcin Nazaruk, Peter Gibson, and Fred Florence. *Drillbotics Competition Adds Human Factors Requirements.* Nov. 9, 2021. URL: https://jpt.spe.org/drillbotics-competition-adds-human-factors-requirements (visited on 03/07/2022).

[13] Jakob Nielsen. *10 Usability Heuristics for User Interface Design.* 1994. URL: https://www.nngroup.com/articles/ten-usability-heuristics/ (visited on 04/04/2022).

[14] S. J. Sawaryn and J. L. Thorogood. *A Compendium of Directional Calculations Based on the Minimum Curvature Method.* Mar. 15, 2005. URL: https://onepetro.org/DC/article/20/01/24/112554/A-Compendium-of-Directional-Calculations-Based-on (visited on 04/27/2022).

[15] Sintef. *Automation and autonomous systems: Human-centred design in drilling and well.* Dec. 15, 2020. URL: https://www.ptil.no/contentassets/72cdb2badf19412da21fc318696b14a6/2020_automatisering_og_autonome_systemermenneskesentrert_design-eng-2.pdf (visited on 05/14/2022).

[16] Wikipedia contributors. *Random walk — Wikipedia, The Free Encyclopedia.* Mar. 1, 2022. URL: https://en.wikipedia.org/w/index.php?title=Random_walk&oldid=1074697533 (visited on 05/14/2022).

[17] Joost de Winter and Peter A. Hancock. *Reflections on the 1951 Fitts List: Do Humans Believe Now that Machines Surpass them?* 2015. URL: `https://www.researchgate.net/publication/281587449_Reflections_on_the_1951_Fitts_List_Do_Humans_Believe_Now_that_Machines_Surpass_them` (visited on 03/10/2022).

Appendix A

Source Code

The source code of our project is published as a GitHub release, and can be found by following this URL: `https://github.com/Drillbotics-UiS/control-system-2022/tree/v1.0`