1 Software

1.1 Overview

To explain the design process of the scanner software, it is first necessary to consider the purpose of this software. The scanner software needs to serve the following primary functions:

- To unify the multiple hardware elements of the scanner into a single point of control for the user.
- To perform the necessary signal processing on the digitised output from the sensor circuit to derive the secondary magnetic field in the material.
- To utilise image processing techniques on those values of secondary magnetic field to find defects in the material and present those results in an interpretable way to the user.

The software provides a layer of abstraction that simplifies operation of the scanner as much as possible, while retaining the necessary functionality so that the user also has as much control as they desire over the scanning process. It should be noted that the signal and image processing components are significant enough to warrant their own Sections, **??** and **??**. Figure 1 shows the position of the software within the system as a whole. From this figure it should be evident that the successful implementation of this software, objective 5, was essential to achieving the project aim. The graphical user interface is the singular point of control where the user interfaces with the scanner, and the operation of the scanner is completely controlled through the scanner class. If the implementation of the software fails, it is not possible to operate the scanner as a unified system and obtain results.

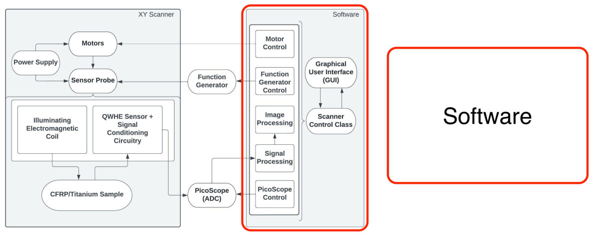

Figure 1: Position of software within system diagram

The remainder of this section will describe the design process behind the implementation of this software. First will be the choice of programming language, followed by how each of the individual pieces of software functionality was implemented, then finally how this was all then combined to function as a single coherent scanning operation.

1.2 Choice of Programming language

The language chosen to implement control of the scanner was Python. Part of this reason for this is that it is extremely popular, which means that there were an enormous amount of resources online to assist with the completion of this project. In part due to it's popularity, it was also the programming language with which most of the team members were already familiar. There were also more technical reasons which made it a suitable choice. It is relatively easy to use as it is a high-level programming language. It uses a simple syntax that tries to imitate written language as much as possible, it is dynamically typed so the programmer does not have to worry about managing data types of variables, and it utilises a garbage collector so the programmer also doesn't have to worry about memory management [1].

While this makes Python code easier to write, it can result in inefficient code that may not have met the performance requirements for the system. It was also possible that there may have been lower-layer functionality needed to control hardware that was not possible to implement in Python. In anticipation of these potential issues, before committing to using Python for this project it was confirmed that C can be easily imported into Python code [2].

1.3 Basic Functionality

With a complex system such as this scanner software it is very helpful to break this larger problem down into smaller sub-tasks to be solved. This helps to identify exactly what every aspect of the software will need to do, and to identify the interdependencies between the different software elements. Another benefit of approaching this software in a modular way is that it makes it more conducive for group work, a team member can be assigned a functional block, knowing what it needs to do, without needing to understand the system as a whole. For this reason the first place to start with the software design was by creating a functional diagram of the system, as shown in Figure 3 below.

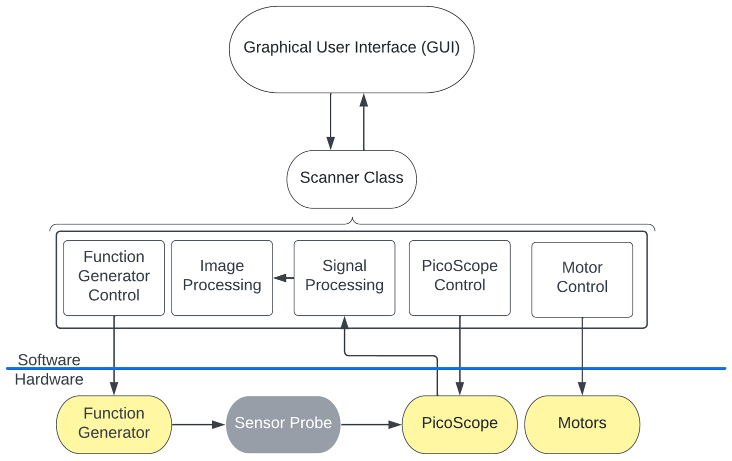

Figure 2: Functional diagram of scanner software and associated hardware devices

Working up from the bottom, the items highlighted in yellow are the 3 main interfaces between the software and the scanner. These are:

- The dual output function generator that provides the input signals to the sensing circuit and the illuminating coil.
- The PicoScope ADC which converts the analogue output of the sensor circuit into digital values.
- The motors which control the position of the probe head of the scanner.

The arrows from the software indicate the relevant control blocks for these hardware blocks (Note the blue line that delineates the barrier between hardware and software). The process of implementing the functionality required for each of these blocks will now be described.

1.4 Functional Blocks

1.4.1 Function Generator Control

The dual output function generator provides the input signals to the sensing circuit and the illuminating coil, setting the waveform, frequency and amplitude of both signals. The frequency of the magnetic field is one of the key scanning parameters as discussed in Section **??**, and the user will therefore need control over this. Alongside this, the magnetic field the coil produces is proportional to the voltage amplitude of the input signal, for reasons discussed in Section **??**, so the user needs control of both these parameters. The input signal to the sensor circuit has less of an impact on the results, however the user still requires control of the frequency, mainly to avoid potential undesirable resonance with the coil. Alongside being able to set these parameters the software will also need to turn both outputs of the function generator off and on.

Achieving function generator control was made relatively simple, namely thanks to the standardisation of instrument control provided by the Virtual Instruments Software Architecture (VISA) [3], and the Standard Commands For Programmable Instruments (SCPI) [4]. VISA manages communication with instruments like function generators over various I/O interfaces, including USB which was the interface used in this project. SCPI defines the syntax of ASCII text commands that can be used to control measurement instruments. The Rigol DG1022 dual output function generator used for this project is compatible with both these standards. PyVISA is the python implementation of the VISA standard and was used to control the function generator using SCPI commands [5]. The final version of the function generator code comprises of 3 functions, one to turn both output channels on, another to turn them off, and a final function that sets the amplitude and frequency of the signals based on the desired input from the user.

1.4.2 Motor Control

The functions that the motors need to perform are fairly simple; they need to be able to move to a desired position, at a desired speed, along both axis. The software also needs to know when the motors have arrived at the instructed position. Despite this simplicity, programming the motors was a more iterative process than the function generator control, as it did not have an equivalent standardised solution. The motors chosen could be controlled by writing assembly-like programs using TMCL's command language in a provided integrated development environment (IDE) [6]. the user can write commands in the specified TMCL command format, assemble them to .bin files, and then download them to the motors. All the functionality required from the motors could be implemented in this IDE, however there are many other facets to the scanner software other than motor control, and with a standalone IDE there is no way to integrate with other code files. So, while this IDE was useful for motor setup and troubleshooting, namely installing the correct firmware and confirming the motors were functioning correctly, a different solution would need to be found for the final software.

The first potential solution tested to try and solve this was another tool provided by TMCL. This took the form of an executable file that could be run from the command line to send assembled .bin files to the motors [7]. The IDE could be used to create .bin files of predefined motor movements, and then the OS module in Python could be used to execute a command line command that sends the .bin file to the motor module and runs it using the download tool. This process has been illustrated in the functional diagram below for clarity.

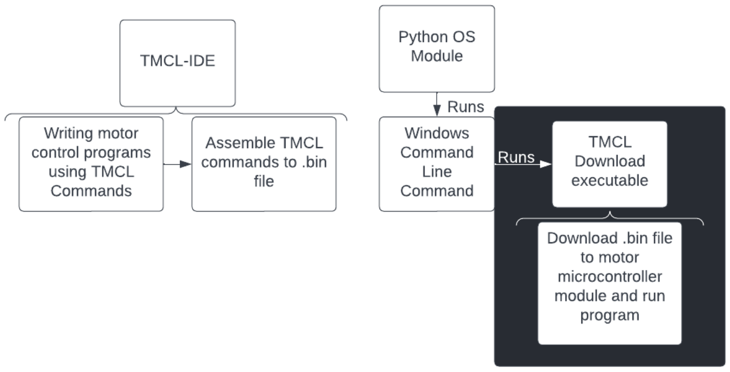
TMCL-IDE
Writing motor control programs using TMCL Commands
Assemble TMCL commands to .bin file
Python OS Module
Runs
Windows Command Line Command
Runs
TMCL Download executable
Download .bin file to motor microcontroller module and run program

Figure 3: Functional Diagram of Scanner Software and Associated Hardware Devices

The functional diagram in Figure 3 should also serve to illustrate some of the issues with this solution. Firstly are the tasks performed by the TMCL-IDE. The creation of the .bin files must be completed prior to running the scanning software, as all the Python can do is send pre-existing .bin files to the motors, it cannot create them. Therefore in these .bin files a certain range of motor movements would have to be pre-empted, and if the user wanted to use the motors in a way that fell outside of those pre-defined movements this would not be possible.

The second issue is with the TMCL download tool, and the problem is illustrated literally in the figure. The TMCL Download tool is a black box, the Python program does not know that the file has been successfully sent to the motor, it cannot know whether it has been executed successfully, or if the motors have returned any errors. The last and least significant issue is that as the TMCL Download application only exists as a .exe file it would limit the scanner software to being Windows only.

While it would have been possible to find workarounds to some of these issues, ultimately the solution would have been inelegant and limiting to the final scanner software. Due to its relatively simplicity its viability was established with some test .bin files, so it existed as a fall back if a superior solution was not found.

One of the reasons Python was chosen was because of the relative ease by which you can import C code to use with Python; and while Trinamic do not provide an application programming interface (API) for programming their products in Python, they do provide C skeleton code for sending single commands to the motors [8]. This skeleton code provides definitions for the TMCL commands' opcodes, and 2 functions. One function to put each command parameter into a buffer and compute a checksum, and one to read the returned parameters from the motor and confirm the checksum of the received data is correct. Significant additions needed to be made to this code in order for it to perform the required operations for the scanner software. The most important element to add was the code to actually send and receive commands from the motors over the USB ports of the computer. The serial communication was implemented using the Win32 API on Windows [9].

The firmware manual [10] provided all the information needed with regards to the parameters necessary to input into each command, as well as the meaning of the status codes returned by the motors. These commands were then put into functions to perform the appropriate control of the motors based on user inputted parameters: Changing the speed of the motors when desired, moving the motors to any desired position, and knowing when the motors have arrived at that position. The .c file containing the functions was then compiled to a dynamic link library file which could then be imported into Python using the foreign function library Ctypes [2].

1.4.3 PicoScope

The PicoScope 5442D is the high resolution ADC that converts the output signal from the sensor circuit into digital values in order to perform signal processing in software. There are a number of parameters which may need to be varied depending on the material under test. Most importantly, the sampling frequency needs to be above the Nyquist criterion [11] in order to get an accurate digital representation of the analogue signal. Alongside the sampling frequency, the PicoScope allows selection of resolution and voltage range, which again may need to be varied depending on the material under test. For example the changes in magnetic field of CFRPs are much smaller than for titanium or other metals, and may require higher resolution and a lower voltage range. As with the function generator the software will also need to be able to toggle the PicoScope on and off as necessary.

The PicoScope was programmed using the picoSDK for Python [12]. The picoSDK facilitated directly setting most of the necessary ADC parameters without any difficulty, such as sampling frequency, range and resolution. The PicoScope has two main modes in which it can operate, streaming mode and block mode. Streaming mode is the simpler of the two, the data collected by the PicoScope is passed directly to the PC with no memory buffers in between. In contrast, in block mode the PicoScope takes a certain number of values and stores it in internal memory, and then when that number of values has been taken it transfers the data to the computer [13].

Block mode was initially appealing, as each block of data could correspond to a row of scanning data, and this segmentation could have been beneficial to processing the data. However it also adds additional complexity over streaming mode as the storage capacity of the PicoScope has to be taken into account as well as any delays involved in transferring the data from the PicoScope memory to the computer. It was decided to first test streaming mode, and if performance issues were encountered regarding the host computer's ability to process the constant stream of data, block mode could have been used instead. As it turned out there was no issues with processing the data from streaming mode, and the operation of the PicoScope could be controlled to activate streaming mode for each row, so the segmentation possible with block mode was defunct anyway.

The PicoSDK provided example code [14] for using the PicoScope in streaming mode, and a lot of this did not require modification. The main alteration made was removing the conversion from raw ADC values into voltage. This was removed as this conversion process took a significant amount of time and it was found the signal processing worked equally well on the raw ADC values as the true voltage values. Other alterations were made but these come under the wider data flow in the scanning code and this will be discussed in the scanner control section as they are not relevant to the operation of the PicoScope itself.

1.5 3rd Party Software Libraries

Alongside the libraries mentioned above used to implement the hardware control, there were a number of 3rd party libaries used to assist with the creation of this software. A full list of all the libraries used, their functions and which components they were used for can be seen in appendix I.

1.6 Scanner Control

Following completion of each of the individual functions of the software, the next task was to integrate these blocks together into a single functioning system that successfully executed the scanning process. The flow diagram shown below in Figure 4 illustrates how this was done.

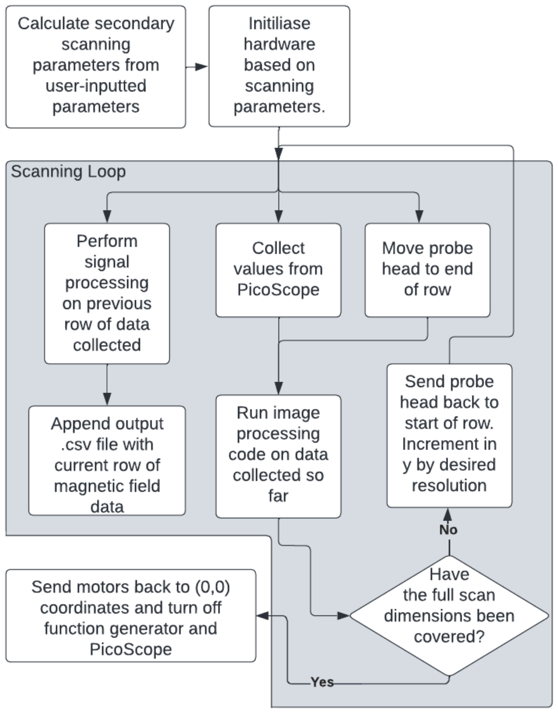
Calculate secondary scanning parameters from user-inputted parameters
Initiliase hardware based on scanning parameters.
Scanning Loop
Perform signal processing on previous row of data collected
Collect values from PicoScope
Move probe head to end of row
Append output .csv file with current row of magnetic field data
Run image processing code on data collected so far
Send probe head back to start of row. Increment in y by desired resolution
No
Have the full scan dimensions been covered?
Send motors back to (0,0) coordinates and turn off function generator and PicoScope
Yes

Figure 4: Flow diagram of scanner control software

Function Generator	PicoScope	Motors
Frequency and amplitude of coil and sensor signals (kHz and Vrms)	Sampling period and timebase	Motor speed (internal motor units) Scan dimensions (Width (mm) x length (mm) Y Axis resolution (mm)

Table 1: Table of primary scanning parameters

The first step of the scanning process is to take the inputted parameters from the user and use them to initialise the hardware. The key parameters are shown in Table 1. Note the lack of X axis resolution as a parameter; the reason for this is explained in Section **??**. There are some secondary parameters that need to be derived from the primary user parameters, these are defined in appendix I.

Once the user parameters have been passed to the hardware the main scanning loop, delineated by the grey box in Figure 4, can commence. The PicoScope begins logging values from the output of the sensor circuit, while the motors move along a row of the material. Unless this is the first row of the scan to be executed, the signal processing is executed on the previous row of data to be collected. This means that upon completion of a row (n) the image processing can be run on the previous (n-1) rows. This means that the output of the scan can be viewed by the user in almost real time. Upon completion of a row, The probe head is sent back to the start of the row, and the motors move the probe by the desired increment in the Y axis. Once the scan is complete, the code runs a shutdown routine where the motors send the probe head back to it's zero position, the PicoScope is turned off, and the function generator switches off it's outputs.

There are a few important details to note with regards to how this specific implementation came to be. From Figure 4 it should be evident that concurrency is an essential element of the scanning loop, as the software must collect data from the PicoScope, while also controlling the motors. Successfully implementing this concurrency was integral to the successful functioning of the software as a whole, so was the first aspect of the scanner control code to be tested. Not only does this need to happen concurrently but the two devices also needed to be synchronised. This is because the PicoScope needs to be taking measurements at different, ideally evenly spaced, points along the material in order to form an image.

If the PicoScope and the motors are not in sync then there will be a certain amount of time where the PicoScope is taking values when the sensor head is not moving. This would then mean multiple pixels would be representing the same part of the material and result in a misleading output from the scan. In order to synchronise the PicoScope and the motors, as well as implementing their concurrent operation, the PicoScope function which commenced data collection was put into a thread. This thread starts immediately before the motors are commanded to move to a position. The delay between the PicoScope thread beginning data collection and the motors moving is fortuitously so small that this fulfils the synchronisation requirements.

The next issue in implementing the scanning process was ensuring an efficient flow of data from the PicoScope to the final output of the scan. The initial versions of the scanner software wrote each row from the PicoScope output to a row of an output .csv file. This file would then be opened by the signal processing code, which would then write to another .csv file. Finally the image processing code would open this .csv file and produce the output image. This method quickly became a problem due to the amount of data involved. The PicoScope sampling frequency can be up to 15.6 MHz, meaning each row can contain millions of values. At first this process was implemented in the data manipulation tool Pandas [15], which meant the file also had to be opened each time to write a new row. This meant by the end of the scan it was taking minutes after each row to write to the file, which was untenable.

The file writing was then rewritten in C and imported using the same method as with the motor code. This facilitated appending to the file without having to open the entire file every time. This meant the write time did not compound as the number of rows increased but still meant a significant write time of about 10 seconds at the end of each row. The eventual solution was to have a third thread, inside the PicoScope threads' data collection function, that passed the data collected by the PicoScope directly to the signal processing code. This brings up a slight simplification in Figure 4. The signal processing code is presented like a separate thread that launches at the start of the scanning loop, but is in fact started within the PicoScope code when data collection has finished. These nested threads were too difficult to represent with any clarity in a diagram such as Figure 4, and the functioning of the scanner loop as represented in Figure 4 is still essentially correct.

1.7 Graphical User Interface

The GUI was created in PyQt5 [16], a cross-platform GUI tool that is freely available to make user interfaces for applications. Figure 5 below shows the layout of the GUI, with the scanning parameters tab on the left hand side and the Scan control and results tab on the right hand side. An issue with concurrency between the GUI thread and the scanning program is impeding the full functionality of the program.

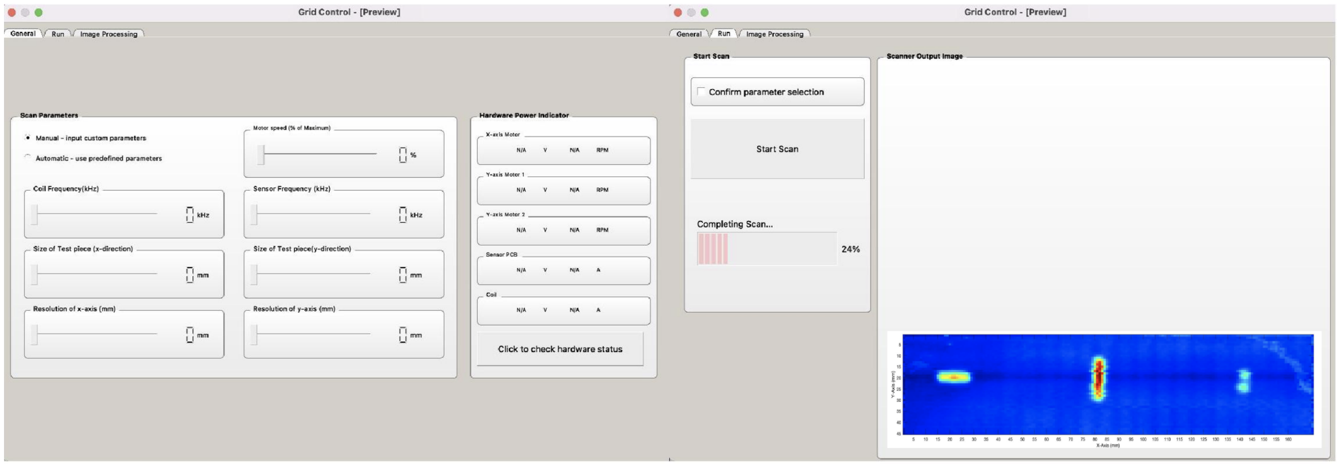

Figure 5: Layout of graphical user interface

1.8 Summary

The software implementation proved extremely successful in this project when assessed with regards to the design criteria outlined in the overview. Scanner control has been unified to a single point of control, and the user has complete control over the key parameters of the scanning process. Python proved a suitable choice of language, it's ability to interface with C was very useful for the motor programming, and the excellent range of libraries allowed effective implementation of the remaining hardware control as well as the signal and image processing. Not only did the software achieve the primary goal of controlling the scanning process and outputting the correct results, the use of concurrency enabled those scanning results can be viewed in almost real time.