FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

FH Aachen

Faculty Electrical Engineering and Information Technology

Information Systems Engineering

Master Thesis

Development of a hardware and software framework for the automated characterization of permanent magnets for low-field MRI systemsg

Submitted by	**Marcel Werner Heinrich Friedrich Ochsendorf**
	Matriculation number: **3120232**
Examiner:	Prof. Dr.-Ing. Thomas Dey
Examiner:	Prof. Dr.-Ing. Volkmar Schulz
Date:	01.01.2024

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe. Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht. Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Nachweis versehen. Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

I hereby certify that I have written this thesis independently and have not used any sources other than those listed in the bibliography. Any passages taken verbatim or in spirit from published or unpublished sources are labelled as such. The drawings or illustrations in this work have been created by myself or have been labelled accordingly. This thesis has not been submitted in the same or a similar form to any other examination authority.

Aachen, _____, _____

Abstract

A large number of magnets are used in the construction of low-field MRI equipment on the basis of permanent magnets. The magnetic properties of these magnets must be similar to a certain degree in order to achieve a homogeneous B0 field, which is necessary for many setups. Due to the complex manufacturing process of neodymium magnets, the different properties, i.e. the direction of magnetisation, can deviate from each other. This affects the homogeneity of the field. A passive shimming process is typically used to adjust the field afterwards. This is complex and time-consuming and requires manual corrections to the magnets used. To avoid this process, magnets can be systematically measured in advance. Data acquisition, storage and subsequent analysis play an important role in this methodology. Several existing open source solutions implement individual parts, but do not provide a complete data processing pipeline from acquisition to analysis, and their data storage formats are not compatible with each other. For this use case, the MagneticReadoutProcessing library has been created. It implements all important aspects of acquisition, storage and analysis, and each intermediate step can be customised by the user without having to create everything from scratch, thus encouraging exchange between different user groups. Complete documentation, tutorials and tests enable users to use and adapt the framework as quickly as possible. The framework was used to characterise different magnets, which requires integrating magnetic field sensors.

List of abbreviations

CAD Computer Aided Design. 9, 23

CDC Communication Device Class. 12

CLI Command Line Interface. 12, 13, 17, 32, 34, 35, 43–46, 52, 56

GPIO General Purpose Input/Output. 10, 15, 21

GUI Graphical User Interface. 44

HAL Hardware Abstraction Layer. 29, 32

HTML Hypertext Markup Language. 53, 54

I2C Inter-Integrated Circuit. 8, 12

IC Integrated Circuit. 9, 10

IDE Integrated Development Environment. 49, 53

IP Internet Protocol. 33

JSON JavaScript Object Notation. 23, 27, 28, 38

LUT Lookup Table. 12, 34–36

MRI Magnetic Resonance Imaging. 23, 24

MRP MagneticReadoutProcessing. 7, 13, 15, 17, 34, 37, 39, 44, 46, 49–52, 54, 55, 62

OS Operating System. 15

PC Personal Computer. 9, 10, 12–14, 20, 21, 30–34, 38, 45, 50

PCB Printed Circuit Board. 10

PDF Portable Document Format. 53

PIP Python Package Installer. 51

PPS Puls Per Second. 34

PTP Precision Time Protocol. 34

REST Representational State Transfer. 31, 32

SBC Single Board Computer. 15, 21, 31, 34

UART Universal Asynchronous Receiver / Transmitter. 12, 21

USB Universal Serial Bus. 12, 14, 15, 29, 34

UUID Universally Unique Identifier. 17, 28, 34, 36, 39

Contents

1 Introduction

1.1 Background and Motivation

1.1.1 Low-Field MRI

1.1.2 Shimming Procedure

Der Shimming-Prozess ist ein entscheidender Schritt bei der Magnetresonanztomographie (MRT), um homogene Magnetfelder für präzise Bildgebung zu gewährleisten. Shimming korrigiert Unregelmäßigkeiten im statischen Magnetfeld, die durch äußere Einflüsse oder interne Systemfehler entstehen können. Dieser Prozess optimiert die Feldhomogenität, was für hochauflösende und artefaktfreie Bilder entscheidend ist.

Die Quellen für den Shimming-Prozess können Hardware- und Software-basiert sein. Hardware-Shimming beinhaltet den Einsatz von Gradienten- und Radiofrequenzspulen, die gezielt platziert werden, um das Magnetfeld auszurichten. Software-Shimming hingegen verwendet Algorithmen, um die Steuerparameter des MRT-Systems anzupassen und die Homogenität zu verbessern.

Forschungspublikationen wie “Shimming Techniques in Magnetic Resonance Imaging” (DOI: 10.1007/978-1-4939-2815-6_3) bieten detaillierte Einblicke in verschiedene Shimming-Methoden und Techniken. Diese Fortschritte in der Shimming-Technologie tragen dazu bei, die diagnostische Genauigkeit und Bildqualität in der MRT erheblich zu verbessern.

1.1.3 Charakterisation of Permanent Magnets

- zeigt ansätzte einfacher magnet vermesseungen [19]
- contruction of hallbach ringe [18]

1.2 Aim of this Thesis

The present work aims to provide an efficient and comprehensive solution for the design of low-field MRI devices by developing and implementing a software and hardware framework.

The scope of the software library is to lay the foundation for the systematic characterization of magnets based on permanent magnets. The library will enable data acquisition, storage and analysis of magnetic properties, with customization possible at each step of the process.

The work aims to facilitate magnetic field characterization and improve the exchange of data between different user groups. Complete documentation, tutorials and tests will enable users to use the framework efficiently and adapt it to their specific requirements.

The application of the developed framework for the characterization of different magnets and the integration of magnetic field sensors serve as practical applications and validation of the developed solution.

1.3 Structure

This work is divided into six main chapters, which deal with the approach and implementation. The techniques and concepts used are explained in detail. Specific examples provide an overview of the possible use of the developed solution by the user.

- Chapter 2. **State of the Art** describes the current state of technology and forms the basis for further development. Existing measurement methods and findings

are shown here in order to define the context for the current project.

- Chapter 3. **Usecases** are the application cases in which the project is to be used. They illustrate the practical scenarios and define how the product can be used in the real world. This is crucial for understanding the needs of the target group and designing the end result accordingly.

- Chapter 4. **Unified Sensor** refers to the integration of different sensors into a standardised solution. This enables simple data acquisition and serves as a basic hardware system on which the subsequent data processing library can be applied.

- Chapter 5. **Software Readout Framework** describes the implementation of the data readout framework. This includes an explanation of the various modules and specific application examples.

- Chapter 6. **Usability Improvements** refers to additional activities to improve user-friendliness. This includes the optimisation of interfaces, interactions and processes to ensure intuitive and efficient use of the product. This also includes the documentation of code and the distribution of the source code as a package to users.

- Chapter 7. **Evaluation** comprises the systematic review and assessment of the overall system. This includes the demonstration of the implemented capabilities of the overall system against the previously defined usecases.

2 State of the Art

2.1 Opensource Projects

2.2 Conceptual Design

- etnwicklung eines hardware uns software framework zur einfachen aquirierung von Meagnetfelddaten
- analysetools und funktionen

accomplished

3 Usecases

The following section defines some usecases that the future project should be able to cover. These illustrate practical situations to understand the functionality and added value of the implemented solution for the user. These were defined in the course of project planning and provide an overview of how the user interacts with the project and what functionalities can be expected. In the later accomplished evaluation process 7, the defined usecases are also used as a reference to demonstrate the implemented capabilities of the solution.

1. **Ready to use hardware sensor designs**

A universal, easy-to-integrate Hall sensor design allows users to evaluate the framework quickly and cost-effectively. The pre-built hardware sensors provide an optimal solution for research, reducing development time and achieving repeatable measurement results. Once successfully evaluated, the Hall sensor design should be easily adaptable to other sensors without the need for major firmware changes.

2. **Taking automatic measurements from sensors**

The purpose of the framework is to enable the automated acquisition of measurement data from various connected hadware sensors. The user should be able to configure various measurement series, which should then be carried out by the framework without further user interaction.

3. **Open storage formats for data export**

The use of open storage formats for the export of data enables an interoperable data exchange environment. The implementation of standardized formats improves the portability and long-term availability of data. This encourages the exchange and further processing of measurement data in other software tools.

4. **Ready to integrate data analysis functions**

Once it is possible to record and store measured values, it should be possible for the user to analyse and visualize this data using various algorithms. The focus here should be on extending the framework with user-created algorithms.

5. **User programmable data processing pipelines**

User-programmable data processing pipelines enable the flexible design of data processing sequences as pipeline by users. The framework should enable users to create their own pipelines with the previously defined data analysis functions.

4 Unified Sensor

A defined main objective of this project is the development of a cost-effective magnetic field sensors interface that is also universally expandable. The focus is on mapping different sensors and being compatible with different magnet types and shapes. This ensures broad applicability in different scenarios. Another goal is reproducibility to ensure consistent results. Easy communication with standard PC hardware maximises the user-friendliness of the interface. The flexibility to support different sensors and magnets makes the system versatile and opens the possibility for use in different applications. A low-cost magnetic field sensors interface will therefore not only be economically attractive, but also facilitate the integration of magnetic field sensors in different contexts. In addition, the low-cost sensor interface will serve as a development platform for the data evaluation MagneticReadoutProcessing (MRP) library and provide real measurement data from magnets. In addition, the interface firmware creates a basis for the development of a data protocol for exchanging measured values. This makes it easy to integrate your own measuring devices into the MRP ecosystem at a later date. This is only possible with a minimal functional hardware and firmware setup and was developed for this purpose first.

4.1 Sensor Selection

The selection process for possible magnetic field sensors initially focussed on the most common and cost-effective ones, especially those that are already used in smartphones and are therefore widely available. A key aspect of this selection was the preference for sensors with digital interfaces to facilitate implementation in the circuit layout. The integration of integrated temperature sensors represents a significant enhancement that will later enable precise temperature compensation.

The use of analog sensors was purposefully avoided, though they are suitable for more precise measurements and extended measuring ranges. They were excluded

because they require more carefully designed circuits and more complicated energy management. In the context of the desired goal of developing a cost-efficient and universally expandable Hall sensor interface, the decision in favour of digital sensors seems appropriate.

Focussing on the digital Inter-Integrated Circuit (I2C) interface not only facilitates implementation, but also contributes to overall cost efficiency. At the same time, the integration of temperature sensors enables precise measurements under varying environmental conditions. This strategic choice forms the basis for a flexible, universally applicable Hall sensor interface that can be seamlessly integrated into various existing systems.

Tabelle 4.1: Implemented digital magnetic field sensors

	TLV493D-A1B6	HMC5883L	MMC5603NJ	AS5510
Readout-Axis	3D	3D	3D	1D
Temperature-Sensor	yes	no	yes	no
Resolution [uT]	98	0.2	0.007	48
Range [mT]	±130.0	±0.8	±3	±50
Interface	I2C	I2C	I2C	I2C

The table 4.1 shows a selection of sensors for which hardware and software support has been implemented. The resolution of the selected sensors covers the expected range of values required by the various magnets to be tested.

4.2 Mechanical Structure

The mechanical design of a sensor would be kept as simple as possible so that it can be replicated as easily as possible. The focus was on providing a stable foundation for

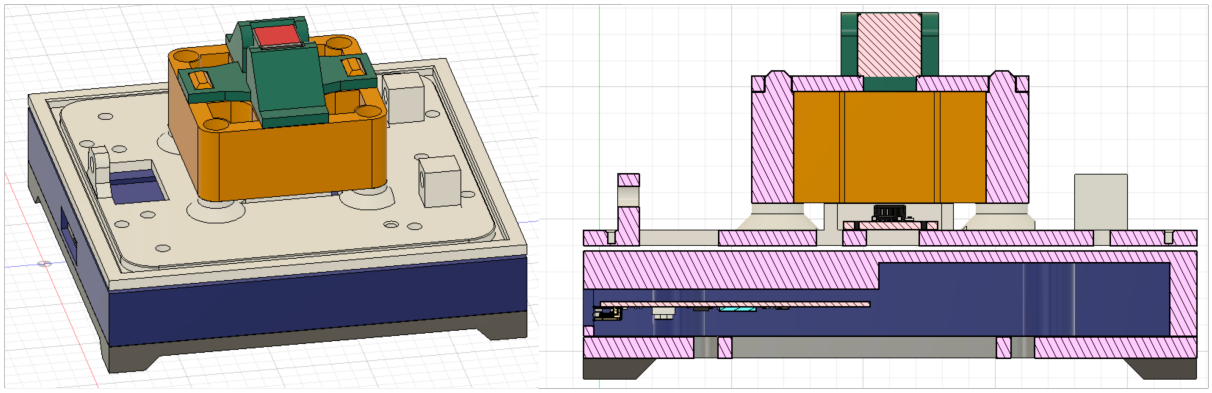

Figure **4**-1: 1D sensor mechanical 3D printed structure

the sensor Integrated Circuit (IC) and an exchangeable holder for different magnets.

The following figure **4**-1, shows a sectional view of the Computer Aided Design (CAD) drawing of the 1D-Single sensor 4.5.1.

All parts were produced using the 3D printing additive manufacturing processes. The sensor circuit board was glued underneath the magnet holder. This is interchangeable, so different distances between sensor and magnet can be realised.

The exchangeable magnetic holder (shown in green) can be adapted to different magnets. It can be produced quickly due to the small amount of parts used. The two recesses lock the magnet holder with the inserted magnet over the sensor. The specified tolerances allow the magnet to be inserted into the holder with repeat accuracy and without backlash. This is important if several magnets have to be measured, where the positioning over the sensor must always be the same.

4.3 Electrical Interface

The electronics consist of the magnetic field sensor and the electrical interface to connect it to a Personal Computer (PC) in the form of a microcontroller.

The focus here was on utilising existing microcontroller development and evaluation boards, which already integrate all the components required for basic operation. This not only enabled a time-saving implementation, but also ensured a cost-efficient realisation.

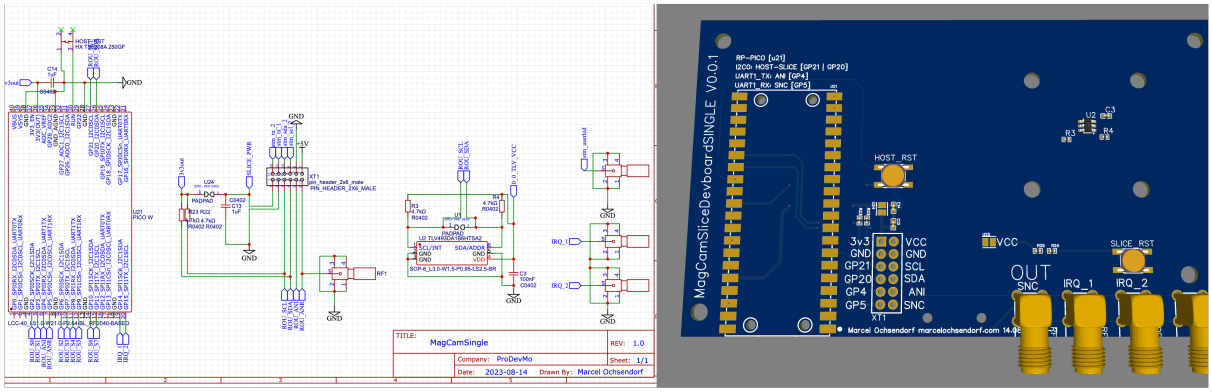

Figure **4**-2: 1D sensor schematic and circuit board

All the necessary components and their circuitry were then recorded on a Printed Circuit Board (PCB) **4**-2 and subsequently manufactured. In addition, footprints were provided for various sensor IC packages. By placing mounting holes on the PCB, it is possible to attach various mechanical mounts ontop of the sensor ICs.

Special attention was paid to the provision of an accessible SYNC-General Purpose Input/Output (GPIO) connector. This enables subsequent multi-sensor synchronization and also offers options for later extensions. This functionality opens up the possibility of synchronising data from different sensors to achieve precise and coherent measurement results. Overall, this integrated approach represents an effective solution for the flexible evaluation of sensors and helps to optimise the development process.

4.4 Firmware

The microcontroller firmware is software that is executed on a microcontroller in an embedded system. It controls the hardware and enables the execution of predefined functions. The firmware is used to process input data, control output devices and performs specific tasks according to the program code. It handles communication with sensors, actuators and other peripheral devices, processing data and making decisions. Firmware is critical to the functioning of devices.

The firmware is responsible for detecting the possible connected sensors 4.1 and query measurements. This measured data can be forwarded to a host PC via a user interface and can then be further processed there.

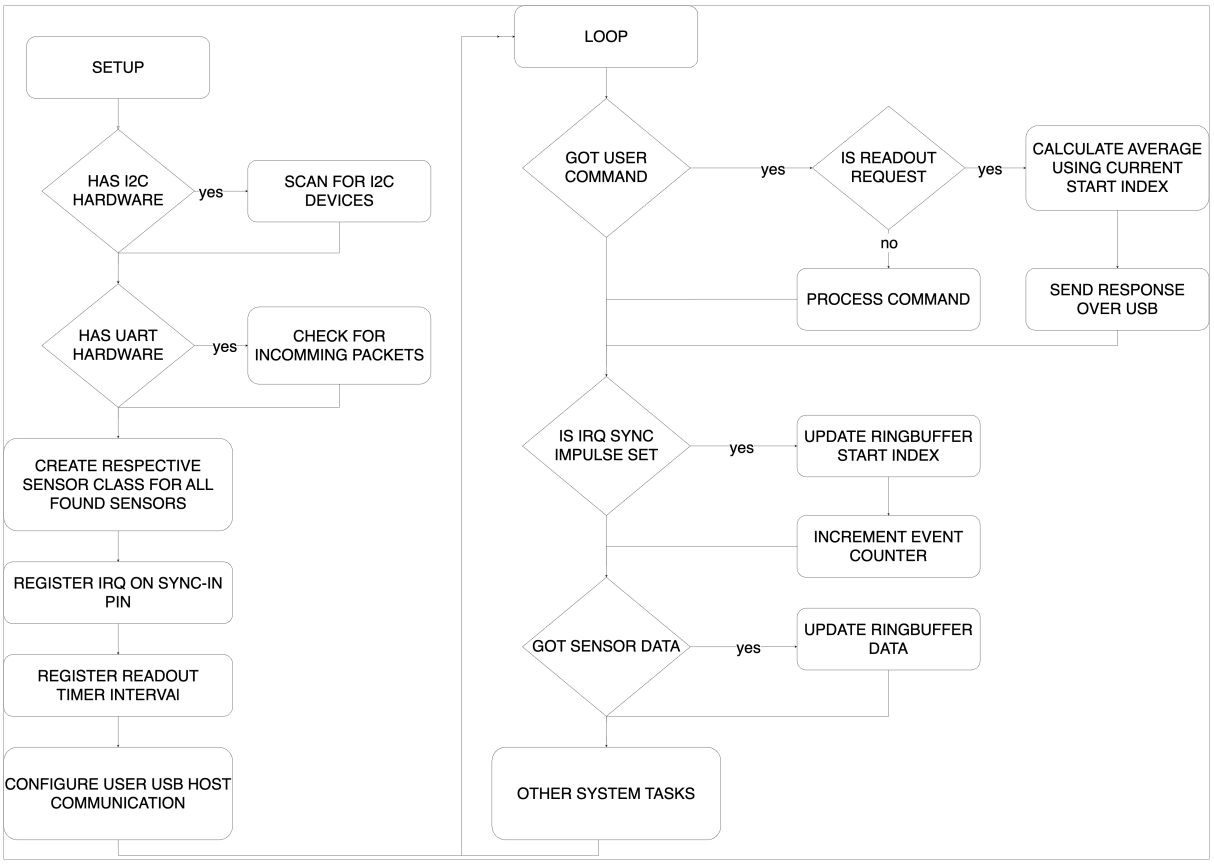

Figure **4**-3: Unified sensor firmware simplified program strucutre

An important component is that as many common sensors as possible can be easily connected without having to adapt the firmware. This modularity was implemented using abstract class design. These are initiated according to the sensors found at startup. If new hardware is to be integrated, only the required functions 4.1 need to be implemented.

```
#ifndef __CustomSensor_h__
#define __CustomSensor_h__
// register your custom sensor in implemented_sensors.h also
class CustomSensor: public baseSensor
{
public:
  CustomSensor();
  ~CustomSensor();
  // implement depending sensor communication interface
  bool begin(TwoWire& _wire_instance); // I2C
  bool begin(HardwareSerial& _serial_instance); // UART
  bool begin(Pin& _pin_instance); // ANALOG or DIGITAL PIN like onewire
  // FUNCTIONS USED BY READOUT LOGIC
  bool is_valid() override;
  String capabilities() override;
  String get_sensor_name() override;
```

```
17   bool query_sensor() override;
18   sensor_result get_result() override;
19 };
20 #endif
```

Listing 4.1: CustomSensor-Class for adding new sensor hardware support

The flow chart **4**-3 shows the start process and the subsequent main loop for processing the user commands and sensor results. When the microcontroller is started, the software checks whether known sensors are connected to I2C or Universal Asynchronous Receiver / Transmitter (UART) interfaces. If any are found (using a dedicated Lookup Table (LUT) with sensor address translation information), the appropriate class instances are created and these can later be used to read out measurement results.

The next initialisation system is dedicated for multi-sensor synchronisation 4.4.2. The last step in the setup is to configure communication with the host or connected PC. All implemented microcontroller platforms used (Raspberry Pi Pico, STM32F4) have a Universal Serial Bus (USB) slave port. The used usb descriptor is a USB Communication Device Class (CDC). This is used to emulate a virtual RS232 communication port using a USB port on a PC and usually no additional driver is needed on modern host systems.

After the setup process is complete, the system switches to an infinite loop, which processes several possible actions. One task is, to react to user commands which can be sent to the system by the user via the integrated Command Line Interface (CLI). All sensors are read out via a timer interval set in the setup procedure and their values are stored in a ringbuffer. Ring buffer offers efficient data management in limited memory. Its cyclic structure enables continuous overwriting of older data, saves memory space and facilitates seamless processing of real-time data.

Ring buffers are well suited for applications with variable data rates and minimise the need for complex memory management. The buffer can be read out by command and the result of the measurement is sent to the host. Each sensor measurement result is transmitted from the buffer to the host together with a time stamp and a sequential number. This ensures that in a multi-sensor setup with several sensors. The measurements are synchronized 4.4.2 in time and are not out of sequence or drift.

```
help
================================================================================================
> help                          shows this message
> version                       prints version information
> id                            sensor serial number for identification purposes
> sysstate                      returns current system state machine state
> opmode                        returns 1 if in single mode
> sensorcnt                    returns found sensorcount
> readsensor x <0-senorcount>  returns the readout result for a given sensor index for X axis
> readsensor y <0-senorcount>  returns the readout result for a given sensor index for Y axis
> readsensor z <0-senorcount>  returns the readout result for a given sensor index for Z axis
> readsensor b <0-senorcount>  returns the readout result for a given sensor index for B axis
> temp                          returns the system temperature
> anc <base_id>                 perform a autonumbering sequence manually
> ancid                         returns the current set autonumbering base id (-1 in singlemode)
> reset                         performs reset of the system
> info                          logs sensor capabilities
> commands                      lists sensor implemented commamnds which can be used by hal
================================================================================================
```

Figure **4**-4: Sensors CLI

```
readsensor b 0
3279.99
```

Figure **4**-5: Query sensors b value using CLI

4.4.1 Communication interface

Each sensor that has been loaded with the firmware, registeres on to the host PC as a serial interface. There are several ways for the user to interact with the sensor:

- Use with MRP 5-libaray
- Stand-alone mode via sending commands using built-in CLI

The CLI mode is a simple text-based interface with which it is possible to read out current measured values, obtain debug information and set operating parameters. This allows to quickly determine whether the hardware is working properly after installation. The CLI behaves like terminal programmes, displaying a detailed command reference **4**-4 to the user after connecting. The current measured value can be output using the readout command**4**-5.

The other option is to use the MRP 5-library. The serial interface is also used here. However, after a connection attempt by the MRPHal**??** module of the MRP 5-library, the system switches to binary mode, which is initiated using the sbm command. The same commands are available as for CLI-based communication.

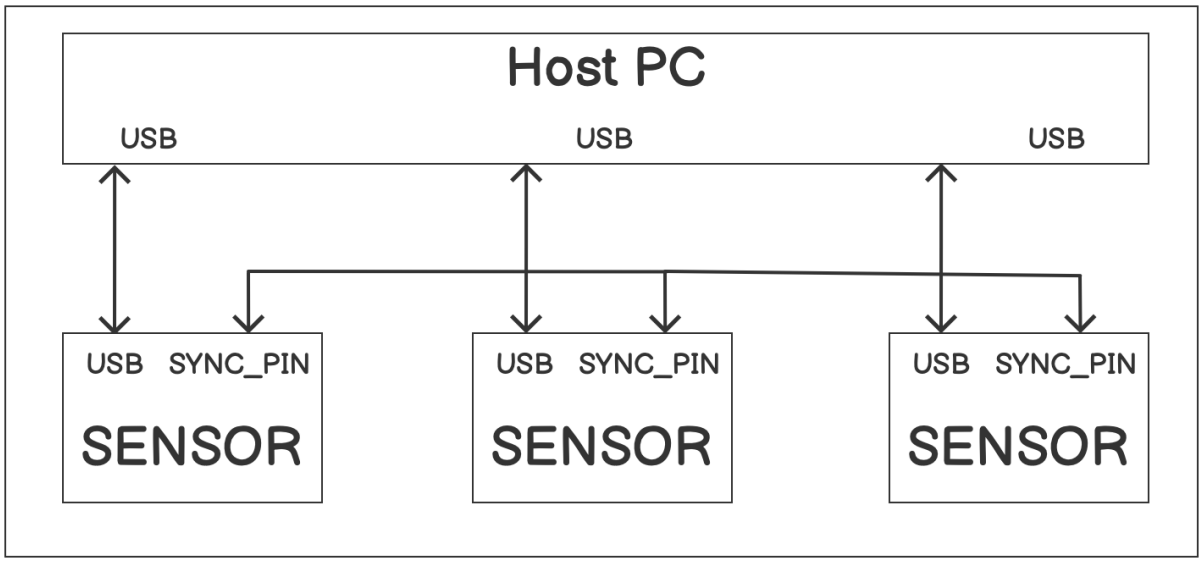

Figure **4**-6: Multi sensor synchronisation wiring example

4.4.2 Sensor Syncronisation Interface

One problem with the use of several sensors on one readout host PC is that the measurements may drift over time. On the one hand, USB latencies can occur. This can occur due to various factors, including device drivers, data transfer speed and system resources. High-quality USB devices and modern drivers often minimise latencies. Nevertheless, complex data processing tasks and overloaded USB ports can lead to delays.

Tabelle 4.2: Measured sensor readout to processing using host software

Data-Points	Run runtime [ms]	Average Sensor communication time per reading [ms]	Communication jitter time [ms]	Comment
1	9453	1.44	0	
1	9864	1.5	0	
10	12984	1.22	0.9	
10	12673	1.13	1.1	

Data-Points	Run runtime [ms]	Average Sensor communication time per reading [ms]	Communication jitter time [ms]	Comment
10	43264	2.19	8.2	96% system load

The table shows4.2 shows various jitter measurements. These were performed on a RaspberryPi 4 4GB-Single Board Computer (SBC) together with an 1D: Single Sensor4.5.1 and the following software settings:

- Raspberry Pi OS Lite - Operating System (OS) debian bookworm x64,
- MRP 5-library - Version 1.4.1
- Unified Sensor 4-firmware - Version 1.0.1

It can be seen that a jitter time of up to an additional 1ms is added between the triggering of the measurements by the host system and the receipt of the command by the sensor hardware. If the host system is still under load, this value increases many times over. This means that synchronising several sensors via the USB connection alone is not sufficient.

The other issue is sending the trigger signal from the readout software 5. Here too, unpredictable latencies can occur, depending on which other tasks are also executed on this port.

In order to enable the most stable possible synchronisation between several sensors, an option has already been created to establish an electrical connection between sensors. This is used together with the firmware to synchronise the readout intervals. The schematic **4**-6 shows how several sensors must be wired together in order to implement this form of synchronisation.

Once the hardware has been prepared, the task of the firmware of the various sensors is to find a common synchronisation clock. To do this, the function register irq on sync pin is overwritten. To set one primary and several secondary sensors, each sensor waits for an intial pulse on the SYNC-GPIO **4**-7. Each sensor starts a random timer beforehand, which sends a pulse on the sync line. All others receive this and switch to secondary mode and synchronise the measurements based on each sync pulse received. Since

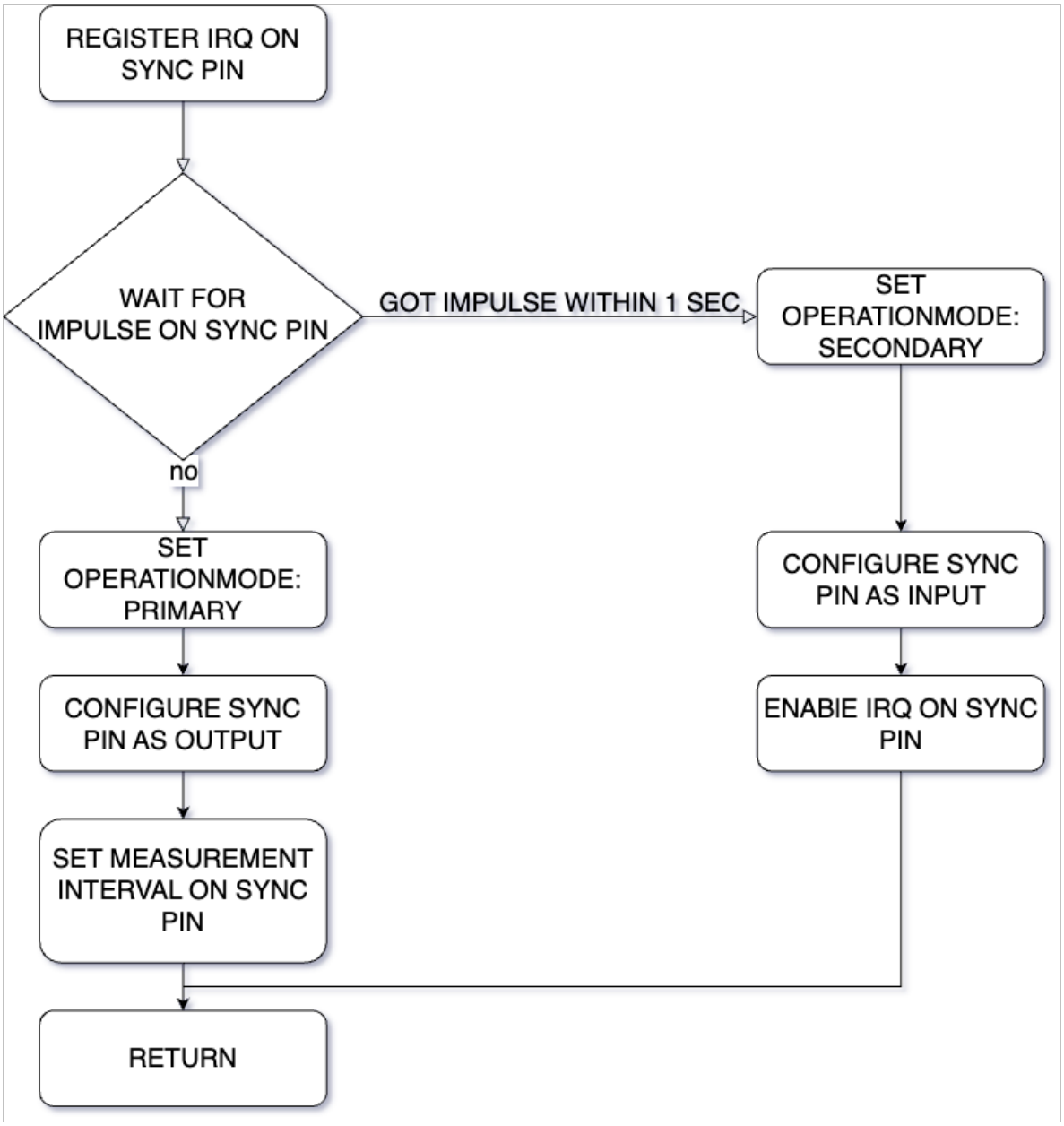

Figure **4**-7: Unified sensor firmware multi sensor synchronisation procedure

Figure **4**-8: Query opmode using CLI

the presumed primary sensor cannot register its own sync pulse (because the pin is switched to output), there is a timeout branch condition got pulse within 1000ms and this becomes the primary sensor. This means that in a chain of sensors there is exactly one primary and many secondary sensors. In single-sensor operation, this automatically jumps to primary sensor operation through the got impulse within 1000ms branch result. The synchronisation status can be queried via the user interface 4.4.1 using the opmode **4**-8 command. An important aspect of the implementation here was that there is no numbering or sequence of the individual sensors. This means that for the subsequent readout of the measurements, it is only important that they are taken at the same interval across all sensors. The sensor differentiation takes place later in the MRP 5-library by using the sensor Universally Unique Identifier (UUID).

4.5 Example sensors

Two functional sensor platforms 4.3 were built in order to create a solid test platform for later tests and for the development of the MRP 5-library with the previously developed sensor concepts.

Tabelle 4.3: Build sensors with different capabilities

	1D4.5.1	1D: dual sensor	3D: Fullsphere4.5.2
Maximal magnet size	Cubic 30x30x30	Cubic 30x30x30	Cubic 20x20x20
Sensor type	MMC5603NJ	TLV493D	TLV493D
Sensor count	1	2	1
Scanmode	static (1 point)	static (2 points)	dynamic (fullsphere)

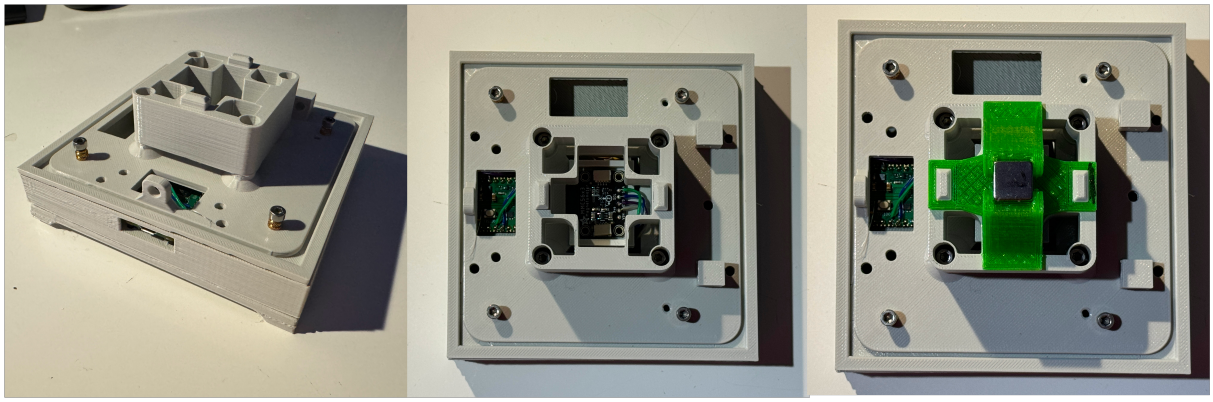

Figure **4**-9: 1D sensor construction with universal magnet mount

These cover all the required functions described in the usecases 3. The most important difference, apart from the sensor used, is the scan mode. In this context, this describes whether the sensor can measure a **static** fixed point on the magnet or if the sensor can move dynamically around the magnet using a controllable manipulator.

In the following, the hardware structure of a **static** and dynamic sensor is described. For the **static** sensor, only the 1D variant is shown, as this does not differ significantly from the structure of the 1D: dual sensor, except it uses two TLV493D sensors, mounted above and on top of the magnet.

4.5.1 1D: Single Sensor

The 1D sensor **4**-9 is the simplest possible sensor that is compatible with the Unified Sensor firmware 4.4 platform.

The electrical level here is based on a Raspberry–Pi Pico together with the MMC5603NJ magnetic sensor. The mechanical setup consists of four 3D printed components, which are fixed together with nylon screws to minimise possible influences on the measurement.

Since the MMC5603NJ only has limited measurement range of total 6uT, even small coin sized neodymium magnets already saturates the sensor. It is possible to mount 3D printed spacers over the sensor to increase the distance between the magnet and the sensor and thus also measure these magnets.

The designed magnet holder can be adapted for different magnet shapes and can

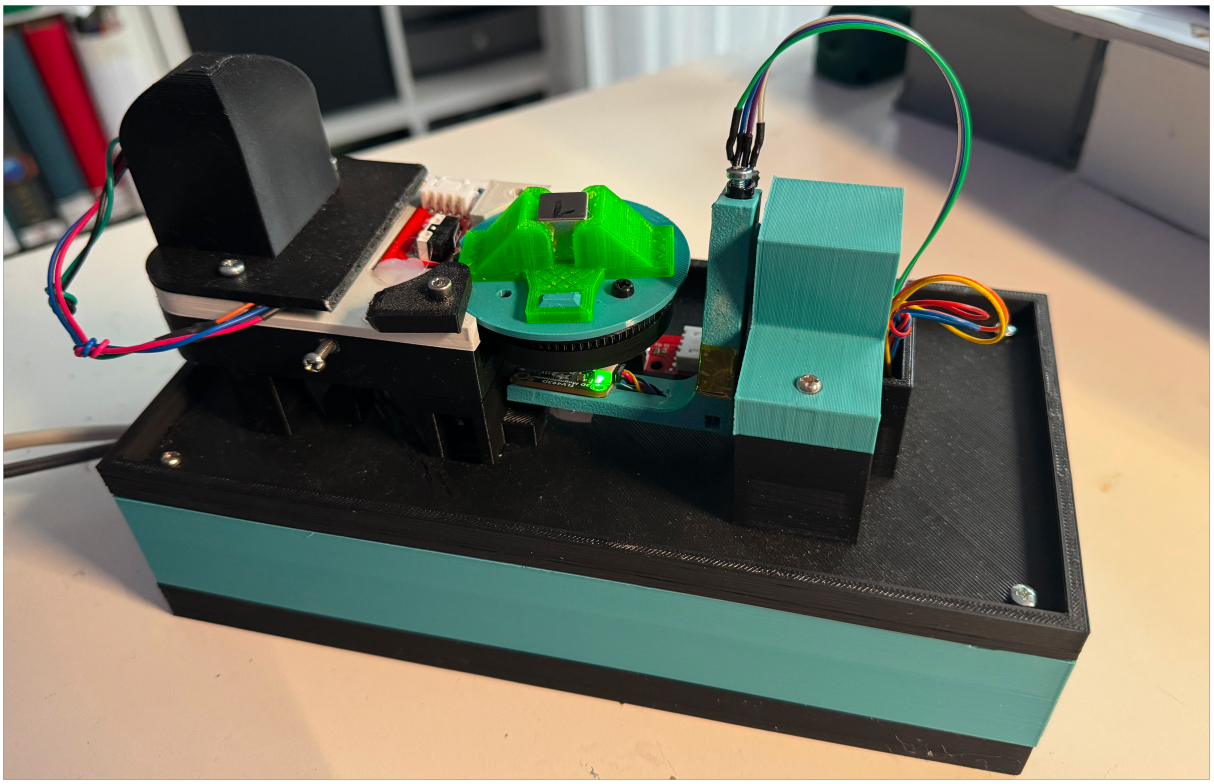

Figure **4**-10: Full-Sphere sensor implementation using two Nema17 stepper motors in a polar coordinate system

be placed on the spacer without backlash in order to be able to perform a repeatable measurement without introducing measurement irregularities by mechanically changing the magnet.

4.5.2 3D: Fullsphere

The 3D fullsphere sensor **4**-10 offers the possibility to create a 3D map of the inserted magnet.

The graphic **4**-11 shows the visualisation of such a scan in the form of a spherical 3D map. On the sphere is the magnetic field strength, which was detected by the sensor at the position. The transition from a fully positive field strength (red) to a negative field strength (blue) is clearly recognisable and corresponds to the orientation of the magnet in the holder.

The magnet sensor is mounted on a movable arm, which can move 180 degrees around the magnet on one axis. In order to be able to map the full sphere, the magnet is mounted on a turntable. This permits the manipulator to move a polar coordinate

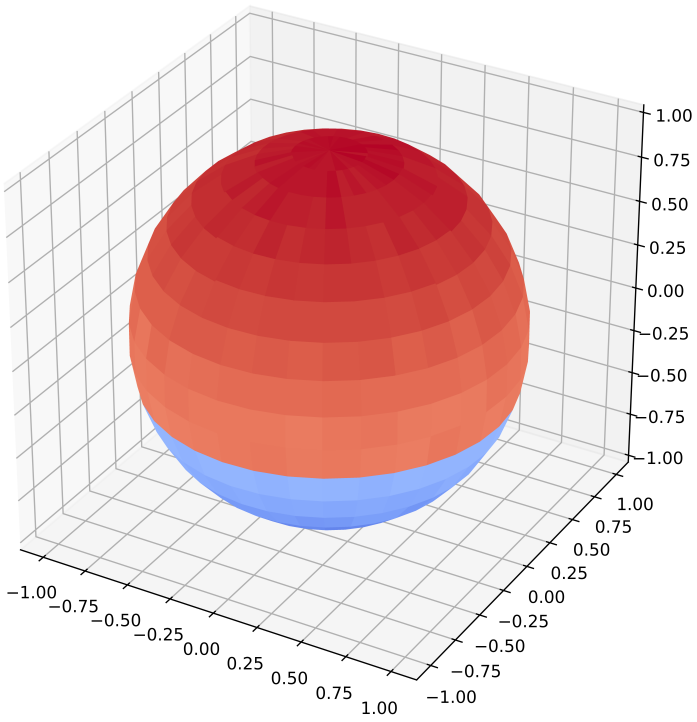

Figure **4**-11: 3D plot of an N45 12x12x12 magnet using the 3D fullsphere sensor

system.

As the magnets in the motors, as with the screws used in the 1D sensor, can influence the measurements of the magnetic field sensor, the distance between these components and the sensor or magnets was increased. The turntable and its drive motor are connected to each other via a belt.

On the electrical side, it also consists of a SKR–Pico stepper motor controller, together with the TLV493D magnetic field sensor. This was chosen because of its larger measuring range and can therefore be used more universally without having to change the sensor of the arm.

4.5.3 Integration of an industry teslameter

As the sensors shown so far relate exclusively to self-built, low-cost hardware, the following section shows how existing hardware can be integrated into the system. A temperature-compensated Voltcraft GM–70 telsameter **4**-12 is used, which has a measuring range of 0–3T with a resolution of 0.1mT. It offers an RS232 interface with a documented protocol 4.4 for connection to a PC. This connectivity makes it possible to make the device compatible with the Unified Sensor ecosystem using a separate

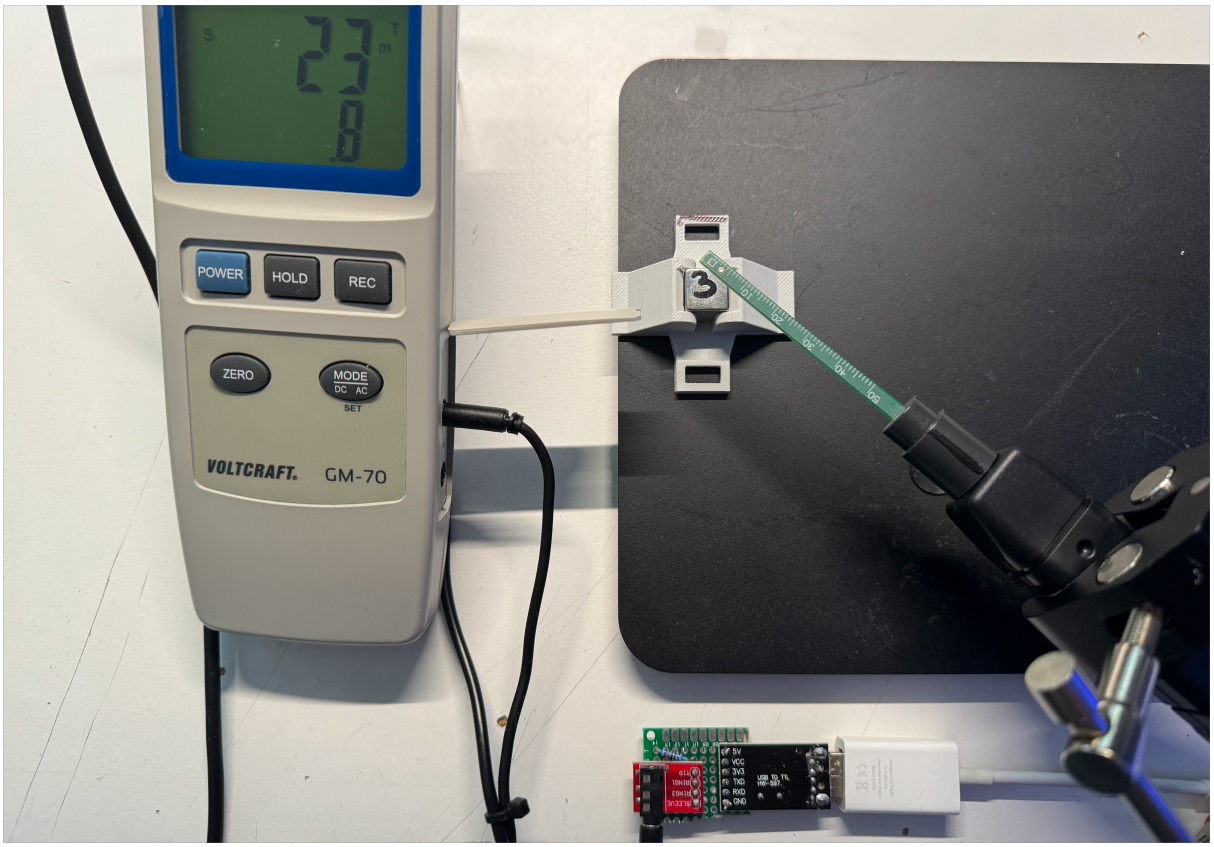

Figure **4**-12: Voltcraft GM70 teslameter with custom PC interface board

interface software [15] executable on the host PC. However, it does not offer the range of functions that the Unified Sensor firmware offers.

Another option is a custom interface board between the meter and the PC. This is a good option as many modern PCs or SBCs no longer offer an physical RS232 interface. As with the other sensors, this interface consists of your RaspberryPi Pico with an additional level shifter. The Teslameter is connected to the microcontroller using two free GPIOs in UART mode. The firmware was adapted using a separate build configuration. In order to be able to read and correctly interpret the data from the microcontoller, the serial protocol from table 4.4 of the sensor was implemented in a customized version of the CustomSensor class 4.1.

Tabelle 4.4: Voltcraft GM70 serial protocol

BYTE-INDEX	REPRESENTATION	VALUE
0	PREAMBLE	0x2

BYTE-INDEX	REPRESENTATION	VALUE
1		0x1
2		0x4
3	UNIT	‘B’ => Gauss ‘E’ => mT
5	POLARITY	‘1’ => *0.1 ‘2’* => 0.01
6	value MSB	0x-0xFF
13	value LSB	0x-0xFF
14	STOP	0x3

This software or hardware integration can be carried out on any other measuring device with a suitable communication interface and a known protocol thanks to the modular design.

5 Software readout framework

The software readout framework is the central software component that was developed as part of this work. This software framework is intended to provide a user-oriented data acquisition and analysis environment. For this purpose, typical individual steps that occur in relation to these tasks were implemented:

- Data acquisition - e.g. from hardware sensors 4
- Storage - export of data in various open formats 5.2.1
- Analysis - algorithms to analyze different data sets 5.2.2

All these possible task parts were divided into different blocks and users were given the possibility of adding their own functionalities.

As the following, this concept is referred to as user interaction points5.1 and is explained using a concrete example.

5.1 User Interaction Points

A project called HalbachOptimization[**?**] implements a data analysis step and optimizes a magnetic field that is as homogeneous as possible within a circular section using given mechanical dimensions as input parameters of the magnets used. For this purpose, a mutation of the magnet positions and rotations is performed. The result is a list of positions for each magnet.

The HalbachMRIDesigner[1] opensource project, can generate basic CAD drawings for Magnetic Resonance Imaging (MRI) magnets in a Hallbach configuration. To do this, the number of magnets and additional parameters for the properties of the CAD model to be created are passed to the function provided as input parameters using a JavaScript

Object Notation (JSON) file. The result is then an OpenSCAD[13] based 3D model of the magnet holder.

As a result, there are two projects which are both suitable for the task of optimizing and creating Hallbach magnets for MRI applications. However, their data structures are not compatible with each other. However, they are executed manually one after the other in order to obtain a final result.

The library created is intended to solve one problem by providing standardized and flexible data structures for use with this form of magnetic field data. On the other hand, by separating the processing pipeline into defined sub-steps, it should be possible to make individual modules and as a result functionalities interchangeable by the user.

The implementation of the same functionalities looks as follows after using the library:

1. **Create a static set of magnets**

The input parameters of the HalbachOptimization[**?**] project are on the one hand the mechanical dimensions and the number of magnets to be used. We assume ideal magnets here. However, it should also be possible to import field data in a more measured form later on. Using the DataAquisition sub-step, it is possible to generate any number of ideal magnets.

2. **Add custom analysis processing step**

Next, the user creates his own analysis step in order to be able to call up its functions. In the case of the HalbachOptimization[**?**] project, the function signature of the start function must be changed. This receives the result of the previous step, in this case the generated magnet data. By optionally setting meta data in the universal bilbiothekt data type, constants can be replaced in the analysis function and made dynamically configurable. The return result also corresponds to this data type so that subsequent steps are compatible.

User interaction points represent the core concept of the developed library and are intended to provide user-friendliness on the one hand and the rapid development of own analysis and optimization algorithms on the other. For this purpose, the library was divided into individual modules, which are shown in the graphic **5**-1. In combination, these represent a typical measurement-analysis-evaluation workflow of data. For this

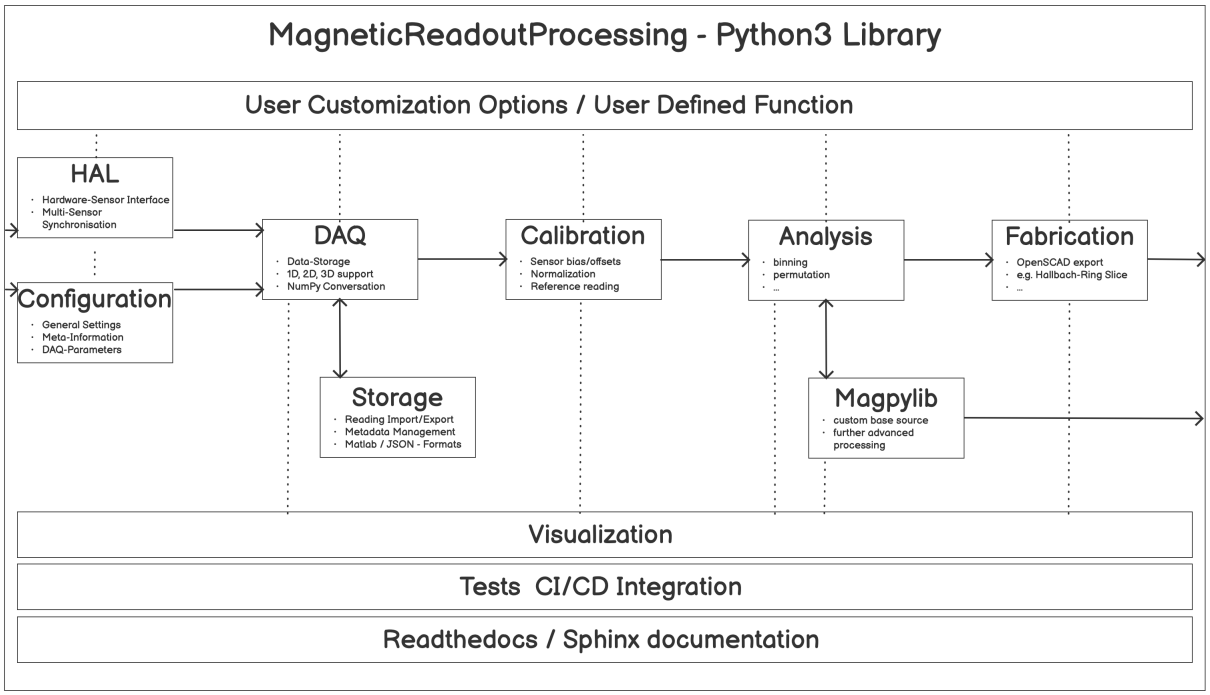

Figure **5**-1: MRP library module high level overview

purpose, a module system with standardised functional patterns and data types was developed and packed together in a extendable Python library.

According to this concept, the user should be able to replace individual components from this chain with their own modules without having to worry about implementing other of these to make the project work.

The individual modules and their functions are explained in detail in the following chapters.

5.2 Modules

In order to realise the concept of user interaction points, the library was divided into different modules. These can be divided into two main categories:

- Core - All modules related to measurement data management
- Extensions - Contains modules for visualizations, hardware sensor access

In each of these categories there are then several sub-categories divided into User

Interaction Points. An overview of these is given in the subchapters as the following. There are also introductory examples which provide an overview of the basic functions in the Examples5.4 section, as well as further examples in the online documentation[3].

5.2.1 Core Modules

The included core modules are essential for using the library. Basic data types are implemented, as well as functions for import and export. In addition, there are other support scripts that are required internally.

The following modules are implemented in detail:

- MRPReading - storage of measured values
- MRPMeasurementConfig - storage of the measurement parameters
- MRPMagnetTypes - various physical constants for basic magnet types

The MRPReading module performs a crucial role in streamlining the centralized management of measurement data. It serves as a storage provider for various measurements, offering functionalities that facilitate the creation and addition of data records. To customize and add meta-data, users have the flexibility to configure parameters through the dedicated MRPMeasurementConfig module into an MRPReading instance.

Within the realm of measurement data, a diverse range of data points can be seamlessly incorporated. The process is initiated by employing specialized functions designed for the creation and addition of data records. To configure parameters, ensuring a tailored approach to the entire measurement process, these parameters act as a crucial bridge between user preferences and the robust capabilities of the MRPReading module.

The system is also designed to be compatible with Extension Modules5.2.2, allowing the generation of measurement data through various modules. This extensibility enhances the versatility of the system, accommodating diverse measurement scenarios and expanding its utility across different domains.

To enhance the accessibility and interpretability of the recorded data, a dedicated module, MRPMagnetTypes, comes into play. This module is specifically designed for the storage of physical parameters pertaining to the magnets targeted for measurement. By centralizing this information, users can streamline the subsequent phases of evaluation

and analysis, simplifying the overall process and ensuring a more efficient and insightful exploration of the collected data.

At the end of processing, the collected and modified data are typically exported; various functions are provided for this purpose. This process is described in the following subchapter.

Storage and Datamanagement

An important aspect is data import and export. On the one hand, this allows the library to reuse and archive the measurements. On the other hand, the focus during development was that it is also possible to use the data in other programs. For this purpose, an open, documented export format must be selected. Ideally, this should be human-readable and viewable with a simple text editor. This eliminates all binary-based formats such as the Python pickle built into Python.

Taking these points into account, the JSON format was chosen. This is human and machine readable and there is a compatible parser for almost every programming language.

The following code snippet 5.1 shows the JSON structure which is generated when a measurement thatt using the library is exported. Here you can see that by using the JSON format, all measurement points and metadata are available in readable plain text. This means that they can also be read out in other programs. Using serialization, the MRPReading class inherited from Python-Object class is serialized via an dictionary conversion step. This JSON string can then be processed directly or written to the file system as a file.

```
1 {
2   "time_start": "Wed Sep 20 08:50:13 2023",
3   "time_end": "Wed Sep 20 08:54:13 2023",
4   "additional_data": {
5       "sensor_device_path": "/dev/cu.usbmodem3867315334391",
6       "sensor_name": "Unified Sensor Single Sensor",
7       "sensor_id": "386731533439",
8       "sensor_capabilities": ["static", "axis_b", "axis_x", "axis_y", "
     axis_z", "axis_temp", "axis_stimestamp"],
9       "configname": "calibrationtemp30.yaml",
10      "runner": "cli"
11  },
```

```
12   "data": [{
13       "value": 0.135,
14       "is_valid": true,
15       "id": 0,
16       "temperature": 34.32
17   }],
18   "measurement_config": {
19       "id": "525771256544952",
20       "sensor_distance_radius": 40.0,
21       "magnet_type": 0
22   },
23   "name": "calibrationtemp30",
24 }
```

Listing 5.1: JSON export structure of an MRPReading based measurement

The exported example 5.1 contains the following different object keys, which contain the following information:

- additional_data - Additional user-defined metadata
- data - Measured values with value and measured value UUID
- measurement_config - Information about the sensor used for measurement

In addition, further custom objects can be inserted into the JSON using the functions provided.

Since there are popular data processing frameworks such as Numpy[12], or the program for mathematical calculations, Matlab are often used, the library also supports export formats for these systems.

The different formats can be triggered by the user by calling up the corresponding MRPReading class functions:

- .dump() - JSON
- .to_numpy_matrix() - Numpy-Array of data object with different options
- .dump_savemat() - Matlab mat-file with measurement values and temperatures

Currently, data re-import of an exported measurement is only supported via the JSON format, as an export using the other options (Numpy, Matlab)loses data during the export procedure.

5.2.2 Extension Modules

The extention modules build on the core modules and offer the user additional basic functionalities. These include functions for data acquisition, visualization and analysis.

Sensor Interface

Another collection of optional modules provided by the library is the connection of external hardware sensors. All compatible sensors that are compatible with the firmware developed in the Unified Sensor4 chapter are supported here. The library provides the following sensor Hardware Abstraction Layer (HAL) modules for this purpose:

- MRPHal - Firmware protocol implementation
- MRPHalLocal - USB sensor interface
- MRPHalRemote - Remote sensor interface

These provide functions to communicate with a connected hardware sensor and send commands to it. To generate these and convert the received measurement data into the appropriate format for the core modules, there is a suitable module for each sensor type:

- MRPReadingSourceStatic - for 1D and 2D sensors such as 1D: Single Sensor4.5.1
- MRPReadingSourceFullsphere - for 3D sensors such as 3D: Fullsphere4.5.2

The decision which of these modules to use is made automatically depending on the connected hardware. For this purpose, a static function createReadingSourceInstance is implemented in the base class MRPReadingSource, which automatically creates the appropriate instance based on the sensor capabilites.

Visualisation

In order to give the user the possibility to display the recorded data visually, two modules were created, which can graphically prepare MRPReading instances:

- MRPVisualization - different table and graph based plots

- MRPPolarVisualization - fullsphere map plots

On the one hand, it is possible with the MRPVisualization module to display measurement data as a table or streamplot, lineplot. This makes it possible, for example, to visually identify outliers or trends in the measurement data. These can also be saved as an image file. The module is compatible with all measurement data.

In contrast to the MRPPolarVisualization module, this provides functions to create 2D map plots a polar coordinate system. This requires measurement data with additionally set spatial coordinates. These can be generated automatically with the 3D: Fullsphere4.5.2 sensor, or the user must provide the spatial information from another source.

Analysis

Data analysis offers the user the greatest flexibility to implement their own modules. For this reason MRPAnalysis contains functions for calculating the following data analyses, which are compatible with class instances of MRPReading:

- std_deviation - standard deviation
- mean - mean value
- variance - variance
- center_of_gravity - center of gravity
- binning - distribution of a sample by means of a histogram
- k–nearest - k-nearest neighbors

In addition, the export function .to_numpy_matrix enables further processing of the data in the Numpy[12] framework, in which many other standard analysis functions are implemented.

5.3 Multi Sensor Setup

At the current described scenarios, it is only possible to detect and use sensors that are directly connected to the host PC. It has the disadvantage that there must always be a physical connection. This can make it difficult to install multiple sensors in measurement setups where space or cable routing options are limited. So multibe sensor can be

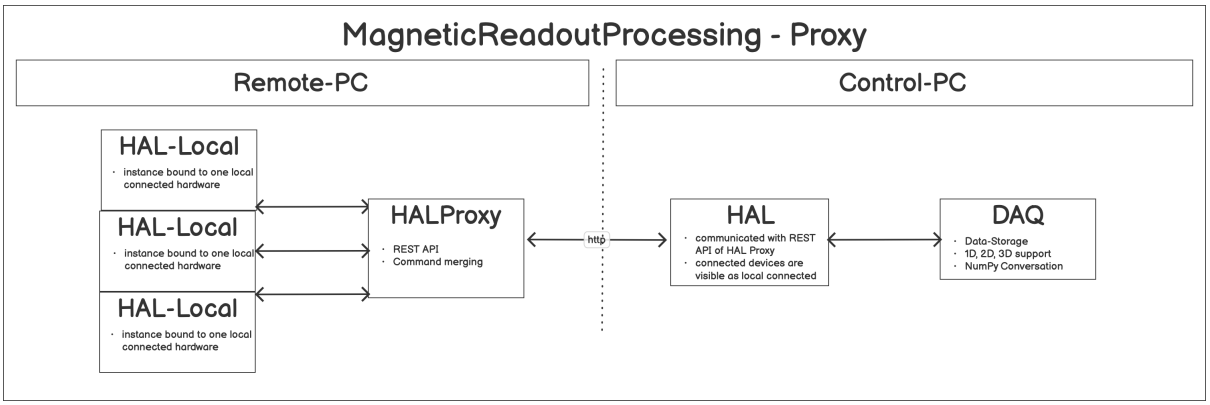

Figure **5**-2: MRPlib Proxy Module

connected to any PC which is available on the network. This can be a SBC (e.g. a Raspberry Pi). The small footprint and low power consumption make it a good choice. It can also be used in a temperature chamber.

The MRPProxy**5**-2 module has been developed to allow forwarding and interaction with several sensors over a network connection using a Representational State Transfer (REST) interface.

The approach of implementing this via a REST interface also offers the advantage that several measurements or experiments can be recorded at the same time with one remote sensor setup.

Another application example is when sensors are physically separated or there are long distances between them. By connecting several sensors via the proxy module, it is possible to link several instances and all sensors available in the network are available to the control PC.

The figure **5**-3 shows the modified multi–proxy – multi–sensor topology. Here, both proxy instances do not communicate directly with the control PC, but the proxy instance remotePC#2 can acces the proxy running on remotePC#1.

The control PC only communicates with the remote PC #1, but can access all sensors in this chain.

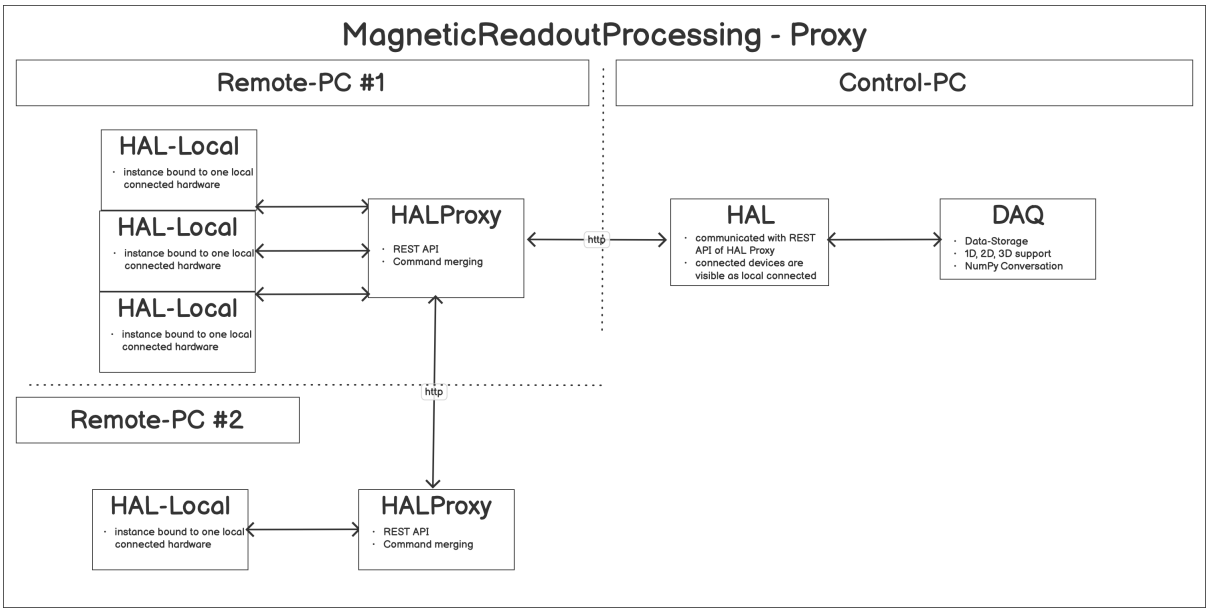

Figure **5**-3: mrp proxy multi

5.3.1 Network-Proxy

The figure **5**-2 shows the separation of the various HAL instances, which communicate with the physically connected sensors on the remote PC and the control PC side, which communicates with the remote side via the network. For the user, nothing changes in the procedure for setting up a measurement. The MRPProxy CLI application must always be started 5.2 on the PC with connected hardware sensors attached.

```
1 # START PROXY INSTNACE WITH TWO LOCALLY CONNECTED SENSORS
2 $ python3 mrpproxy.py proxy launch /dev/ttySENSOR_A /dev/ttySENSOR_B #
    add another proxy instance http://proxyinstance_2.local for multi-
    sensor, multi-proxy chain
3 Proxy started. http://remotepc.local:5556/
4 PRECHECK: SENSOR_HAL: 1337 # SENSOR A FOUND
5 PRECHECK: SENSOR_HAL: 4242 # SENSOR B FOUND
6 Terminate  Proxy instance [y/N] [n]:
```

Listing 5.2: MRPproxy usage to enable local sensor usage over network

After the proxy instance has been successfully started, it is optionally possible to check the status via the REST interface: 5.3

```
1 # GET PROXY STATUS
2 $ wget http://proxyinstance.local:5556/proxy/status
3 {
4 "capabilities":[
```

```
    "static",
    "axis_b",
    "axis_x",
    "axis_y",
    "axis_z",
    "axis_temp",
    "axis_stimestamp"
],
"commands":[
    "status",
    "initialize",
    "disconnect",
    "combinedsensorcnt",
    "sensorcnt",
    "readsensor",
    "temp"
]}
# RUN A SENSOR COMMAND AND GET THE TOTAL SENSOR COUNT
$ wget http://proxyinstance.local:5556/proxy/command?cmd=
    combinedsensorcnt
{
"output":[
    "2"
]}
```

Listing 5.3: MRPProxy REST endpoint query examples

The query result shows that the sensors are connected correctly and that their capabilites have also been recognised correctly. To be able to configure a measurement on the control PC, only the Internet Protocol (IP) address or hostname of the PC running an MRPProxy instance is required5.4.

```
# CONFIGURE MEASUREMENT JOB USING A PROXY INSTANCE
$ MRPcli config setupsensor testcfg --path http://proxyinstance.local
    :5556
> remote sensor connected: True using proxy connection:
> http://proxyinstance.local:5556 with 1 local sensor connected
```

Listing 5.4: MRPcli usage example to connect with a network sensor

5.3.2 Sensor Syncronisation

Another important aspect when using several sensors via the proxy system is the synchronisation of the measurement intervals between the sensors. Individual sensor setups do not require any additional synchronisation information, as this is communicated via the USB interface. If several sensors are connected locally, they can be connected to each other via their sync input using short cables. One sensor acts as the central clock as described in 4.4.2. this no longer works for long distances and the syncronisation must be made via a shared network connection.

If time-critical synchronisation over the network is required, Precision Time Protocol (PTP) and Puls Per Second (PPS) output functionality[10] can be used on many SBC, such as the Raspberry–Pi Compute Module 4.

5.3.3 Command-Router

As it is possible to connect many identical sensors to one host, it must be possible to address them separately. This separation is done by the MRPProxy module and is a separate part from the core MRP-library, to keep installation package dependencies small.

Each connected sensor is accessed via the text-based CLI, this is initially the same for each sensor. The only identification feature is the sensor UUID by using the id command of the sensor CLI.

The MRPProxy instance claims to be a sensor to the host PC running MRP CLI, so the multiple sensors must be combined into one virtual one. This is done in several steps, start procedure described by the following sub-chapters.

Construct the Sensor ID LookUp-Table

Immediately after starting the MRPProxy, the UUIDs of all locally connected sensors are read out. These are stored together with the class instance of the MRPHal module in a LUT. This makes it possible to address a sensor directly using its UUID.

Merging the sensor capabilities

Tabelle 5.1: Sensor capabilities merging

SENSOR A	SENSOR B	MERGED CAPABILITIES	CAPABLE SENSORS ID LUT
static		static	A
	dynamic	dynamic	B
axis_temp	axis_temp	axis_temp	A B
axis_x	axis_x	axis_x	A B

When using sensors with different capabilites, these must be combined. These are used to select the appropriate measurement mode for a measurement. For this purpose, the info command of each sensor is queried. This information is added to the previously created LUT. Duplicate entries are summarized (see Table 5.1) and returned to the host when the info5.5 command is received over network.

```
1 # QUERY Network-Proxy capabilities
2 $ wget http://proxyinstance.local:5556/proxy/status
3 {"capabilities":[
4 "static",
5 "dynamic",
6 "axis_temp",
7 "axis_x"
8 ]}
```

Listing 5.5: MRPproxy REST enpoiint query examples

The same procedure is performed for the commands CLI-command of each sensor, to merge available commands of connected sensors into the LUT.

Dynamic extension of the available network proxy commands

In order for the host to be able to send a command to the network multi-sensor setup, the command received must be forwarded to the correct sensor. In addition, there are commands such as the previously used info or status command, which must be intercepted by the MRPProxy module so that it can be handled differently (see example 5.5).

To realize this, a LUT was created in the previous steps, which contains information regarding requested capability -> sensor-UUID -> physical sensor and allows the commands to be routed.

For commands where there are several entries for CAPABLE SENSORS ID LUT5.1, there are two possible approaches to how the command is processed:

- Redirect to each capable sensor
- Extend commands using an id parameter

These two methods have been implemented and are applied automatically. The decision is based on which hardware sensors are connected. In a setup where only the same sensor variants are connected, redirect to each capable sensor is applied. This offers a time advantage as fewer commands need to be sent from the host. Thus, with a readsensor command, all sensors are read out via one command and the summarized result is transmitted to the host.

The extend commands using an id parameter strategy is used for different sensors. Each command is extended on the Network–Proxy5.3.1 by another UUID parameter, according to the following scheme:

- readsensor <axis> <sensor number>-> readsensor <axis> <ID>
- opmode-> opmode <ID>

This allows the host to address individual sensors directly via their specific UUID.

5.4 Examples

The following shows some examples of how the MRP-library can be used. These examples are limited to a functional minimum for selected modules of the MRP-library. The documentation 6.4.1 contains further and more detailed examples. Many basic examples are also supplied in the form of the test scripts used for testing 6.3.

5.4.1 MRPReading

The MRPReading is the key module of the MRP core. It is used to manage the measurement data and can be imported and exported. The following example 5.6 shows how a measurement is created and measurement points are added in the form of MRPReadingEntry instances. An important point is the management of the meta data, which further describes the measurement. This is realised in the example using the set_additional_data function.

```
from MRP import MRPReading, MRPMeasurementConfig
# [OPTIONAL] CONFIGURE READING USING MEASUREMENT CONFIG INSTANCE
config: MRPMeasurementConfig = MRPMeasurementConfig
config.sensor_distance_radius(40) # 40mm DISTANCE BETWEEN MAGNET AND SENSOR
config.magnet_type(N45_CUBIC_12x12x12) # CHECK MRPMagnetTypes.py FOR IMPLEMENTED TYPES
# CREATE READING
reading: MRPReading = MRPReading(config)
# ADD METADATA
reading.set_name("example reading")
## ADD FURTHER DETAILS
reading.set_additional_data("description", "abc")
reading.set_additional_data("test-number", 1)
# INSERT A DATAPOINT
measurement = MRPReadingEntry.MRPReadingEntry()
measurement.value = random.random()
reading.insert_reading_instance(measurement, False)
# USE MEASURED VALUES IN OTHER FRAMEWORKS / DATAFORMATS
## NUMPY
npmatrix: np.ndarray = reading.to_numpy_matrix()
## CSV
csv: []= reading.to_value_array()
## JSON
js: dict= reading.dump()
# EXPORT READING TO FILE
```

```
reading.dump_to_file("exported_reading.mag.json")
# IMPORT READING
imported_reading: MRPReading = MRPReading()
imported_reading.load_from_file("exported_reading.mag.json")
```

Listing 5.6: MRPReading example for setting up an basic measurement

Finally, the measurement is exported for archiving and further processing; various export formats are available. Using the dump_to_file function, the measurement can be converted into an open JSON format. This file can then be imported again using the load_from_file function.

5.4.2 MRPHal

After generating simple measurements with random values in the previous example 5.4.1, the next step is to record real sensor data. For this purpose, the MRPHal module was developed, which can interact with all Unified Sensor4-compatible sensors. In the following example 5.7, an 1D: Single Sensor4.5.1 is connected locally to the host PC.

```
from MRP import MRPHalSerialPortInformation, MRPHal, MRPBaseSensor,
    MRPReadingSource
# SEARCH FOR CONNECTED SENSORS
## LISTS LOCAL CONNECTED OR NETWORK SENSORS
system_ports = MRPHalSerialPortInformation.list_sensors()
sensor = MRPHal(system_ports[0])
# OR USE SPECIFIED SENSOR DEVICE
device_path = MRPHalSerialPortInformation("UNFSensor1")
sensor = MRPHal(device_path)
# RAW SENSOR INTERACTION MODE
sensor.connect()
basesensor = MRPBaseSensor.MRPBaseSensor(sensor)
basesensor.query_readout()
print(basesensor.get_b()) # GET RAW MEASUREMENT
print(basesensor.get_b(1)) # GET RAW DATA FROM SENSOR WITH ID 1
# TO GENERATE A READING THE perform_measurement FUNCTION CAN BE USED
reading_source = MRPReadingSourceHelper.createReadingSourceInstance(
    sensor)
result_readings: [MRPReading] = reading_source.perform_measurement(
    _readings=1, _hwavg=1)
```

Listing 5.7: MRPHal example to use an connected hardware sensor to store readings inside of an measurement

In general, a sensor can be connected using its specific system path or the sensor-UUID via the MRPHalSerialPortInformation function. Locally connected or network sensors can also be automatically recognised using the list_sensors function. Once connected, these are then converted into a usable data source using the MRPReadingSource module. This automatically recognises the type of sensor and generated an MRPReading instance with the measured values of the sensor.

5.4.3 MRPSimulation

If no hardware sensor is available or for the generation of test data, the MRPSimulation module is available. This contains a series of functions that simulate various magnets and their fields. The result is a complete MRPReading measurement with a wide range of set meta data. The example 5.8 illustrated the basic usage. Different variations of the generate_reading function offers the user additional parameterisation options, such as random polarisation direction or a defined centre-of-gravity vector. The data is generated in the background using the magpylib[16] library according to the specified parameters.

```
1 from MRP import MRPSimulation, MRPPolarVisualization, MRPReading
2 # GENERATE SILUMATED READING USING A SIMULATED HALLSENSOR FROM magpy
      LIBRARY
3 reading = MRPSimulation.generate_reading(MagnetType.N45_CUBIC_12x12x12,
      _add_random_polarisation=True)
4 # GENERATE A FULLSPHERE MAP READING
5 reading_fullsphere = MRPSimulation.generate_random_full_sphere_reading
      ()
6 # RENDER READING TO FILE IN 3D
7 visu = MRPPolarVisualization(reading)
8 visu.plot3d(None)
9 visu.plot3d("simulated_reading.png")
10 # EXPORT READING
11 reading.dump_to_file("simulated_reading.mag.json")
```

Listing 5.8: MRPSimulation example illustrates the usage of several data analysis functions

5.4.4 MRPAnalysis

Once data can be acquired using hardware or software sensors, the next step is to analyse this data. MRP provides some simple analysis functions for this purpose. The

code example shows the basic use of the module. The **Evaluation**7 chapter shows how the user can implement their own algorithms and add them to the library.

```
from MRP import MRPAnalysis, MRPReading
# CREATE A SAMPLE MEASUREMENT WITH SIMULATED DATA
reading = MRPSimulation.generate_reading(MagnetType.N45_CUBIC_10x10x10)
# CALCULATE MEAN
print(MRPAnalysis.calculate_mean(reading))
# CALCULATE STD DEVIATION ON TEMPERATURE AXIS
print(MRPAnalysis.calculate_std_deviation(reading, _temperature_axis=
    True))
# CALCULATE CENTER OF GRAVITY
(x, y, z) = MRPAnalysis.calculate_center_of_gravity(reading)
# APPLY CALIBRATION READING TO REMOVE BACKGROUND NOISE
calibration_reading = MRPSimulation.generate_reading(MagnetType.
    N45_CUBIC_10x10x10, _ideal = True)
MRPAnalysis.apply_calibration_data_inplace(calibration_reading, reading
    )
```

Listing 5.9: MRPAnalysis example code performs several data analysis steps

5.4.5 MRPVisualisation

This final example shows the use of the MRP Visualisation module, which provides general functions for visualising measurements. This makes it possible to visually assess the results of a measurement. This is particularly helpful for full-sphere measurements recorded with the 3D: Fullsphere4.5.2 sensor. The sub-module MRPPolarVisualisation is specially designed for these. The figure **5**-4 shows a plot of a fullsphere measurement.

It is also possible to export the data from the MRPAnalysis module graphically as diagrams. The MRPVisualisation modules are used here. The following example 5.10 shows the usage of both modules.

```
from MRP import MRPPolarVisualization
# CREATE MRPPolarVisualization INSTANCE
## IT CAN BE REUSED CALLING plot2d AGAIN, AFTER LINKED READING DATA
    WERE MODIFIED
visu = MRPPolarVisualization.MRPPolarVisualization(reading)
# 2D PLOT INTO A WINDOW
visu.plot2d_top(None)
visu.plot2d_side(None)
# 3D PLOT TO FILE
visu.plot3d(os.path.join('./plot3d_3d.png'))
```

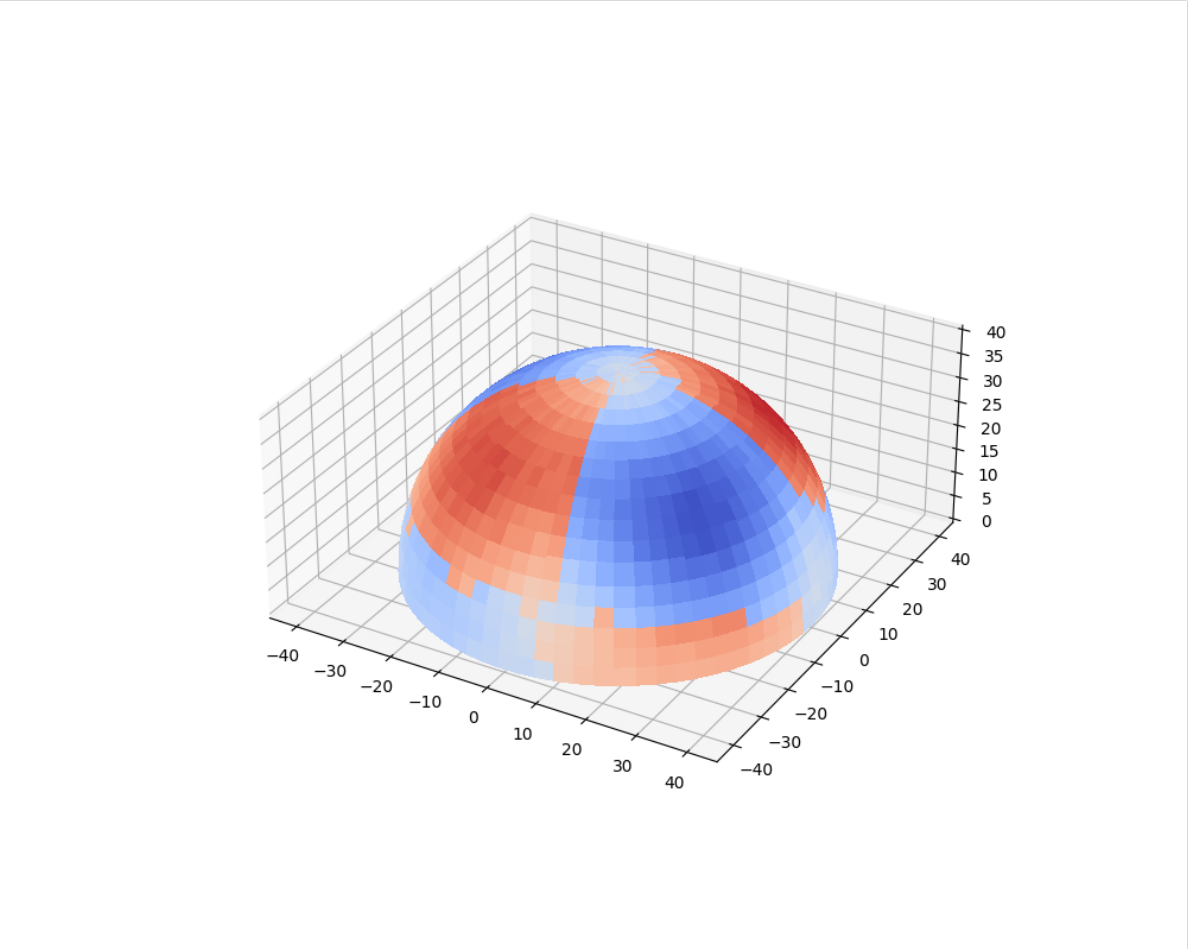

Figure **5**-4: Example full sphere plot of an measurement using the MRPVisualisation module

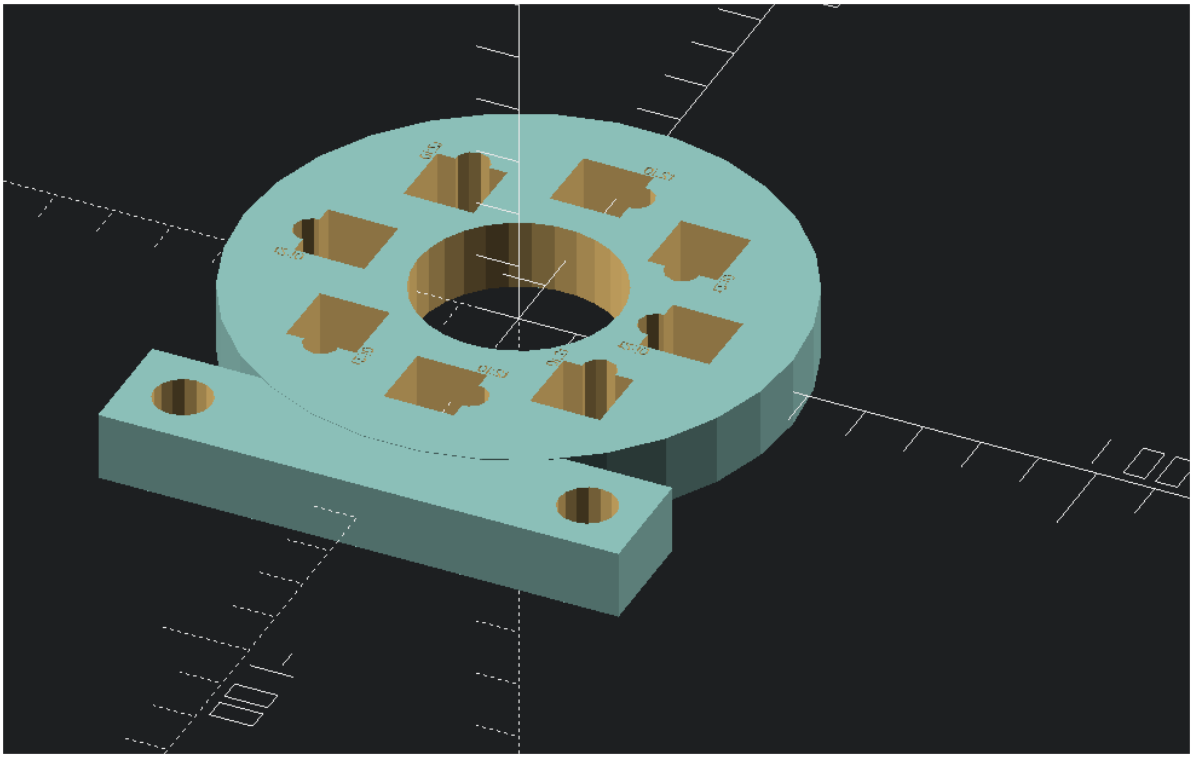

Figure **5**-5: Generated hallbach array with generated cutouts for eight magnets

```
10
11 # PLOT ANALYSIS RESULTS
12 from MRP import MRPDataVisualization
13 MRPDataVisualization.MRPDataVisualization.plot_error([reading_a,
       reading_b, reading_c])
```

Listing 5.10: MRPVisualisation example which plots a fullsphere to an image file

5.4.6 MRPHallbachArrayGenerator

The following example code 5.11, shows how a simple hallbach magnetic ring can be generated. Eight random measurements are generated here. It is important that the magnet type (here N45_CUBIC_15x15x15) is specified. This is necessary so that the correct magnet cutouts can be generated when creating the 3D model. After the measurements have been generated, they are provided with a position and rotation offset according to the Hallbach design and calucation scheme[17] using the MRPHallbachArrayGenerator module.

```
1 readings = []
2 for idx in range(8):
```

```
3   # GENERATE EXAMPLE READINGS USING N45 CUBIC 15x15x15 MAGNETS
4   readings.append(MRPSimulation.MRPSimulation.generate_reading(
      MRPMagnetTypes.MagnetType.N45_CUBIC_15x15))
5 ## GENERATE HALLBACH
6 hallbach_array: MRPHallbachArrayGenerator.MRPHallbachArrayResult =
      MRPHallbachArrayGenerator.MRPHallbachArrayGenerator.
      generate_1k_hallbach_using_polarisation_direction(readings)
7 # EXPORT TO OPENSCAD
8 ## 2D MODE DXF e.g. for lasercutting
9 MRPHallbachArrayGenerator.MRPHallbachArrayGenerator.
      generate_openscad_model([hallbach_array], "./2d_test.scad",
      _2d_object_code=True)
10 ## 3D MODE e.g. for 3D printing
11 MRPHallbachArrayGenerator.MRPHallbachArrayGenerator.
      generate_openscad_model([hallbach_array], "./3d_test.scad",
      _2d_object_code=False)
```

Listing 5.11: MRPHallbachArrayGenerator example for generating an OpenSCAD based hallbach ring

In the last step, a 3D model with the dimensions of the magnet type set is generated from the generated magnet positions. The result is an OpenSCAD[13] file, which contains the module generated. After computing the model using the OpenSCAD-CLI utility, the following model rendering **5**-5 can be generated.

6 Usability improvements

Usability improvements in software libraries are crucial for efficient and user-friendly development. Intuitive API documentation, clearly structured code examples and improved error messages promote a smooth developer experience. A Graphical User Interface (GUI) or CLI application for complex libraries can make it easier to use, especially for developers with less experience. Continuous feedback through automated tests and comprehensive error logs enable faster bug fixing. The integration of user feedback and regular updates promotes the adaptability of the MRP-library. Effective usability improvements help to speed up development processes and increase the satisfaction of the developer community. In the following, some of these have been added in and around the MRP-library, but they are only optional components for the intended use.

6.1 Command Line Interface

In the first version of the MRP-library, the user had to write his own Python scripts even for short measurement and visualisation tasks. However, this was already time-consuming for reading out a sensor and configuring the measurement parameters and metadata and quickly required more than 100 lines of new Python code. Although such examples are provided in the documentation, it must be possible for programming beginners in particular to use them. To simplify these tasks, a CLI **6**-2 was implemented.

```
CONFIGURE READING
READING-NAME: [read_dualsensor_normal]: >? new_measurement
OUTPUT-FOLDER [./readings/tlv493d_N45_12x12x12/]: >? ./
final output path for reading /Users/marcelochsendorf/Downloads/MagneticReadoutProcessing/src/MagneticReadoutProcessing
SUPPORTED MAGNET TYPES
0 > NOT_SPECIFIED
1 > RANDOM_MAGNET
2 > N45_CUBIC_12x12x12
3 > N45_CUBIC_15x15x15
4 > N45_CUBIC_9x9x9
5 > N45_CYLINDER_5x10
6 > N45_SPHERE_10
Please select one of the listed magnet types [0-6] [2]:
>?  3
```

Figure **6**-1: MRP CLI output to configure a new measurement

The libaray CLI implements the following functionalities:

- Detection of connected sensors
- Configuration of measurement series
- Recording of measured values from stored measurement series
- Simple commands for checking recorded measurement series and their data.

Thanks to this functionality of the CLI, it is now possible to connect a sensor to the PC, configure a measurement series with it and run it at the end. The result is then an exported file with the measured values. These can then be read in again using the MRPReading module and processed further. The following bash code 6.1 shows the setup procedure in detail:

```
# CLI EXAMPLE FOR CONFIGURING A MEASUREMENT RUN
## CONFIGURE THE SENSOR TO USE
$ MRPcli config setupsensor testcfg
> 0 - Unified Sensor 386731533439 - /dev/cu.usbmodem3867315334391
> Please select one of the found sensors [0]:
> sensor connected: True 1243455
## CONFIGURE THE MEASUREMENT
$ MRPcli config setup testcfg
> CONFIGURE testcfg
> READING-NAME: [testreading]: testreading
> OUTPUT-FOLDER [/cli/reading]: /tmp/reading_folder_path
> NUMBER DATAPOINTS: [1]: 10
> NUMBER AVERAGE READINGS PER DATAPOINT: [1]: 100
# RUN THE CONFIGURED MEASUREMENT
$ MRPcli measure run
> STARTING MEASUREMENT RUN WITH FOLLOWING CONFIGS: ['testcfg']
> config-test: OK
> sensor-connection-test: OK
> START MEASUREMENT CYCLE
> sampling 10 datapoints with 100 average readings
> SID:0 DP:0 B:47.359mT TEMP:23.56
> ....
> dump_to_file testreading_ID:525771256544952_SID:0_MAG:
    N45_CUBIC_12x12x12.mag.json
```

Listing 6.1: CLI example for configuring a measurement run

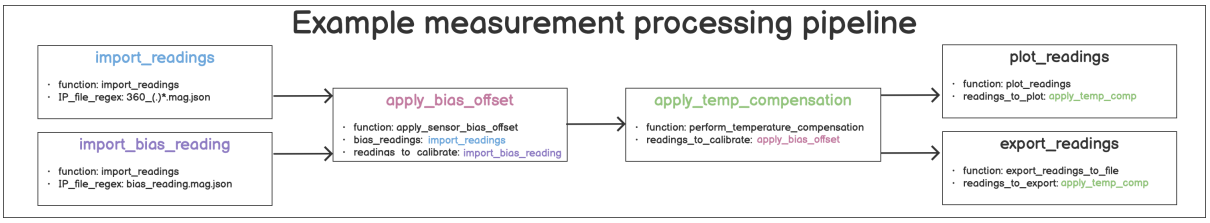

Figure **6**-2: example measurement analysis pipeline

6.2 Programmable data processing pipeline

After it is easy for users to carry out measurements using the CLI, the next logical step is to analyse the recorded data. This can involve one or several hundred data records. Again, the procedure for the user is to write their own evaluation scripts using the MRP-library. This is particularly useful for complex analyses or custom algorithms, but not necessarily for simple standard tasks such as bias compensation or graphical plot outputs.

For this purpose, a further CLI application was created, which enables the user to create and execute complex evaluation pipelines for measurement data without programming. The example **6**-2 shows a typical measurement data analysis pipeline, which consists of the following steps:

- Import the measurements
- Determine sensor bias value from imported measurements using a reference measurement
- Apply linear temperature compensation
- Export the modified measurements
- Create a graphical plot of all measurements with standard deviation

In order to implement such a pipeline, the yaml file format was chosen for the definition of the pipeline, as this is for non programmers to understand and can also be easily edited with a plain text editor. Detailed examples can be found in the documentation[3]. The pipeline definition consists of sections which execute the appropriate Python commands in the background. The signatures in the yaml file are called using reflection and a real-time search of the loaded global() symbol table[7]. This system makes almost all Python functions available to the user. To simplify use, a pre-defined list of verified functions for use in pipelines is listed in the documentation[3]. The following pipeline definition 6.2 shows the previously defined steps **6**-2 as yaml syntax.

```yaml
stage import_readings:
  function: import_readings
  parameters:
    IP_input_folder: ./readings/fullsphere/
    IP_file_regex: 360_(.)*.mag.json

stage import_bias_reading:
  function: import_readings
  parameters:
    IP_input_folder: ./readings/fullsphere/
    IP_file_regex: bias_reading.mag.json

stage apply_bias_offset:
  function: apply_sensor_bias_offset
  parameters:
    bias_readings: stage import_bias_reading # USE RESULT FROM FUNCTION
     import_bias_reading
    readings_to_calibrate: stage import_readings

stage apply_temp_compensation:
  function: apply_temperature_compensation
  parameters:
    readings_to_calibrate: stage import_readings # USE RESULT FROM
     FUNCTION import_readings

stage plot_normal_bias_offset:
  function: plot_readings
  parameters:
    readings_to_plot: stage apply_temp_compensation
    IP_export_folder: ./readings/fullsphere/plots/
    IP_plot_headline_prefix:  Sample N45 12x12x12 magnets calibrated

stage export_readings:
  function: export_readings
  parameters:
    readings_to_plot: stage apply_temp_compensation
    IP_export_folder: ./readings/fullsphere/plots/
```

Listing 6.2: Example YAML code of an user defined processing pipeline with six stages linked together

Each pipeline step is divided into stages, which contain a name, the function to be executed and its parameters. The various steps are then linked by using the stage <name > makro as input parameter of the next function to be executed (see comments in 6.2). It is therefore also possible to use the results of one function in several others without them directly following each other. The disadvantages of this system are the following:

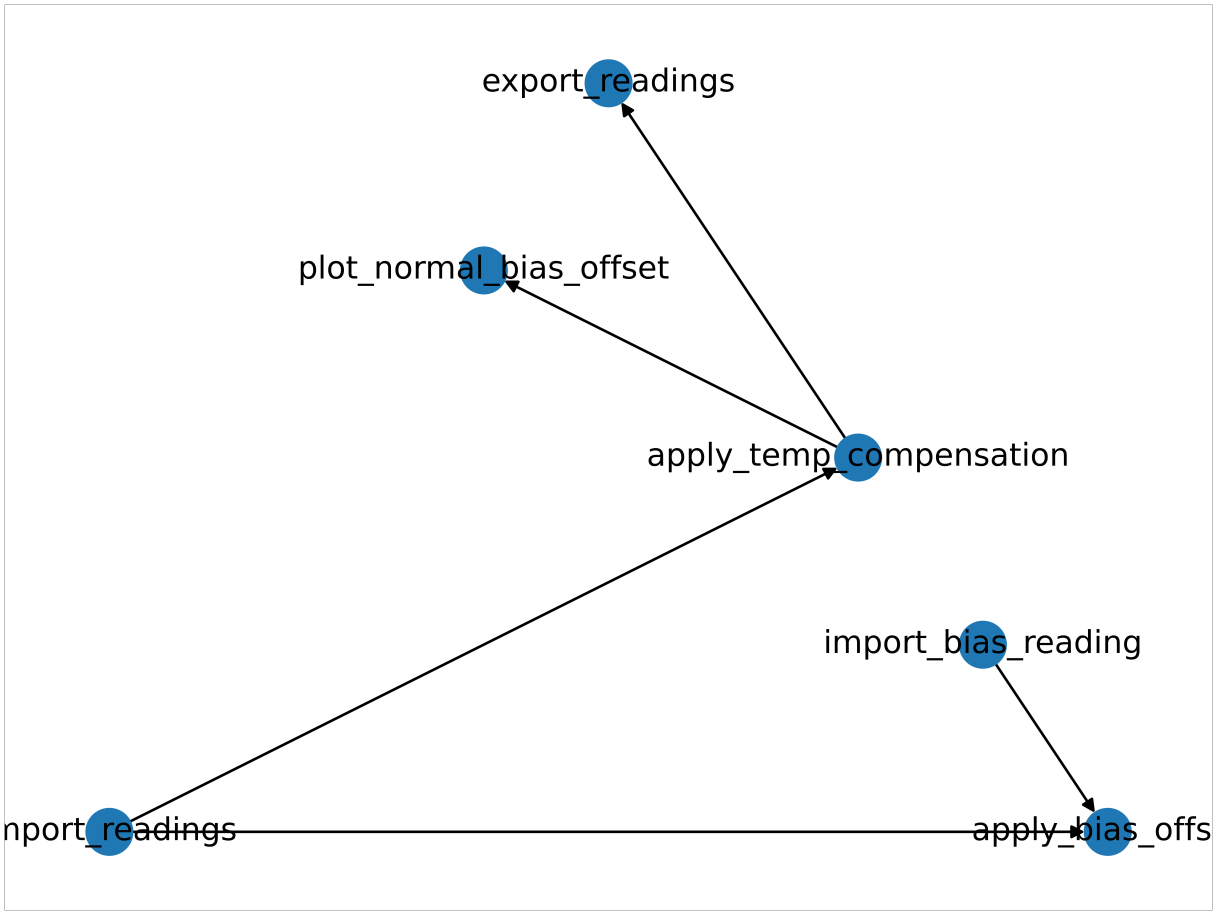

Figure **6**-3: Result step execution tree from user defined processing pipeline example

- No circular parameter dependencies
- Complex determination of the execution sequence of the steps

To determine the order of the pipeline steps, the parser script created converts them into one problem of the graph theories. Each step represents a node in the graph and the steps referred to by the input parameter form the edges.

After several simplification steps, determination of possible start steps and repeated traversal, the final execution sequence can be determined in the form of a call tree **6**-3. The individual steps are then executed along the graph. The intermediate results and the final results **6**-4 are saved for optional later use.

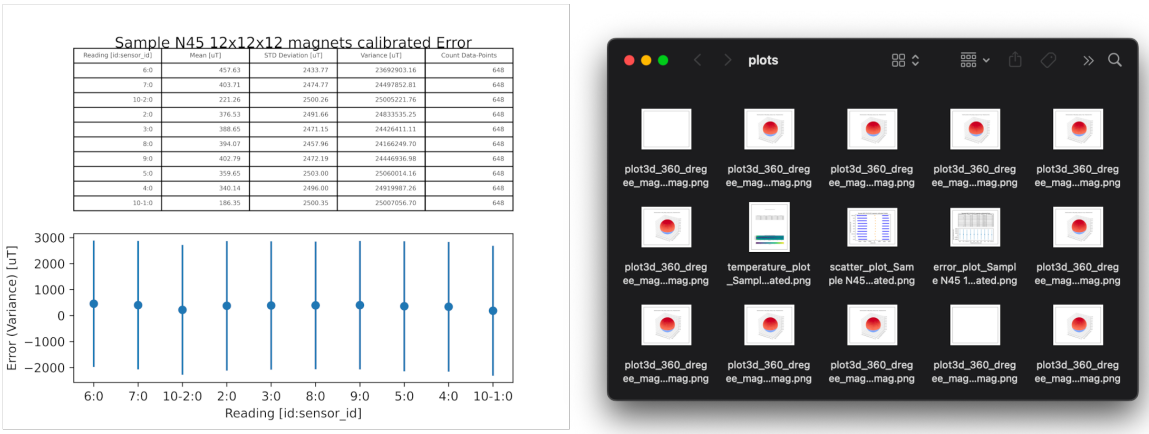

Figure **6**-4: pipeline output files after running example pipeline on a set of readings

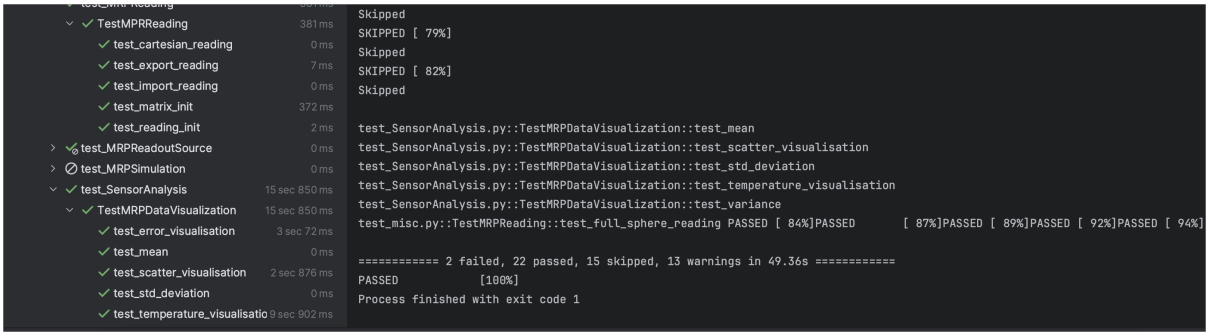

Figure **6**-5: MRP library test results for different submodules executed in PyCharm IDE

6.3 Testing

Software tests in libraries offer numerous advantages for improving quality and efficiency. Firstly, they enable the identification of errors and vulnerabilities before software is published as a new version. This significantly improves the reliability of MRP-library applications. Tests also ensures consistent and reliable performance, which is particularly important when libraries are used by different users and for different usecases.

During the development of the MRP-library, test cases were also created for all important functionalities and usecases. The test framework PyTest[14] was used for this purpose, as it offers direct integration in most IDEs (see **6**-5) and also because it provides detailed and easy-to-understand test reports as output in order to quickly identify and correct errors. It also allows to tag tests, which is useful for grouping tests or excluding certain tests in certain build environment scenarios. Since all intended usecases were mapped using the test cases created, the code of the test cases could later be used in slightly simplified variants 6.3 as examples for the documentation.

```python
class TestMPRReading(unittest.TestCase):
  # PREPARE A INITIAL CONFIGURATION FILE FOR ALL FOLLOWING TEST CASES IN THIS FILE
  def setUp(self) -> None:
    self.test_folder: str = os.path.join(os.path.dirname(os.path.abspath(__file__)), "tmp")
    self.test_file:str = os.path.join(self.import_export_test_folderpath, "tmp")

  def test_matrix(self):
    reading: MRPReading = MRPSimulation.generate_reading()
    matrix: np.ndarray = reading.to_numpy_matrix()
    n_phi: float = reading.measurement_config.n_phi
    n_theta: float = reading.measurement_config.n_theta
    # CHECK MATRIX SHAPE
    self.assertTrue(matrix.shape != (n_theta,))
    self.assertTrue(len(matrix.shape) <= n_phi)

  def test_export_reading(self) -> None:
    reading: MRPReading = MRPSimulation.generate_reading()
    self.assertIsNotNone(reading)
    # EXPORT READING TO A FILE
    reading.dump_to_file(self.test_file)

  def test_import_reading(self):
    # CREATE EMPTY READING AND LOAD FROM FILE
    reading_imported:MRPReading = MRPReading.MRPReading(None)
    reading_imported.load_from_file(self.test_file)
    # COMPARE
    self.assertIsNotNone(reading_imported.compare(MRPSimulation.generate_reading()))
```

Listing 6.3: Example pytest class for testing MRPReading module functions

One problem, however, is the parts of the MRP-library that require direct access to external hardware. These are, for example, the MRPHal and MRPHalRest modules, which are required to read out sensors connected via the network. Two different approaches were used here. In the case of local development, the test runs were carried out on a PC that can reach the network hardware and thus the test run could be carried out with real data.

In the other scenario, the tests are to be carried out before a new release in the repository on the basis of Github Actions[11]. Here there is the possibility to host local runner software, which then has access to the hardware, but then a PC must be permanently available for this task. Instead, the hardware sensors were simulated by

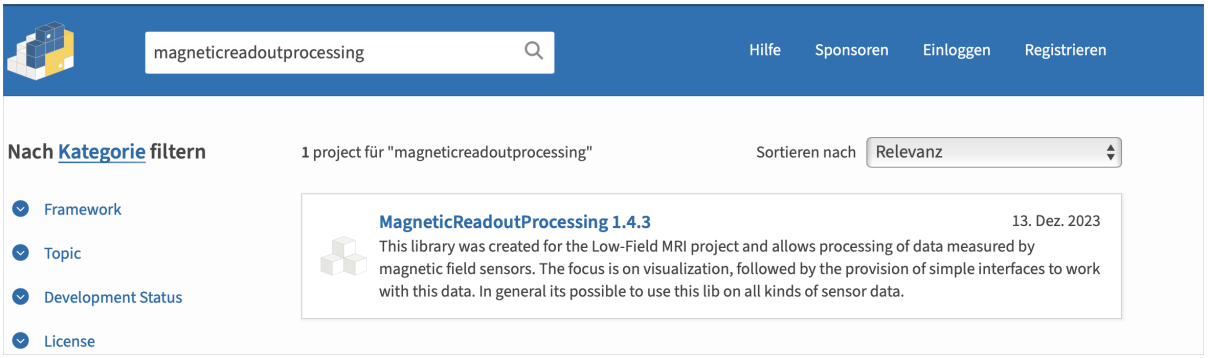

Figure **6**-6: MagneticReadoutProcessing library hosted on PyPi

software and executed via virtualisation on the systems provided by Github Actions[11].

6.4 Package distribution

One important point that improves usability for users is the simple installation of all MRP modules. As it was created in the Python programming language, there are several public package registry where users can provide their software modules. Here, PyPi[6] **6**-6[5] is the most commonly used package registry and offers direct support for the package installation programm Python Package Installer (PIP) 6.4.

In doing so, PIP not only manages possible package dependencies, but also manages the installation of different versions of a package. In addition, the version compatibility is also checked during the installation of a new package, which can be resolved manually by the user in the event of conflicts.

```
1 # https://pypi.org/project/MagneticReadoutProcessing/
2 # install the latest version
3 $ pip3 install MagneticReadoutProcessing
4 # install the specific version 1.4.0
5 $ pip3 install MagneticReadoutProcessing==1.4.0
```

Listing 6.4: Bash commands to install the MagneticReadoutProcessing library using pip

To make the MRP file strucutre compatible with the package registry, Python provides separate installation routines that build a package in an isolated environment and then provide an installation wheel archive. This can then be uploaded to the package registry.

Since the MRP-library requires additional Python dependencies (e.g. numpy, matplotlib), which cannot be assumed to be already installed on the target system, these must be installed prior to the actual installation. These can be specified in the library installation configuration setup.py6.5 for this purpose.

```
# dynamic requirement loading using 'requirements.txt'
req_path = './requirements.txt'
with pathlib.Path(req_path).open() as requirements_txt:
  install_requires = [str(requirement) for requirement in pkg_resources
    .parse_requirements(requirements_txt)]

setup(name='MagneticReadoutProcessing',
  version='1.4.3',
  url='https://github.com/LFB-MRI/MagnetCharacterization/',
  packages= ['MRP', 'MRPcli', 'MRPudpp', 'MRPproxy'],
  install_requires=install_requires,
  entry_points={
    'console_scripts': [
      'MRPCli = MRPcli.cli:run',
      'MRPUdpp = MRPudpp.uddp:run',
      'MRPproxy = MRPproxy.mrpproxy:run'
    ]
  }
)
```

Listing 6.5: setup.py with dynamic requirement parsing used given requirements.txt

To make the CLI scripts written in Python easier for the user to execute without having to use the python3 prefix. This has been configured in the installation configuration using the entry_points option, and the following commands are available to the user:

- MRPcli --help instead of python3 cli.py --help
- MRPudpp --help instead of python3 udpp.py --help
- MRPproxy --help instead of python3 proxy.py --help

In addition, these commands are available globally in the system without the terminal shell being located in the MRP-library folder.

6.4.1 Documentation

In order to provide comprehensive documentation for the enduser, the source code was documented using Python-docstrings[9] and the Python3.5 type annotations:

- Function description
- Input parameters, using param and type
- Return value, using returns, rtype

The use of type annotations also simplifies further development, as modern IDEs can more reliably display possible methods to the user as an assistance. 6.6

```
1 # MRPDataVisualisation.py - example docstring
2 def plot_temperature(_readings: [MRPReading.MRPReading], _title: str =
      '', _filename: str = None, _unit: str = "degree C") -> str:
3   """
4   Plots a temperature bar graph of the reading data entries as figure
5   :param _readings: readings to plot
6   :type _readings: list(MRPReading.MRPReading)
7   :param _title: Title text of the figure, embedded into the head
8   :type _title: str
9   :param _filename: export graphic to an given absolute filepath with .
      png
10  :type _filename: str
11  :returns: returns the abs filepath of the generated file
12  :rtype: str
13  """
14  if _readings is None or len(_readings) <= 0:
15      raise MRPDataVisualizationException("no readings in _reading
      given")
16  num_readings = len(_readings)
17  # ...
```

Listing 6.6: Documentation using Python docstring example

Since docstrings only document the source code, but do not provide simple how-to-use instructions, the documentation framework Sphinx[2] was used for this purpose. This framework makes it possible to generate Hypertext Markup Language (HTML) or Portable Document Format (PDF) documentation from various source code documentation formats, such as the used docstrings. These are converted into a Markdown format in an intermediate step and this also allows to add further user documentation such as examples or installation instructions. In order to make the documentation created by Sphinx accessible to the user, there are, as with the package management by PyPi

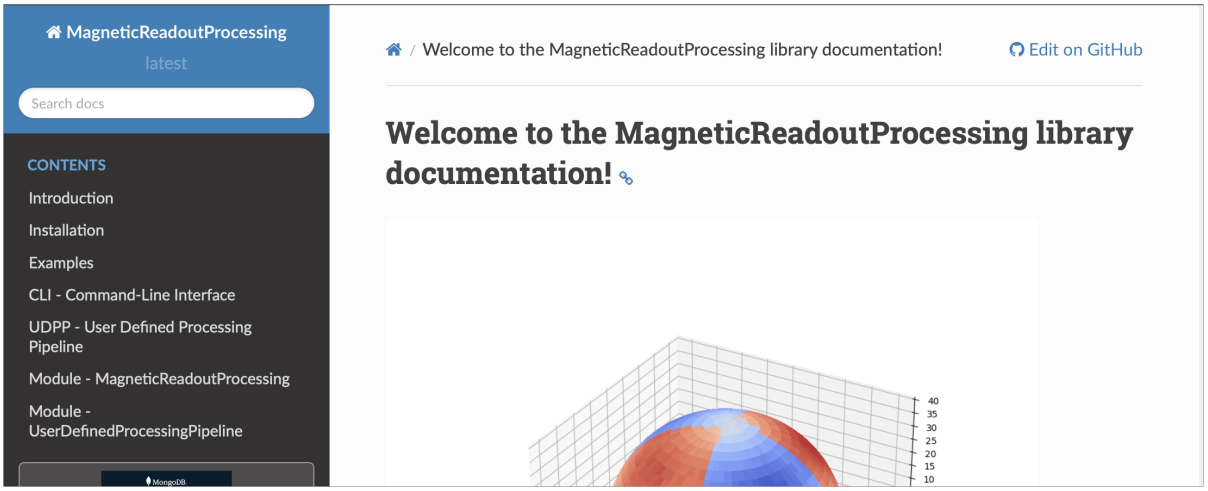

Figure **6**-7: MagneticReadoutProcessing documentation hosted on ReadTheDocs

services, which provide the MRP-library documentation online.

Once the finished documentation has been generated from static HTML files, it is stored in the project repository. Another publication option is to host the documentation via online services such as ReadTheDocs[4], where users can make documentation for typical software projects available to others.

The documentation has also been uploaded for ReadTheDocs[3] and linked in the repository and on the overview page **6**-7 on PyPi.

The process of creating and publishing the documentation has been automated using GitHub Actions[11], so that it is always automatically kept up to date with new features.

7 Evaluation

This work successfully implemented a universal hardware and software framework for the automated characterisation of permanent magnets. This framework consists of a low-cost hardware interface that supports various magnetic field sensors and a library for automating and analysing the measurement data. The process of this framework comprises several steps, which I will explain below:

1. **Hardware preperation** Users can prepare measurements using the implemented framework. This includes the placement of the sensors and the selection of the relevant parameters for the characterisation of the permanent magnets.

2. **Configuration of the measurement** The software provides a user-friendly interface for configuring the measurement parameters. Users can make the desired settings here and customise the framework to their specific requirements.

3. **Custom algorithm implementation** An important contribution of the MRP ecosystem is the possibility for users to implement their own algorithm for data analysis. This allows customisation to specific research questions or experimental requirements.

4. **Execution of analysis pipeline** The analysis pipeline can then be executed with the implemented algorithm. The collected measurement data is automatically processed and analysed to extract characteristic parameters of the permanent magnets.

This process covers all the essential functionalities required for a comprehensive characterisation of permanent magnets. These were previously described in the Usecases 3 chapter. The developed framework not only offers a cost-effective and flexible hardware solution, but also enables customisation of the analysis algorithms to meet the requirements of different research projects.

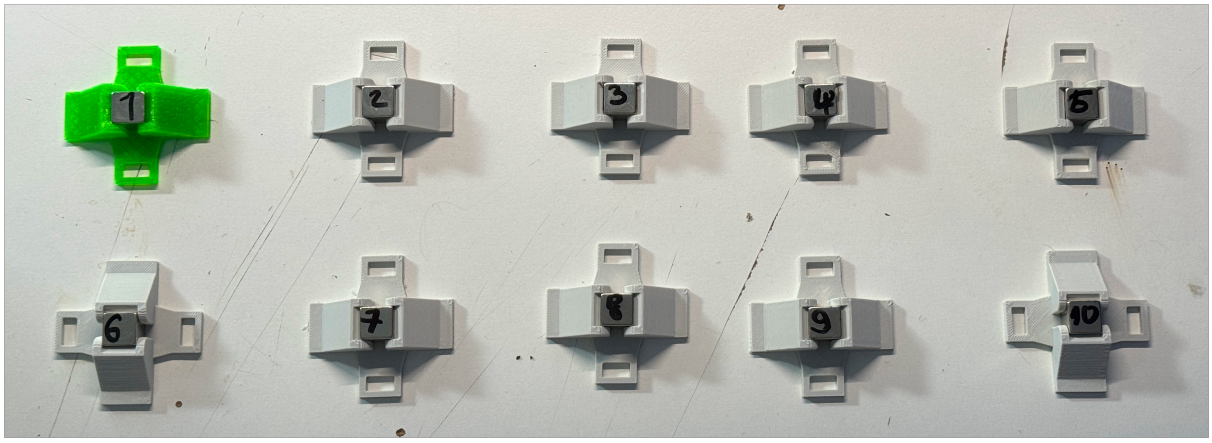

Figure 7-1: Ten numbered test magnets in separate holders

7.1 Hardware preperation

For the hardware setup, the 3D-Fullsphere4.5.2 sensor was used for the evaulation of the framework. As this is equipped with an exchangeable magnetic holder mount, suitable holders are required for the magnets to be measured. Ten random N45 12x12x12mm neodymium magnets were used here. These were placed in modified 3D printed holders 7-1 and then numbered. This allows them to be matched to the measurement results later.

7.2 Configuration of the measurement

The configured hardware was then connected and connected to the host system using the MRPcli config setupsensor-CLI command. The measurement was then configured for an measurement run, using the following configuration commands 7.1.

```
1 ## CONFIGURE THE MEASUREMENT
2 $ MRPcli config setup eval_measurement_config
3 > READING-NAME: 360_eval_magnet_<id>
4 > OUTPUT-FOLDER: ./readings/evaluation/
5 > NUMBER DATAPOINTS: 18 # FOR A FULLSPHERE READING USE MULTIBLE OF 18
6 > NUMBER AVERAGE READINGS PER DATAPOINT: 10
```

Listing 7.1: Measurement configuration for evaluation measurement

The MRPcli measure run command was then called up for each individual magnet to execute a measurement. After each run, the READING-NAME parameter was filled with

the id of the next magnet so that all measurements could be assigned to the physical magnets.

7.3 Custom algorithm implementation

The next step for the user is the implementation of the filter algorithm 7.2. This can have any function signature and is implemented in the file UDPPFunctionCollection.py. This Python file is loaded when the pipeline is started and all functions that are imported here as a module or implemented directly can be called via the pipeline. As this is a short algorithm, it was inserted directly into the file. The parameter _readings should later receive the imported measurements from the stage rawimport7.3 and the optional IP_return_count parameter specifies the number of best measurements that are returned. The return parameter is a list of measurements and should contain the measurements that are closest to the mean value of all measurements after the function has been executed.

```
@staticmethod
def FindSimilarValuesAlgorithm(_readings: [MRPReading.MRPReading],
    IP_return_count: int = -1) -> [MRPReading.MRPReading]:
  import heapq
  heap = []
  # SET RESULT VALUE COUNT
  IP_return_count = max([int(IP_return_count),len(_readings)])
  if IP_return_count < 0:
      IP_return_count = len(_readings) / 5
  # CALCULATE TARGET VALUE: MEAN FROM ALL VALUES
  target_value = 0.0
  for idx, r in enumerate(_readings):
      mean: float = MRPAnalysis.MRPAnalysis.calculate_mean(r)
      target_value = target_value + mean
  target_value = target_value / len(_readings)
  # PUSH READINGS TO HEAP
  for value in _readings:
      # USE DIFF AS PRIORITY VALUE IN MIN-HEAP
      diff = abs(value - target_value)
      heapq.heappush(heap, (diff, value))
  # RETURN X BEST ITEMS FROM HEAP
  similar_values = [item[1] for item in heapq.nsmallest(IP_return_count
    , heap)]
  # CLEAN UP USED LIBRARIES AND RETURN RESULT
  del heapq
  return similar_values
```

Listing 7.2: User implemented custom find most similar readings algorithm

The python heapq[8] module, which implements a priority queue, is used for this purpose. The calculated distances from the mean value of the measurements to the global mean value are inserted into this queue. Subsequently, as many elements of the queue are returned as defined by the IP_return_count parameter. The actual sorting was carried out by the queue in the background.

7.4 Execution of analysis pipeline

Once the filter function has been implemented, it still needs to be integrated into the analysis pipeline7.3. Here, the example pipeline **6**-2 is simplified and an additional stage find_similar_values has been added, which has set FindSimilarValuesAlgorithm as the function to be called. As a final step, the result is used in the plot_filtered stage for visualisation.

```
settings:
  enabled: true
  export_intermediate_results: false
  name: pipeline_mrp_evaluation

stage rawimport:
  function: import_readings
  parameters:
    IP_input_folder: ./readings/evaluation/
    IP_file_regex: 360_(.)*.mag.json

stage find_similar_values:
  function: custom_find_similar_values_algorithm
  parameters:
    _readings: stage rawimport # USE RESULTS FROM rawimport STAGE
    IP_return_count: 4 # RETURN BEST 4 of 10 READINGS

stage plot_filtered:
  function: plot_readings
  parameters:
    readings_to_plot: stage find_similar_values # USE RESULTS FROM
    find_similar_values STAGE
    IP_export_folder: ./readings/evaluation/plots/plot_filtered/
    IP_plot_headline_prefix: MRP evaluation - filtered
```

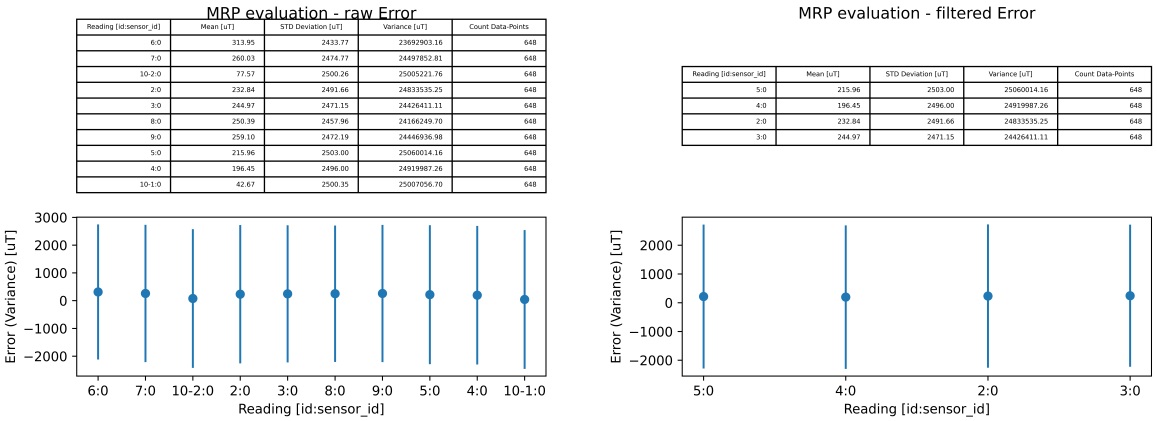

Figure **7**-2: MRP evaluation result

Listing 7.3: User defined processing pipeline using custom implemented filter algorithm

The final pipeline has been saved in the pipeline directory as pipeline_mrp_evaluation.yaml file and is ready for execution. This is carried out using the MRPudpp-(cli) 7.4. After the run has been successfully completed, the results are saved in the result folder specified in the pipeline using the IP_export_folder parameter.

```
1 # LIST ACTIVE PIPELINES IN PIPELINE DIRECTORY
2 $ MRPudpp pipeline listenabledpipelines
3 > Found enabled pipelines:
4 > 1. pipeline_mrp_evaluation.yaml
5 # EXECUTE THE EVALUATION PIPELINE
6 $ MRPudpp pipeline run
7 > loading pipeline pipeline_mrp_evaluation.yaml
8 > stage nodes: ['import', 'find_similar_values', 'plot_raw', '
    plot_filtered']
9 > =====> stage: import
10 > =====> stage: find_similar_values
11 > =====> stage plot_filtered
12 > Process finished with exit code 0
```

Listing 7.4: Bash result log of evaluation pipeline run

The figure **7**-2 shows this result. The plot of the raw measured values is shown on the left. The variance of the determined Mean [uT] mean values is plotted on ten individual measured values. Here you can see that there are measured values with larger deviations (see measurement 7:0,10–2:0,10–1:0).

On the right-hand side **7**-2, the measured values are plotted as a result of the filter algorithm. As the IP_return_count parameter was set to four, only the four most similar measurements were exported here. It can be seen from the plotted mean values Mean [uT] that these are closest to the global mean value 208.8uT. The filter algorithm implemented by the user was thus successfully executed using the user-programmable pipeline. The calculation result was successfully verified using raw measurement data and the final result of the algorithm.

8 Conclusion and dicussion

8.1 Conclusion

This work describes the development of a universal Python library that is used to efficiently process data from magnetic field sensors from acquisition to analysis. In order to ensure a practical application and to give users the opportunity to directly acquire their own magnetic field data, cost-effective and easily reproducible hardware was also developed.

The hardware is based on widely used magnetic field sensors and low-cost microcontrollers, which enables an easily expandable and applicable solution for measuring magnets with repeatable accuracy. A particular focus was placed on expandability by the user. Interchangeable modules allow the user to develop their own analysis algorithms without having to redesign everything from scratch.

This extensibility and customisability was successfully demonstrated during the evaluation. This underlines the performance of the developed framework and shows that it is not only effective in the processing of magnetic field sensor data, but also offers a flexible platform for the implementation of user-specific analyses.

8.2 Outlook

A solid foundation has been built in this version of the framework, which contains all the necessary functions and is ready for immediate use. During development, particular emphasis was placed on comprehensive documentation to make it easier to get started. Together with examples for various usecases, a user can quickly evaluate the framework.

However, it should be noted that the framework has already been released with its first stable version, but extensions and improvements are still necessary. The stable version distributed via the package registry is well suited for the intended purpose. All tests and evaluations took place under normal conditions, especially for the developed hardware sensors, as the MRP library works successfully with the measurement data.

On the software side, the focus is on integration for the support of more professional measuring devices. Only in this way is it possible to evaluate and improve the sensor hardware and quantify the measurement results.

To summarise, it can be said that a solid software framework has been created that can be used directly for the intended purpose. It provides a good foundation, but can be further developed by integrating professional measurement devices to enable a more comprehensive evaluation and improvement of the sensor hardware.

Bibliography

[1] BENJAMIN MENKÜC, Thomas O. Lukas Winter W. Lukas Winter: *HalbachMRI-Designer - Create a Halbach MRI Magnet from parametric definitions*. https://github.com/menkueclab/HalbachMRIDesigner. Version: 01.01.2024

[2] DEVELOPERS, Sphinx: *Sphinx makes it easy to create intelligent and beautiful documentation.* https://www.sphinx-doc.org/en/master/. Version: 01.01.2024

[3] DOCS, Inc Read t.: *MagneticReadoutProcessing - ReadTheDocs*. https://magneticreadoutprocessing.readthedocs.io/en/latest/index.html. Version: 01.01.2024

[4] DOCS, Inc Read t.: *ReadTheDocs - Build, host, and share documentation, all with a single platform.* https://readthedocs.com. Version: 01.01.2024

[5] FOUNDATION, Python S.: *MagneticReadoutProcessing - PyPi*. https://pypi.org/project/MagneticReadoutProcessing/. Version: 01.01.2024

[6] FOUNDATION, Python S.: *PyPi - The Python Package Index (PyPI) is a software directory for the Python programming language.* https://pypi.org. Version: 01.01.2024

[7] FOUNDATION, Python S.: *Python Docs - globals*. https://docs.python.org/3/library/functions.html#globals. Version: 01.01.2024

[8] FOUNDATION, Python S.: *Python Docs - heapq*. https://docs.python.org/3/library/heapq.html. Version: 01.01.2024

[9] FOUNDATION, Python S.: *Python Enhancement Proposals - Docstring Conventions*. https://peps.python.org/pep-0257/. Version: 01.01.2024

[10] GEERLING, Jeff: *PTP and IEEE-1588 hardware timestamping*. https://www.jeffgeerling.com/blog/2022/ptp-and-ieee-1588-hardware-timestamping-on-raspberry-pi-cm4. Version: 01.01.2024

[11] GITHUB, Inc.: *Github Actions - Automate your workflow from idea to production*. https://github.com/features/actions. Version: 01.01.2024

[12] HARRIS, Charles R. ; MILLMAN, K. J. ; WALT, Stéfan J. ; GOMMERS, Ralf ; VIRTANEN, Pauli ; COURNAPEAU, David ; WIESER, Eric ; TAYLOR, Julian ; BERG, Sebastian ; SMITH, Nathaniel J. ; KERN, Robert ; PICUS, Matti ; HOYER, Stephan ; KERKWIJK, Marten H. ; BRETT, Matthew ; HALDANE, Allan ; RÍO, Jaime F. ; WIEBE, Mark ; PETERSON, Pearu ; GÉRARD-MARCHANT, Pierre ; SHEPPARD, Kevin ; REDDY, Tyler ; WECKESSER, Warren ; ABBASI, Hameer ; GOHLKE, Christoph ; OLIPHANT, Travis E.: Array programming with NumPy. In: *Nature* 585 (2020), September, Nr. 7825, 357–362. http://dx.doi.org/10.1038/s41586-020-2649-2. – DOI 10.1038/s41586–020–2649–2

[13] KINTEL, Marius: *OpenSCAD - The Programmers Solid 3D CAD Modeller*. https://openscad.org. Version: 01.01.2024

[14] KREKEL, Holger ; TEAM pytest-dev: *pytest: helps you write better programs*. https://docs.pytest.org/en/7.4.x/. Version: 01.01.2024

[15] OCHSENDORF, Marcel: *VoltcraftGM70 REST Interface*. https://github.com/RBEGamer/VoltcraftGM70Rest. Version: 01.01.2024

[16] ORTNER, Michael ; BANDEIRA, Lucas Gabriel C.: Magpylib: A free Python package for magnetic field computation. In: *SoftwareX* 11 (2020), S. 100466

[17] PETER BLUEMLER, Helmut S.: Practical Concepts for Design, Construction and Application of Halbach Magnets in Magnetic Resonance. In: *Applied Magnetic Resonance* (2023), 05, S. 2522. http://dx.doi.org/10.1007/s00723-023-01602-2. – DOI 10.1007/s00723–023–01602–2

[18] SOLTNER, H. ; BLÜMLER, P.: Dipolar Halbach magnet stacks made from identically shaped permanent magnets for magnetic resonance. In: *Concepts in Magnetic Resonance Part A* 36A (2010), Nr. 4, 211-222. http://dx.doi.org/https://doi.org/10.1002/cmr.a.20165. – DOI https://doi.org/10.1002/cmr.a.20165

[19] WICKENBROCK, Arne ; ZHENG, Huijie ; CHATZIDROSOS, Georgios ; SHAJI REBEIRRO, Joseph ; SCHNEEMANN, Tim ; BLÜMLER, Peter: High homogeneity permanent magnet for diamond magnetometry. In: *Journal of Magnetic Resonance* 322 (2021), Januar, 106867. http://dx.doi.org/10.1016/j.jmr.2020.106867. – DOI 10.1016/j.jmr.2020.106867. – ISSN 1090–7807

List of Figures

List of Tables

Listings