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 Any passages taken verbatim or in spirit from published or unpublished sources are labelled as such.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

In the construction of low-field MRI devices based on permanent magnets, a large number of magnets are used. In order to realize a homogeneous B0 field with these magnets, which is necessary for many setups, the magnetic properties of these magnets have to be as similar to a certain degree. Due to the complex manufacturing process of neodymium magnets, the different properties, the direction of magnetization, can deviate from each other, which affects the homogeneity of the field. To adjust the field afterwards, a passive shimming process is typically performed, which is complex and time-consuming and requires manual corrections to the magnets used. To avoid this process, magnets can be systematically measured in advance. In this methodology, the recording, data storage and subsequent evaluation of the data play an important role. Various existing open-source solutions implement individual parts, but do not provide a complete data processing pipeline from aqusation to analysis and the data storage formats of these are not compatible to each other. For this use case, the MagneticReadoutProcessing library was created, which implements all major aspects of acquisition, storage, analysis, and each intermediate step can be customized by the user without having to create everything from scratch, favoring an exchange between different user groups. Complete documentation, tutorials and tests enable users to use and adapt the Framework as quickly as possible. The framework for the characterization of different magnets, which requires the integration of magnetic field sensors, was used for the evaluation.

List of abbreviations

CAD Computer Aided Design. 7

CDC Communication Device Class. 10

CLI Command Line Interface. 10, 11, 15, 30–33, 39, 43

GPIO General Purpose Input/Output. 8, 18

GUI Graphical User Interface. 31

HAL Hardware Abstraction Layer. 24

HTML Hypertext Markup Language. 40, 41

I2C Inter-Integrated Circuit. 6, 10

IC Integrated Circuit. 6

IDE Integrated development environment. 36, 40

IP Internet Protocol. 25

LUT Lookup Table. 10

MRP MagneticReadoutProcessing. 5, 11, 13, 15, 16, 18, 22, 26, 31–33, 36–39, 41, 42, 49

PC Personal Computer. 8, 10, 12, 18, 19, 22–25, 32, 37

PDF Portable Document Format. 40

PIP Python Package Installer. 38

PPS Puls Per Second. 25

PTP Precision Time Protocol. 25

REST Representational State Transfer. 23, 24

SBC Single Board Computer. 13, 18, 23, 25

UART Universal Asynchronous Receiver / Transmitter. 10, 18

USB Universal Serial Bus. 10, 12, 13, 25

Contents

1 Introduction

1.1 Background and Motivation

1.1.1 Low-Field MRI

1.1.2 Shimming procedure

1.2 Aim of this Thesis

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.

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.

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.

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.

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

6. **Usability improvements** refers to measures to improve user-friendliness. This involves optimising interfaces, interactions and processes to ensure intuitive and efficient use of the product. This includes, code documentation or the distribution of source code to users.

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 use cases.

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

3 Usecases

1.

4 Unified Sensor

The main aim of this project is to develop a low-cost Hall sensor 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 up possibilities for different applications. A low-cost Hall sensor interface will therefore not only be economically attractive, but also facilitate the integration of Hall sensors in different contexts. In addition, the low-cost Hall 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.

Linear analogue Hall sensors were deliberately omitted, although they are suitable for more precise measurements and extended measuring ranges. They were excluded due to the more complex circuit layouts required and more complex power 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.

Tabelle 4.1: Implemented digital halleffect 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	Inter-Integrated Circuit (I2C)	I2C	I2C	I2C

Focussing on digital interfaces 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.

The table4.1 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 structure of a sensor is kept very simple for the following tests. The focus was on providing a stable foundation for the sensor Integrated Circuit (IC) and an exchangeable holder for different magnets.

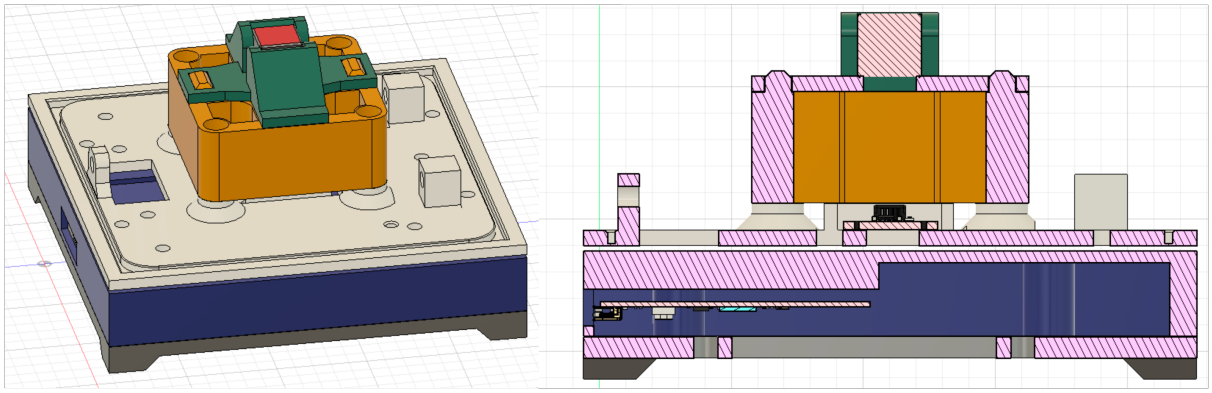

Figure **4**-1: 1D sensor mechanical 3D printed structure

The following figure**4**-1, shows a sectional view of the Computer Aided Design (CAD) drawing of the 1D: Single sensor4.5.1.

All parts were produced using the 3D printing process. 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 material used. The two recesses lock the holder with the inserted magnet over the sensor. Due to predetermined tolerances, the magnet can be inserted into the holder with repeat accuracy and without play. 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

As with the mechanical design, the electronics should be kept as simple as possible so that it is possible to use the microcontroller and a magnetic field sensor.

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

A decisive step in my work was the in-house PCB design**4**-2, which leads out all the

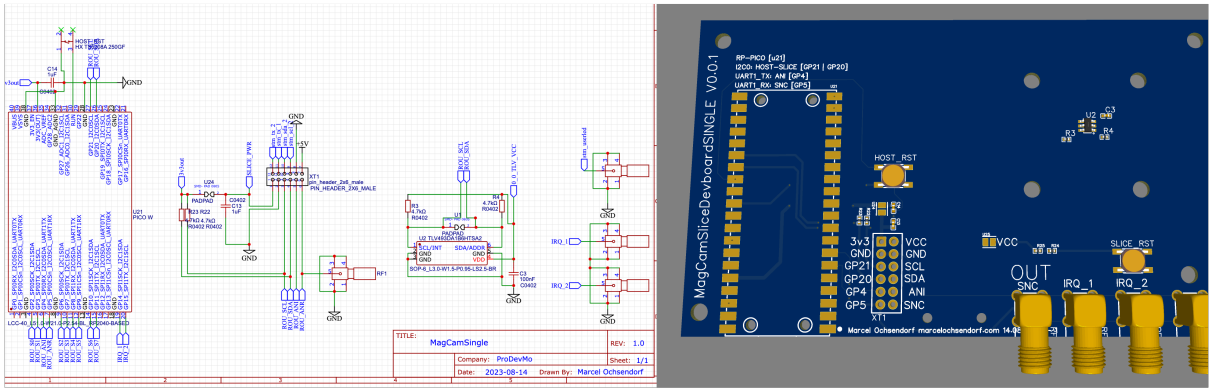

Figure **4**-2: 1D sensor schematic and circuit board

important connections. The placement and alignment of components optimised the wiring, which in turn improved the reliability of the sensor tests. Particular attention was paid to the implementation of a SYNC-General Purpose Input/Output (GPIO), which enables subsequent multi-sensor synchronisation. 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

Microcontroller firmware is software that is executed on a microcontroller in embedded systems. It controls the hardware and enables the execution of predefined functions. The firmware is used to process input data, control output devices and fulfil 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 sensors4.1 and reading them out. This measured data can be forwarded to a host Personal Computer (PC) via a user interface and can then be further processed there.

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 functions4.1 need to be implemented.

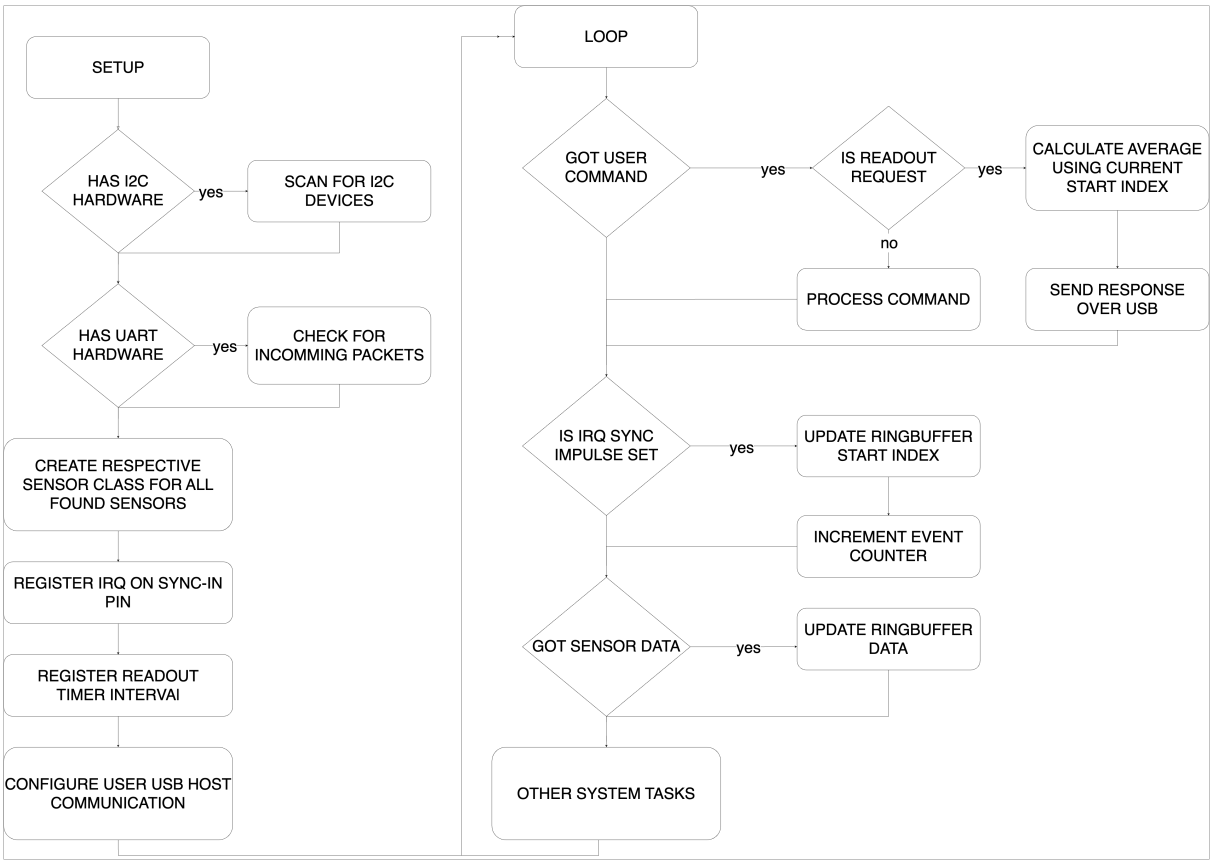

Figure **4**-3: Unified sensor firmware simplified program strucutre

```
#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;
  bool query_sensor() override;
  sensor_result get_result() override;
};
#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 information), the appropriate class instances are created and these can later be used to read out measurement results.

Next, the subsystem for multi-sensor synchronisation4.4.2 is set up. The last step in the setup is to set up communication with the host or connected PC. All microcontroller platforms used here have a Universal Serial Bus (USB) slave port. The used usb descriptor is Serial over (+usb)-((+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 systems.

Now that 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 buffers, offer 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 synchronized4.4.2 in time and are not out of sequence or drift.

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:

```
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

- Use with MRP-library5
- 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-library. The serial interface is also used here. However, after a connection attempt by the MRPHal module5.2.1 of the MRP-library, the system switches to binary mode, which is initiated using the sbm command. The same commands are available as for CLI-based communication.

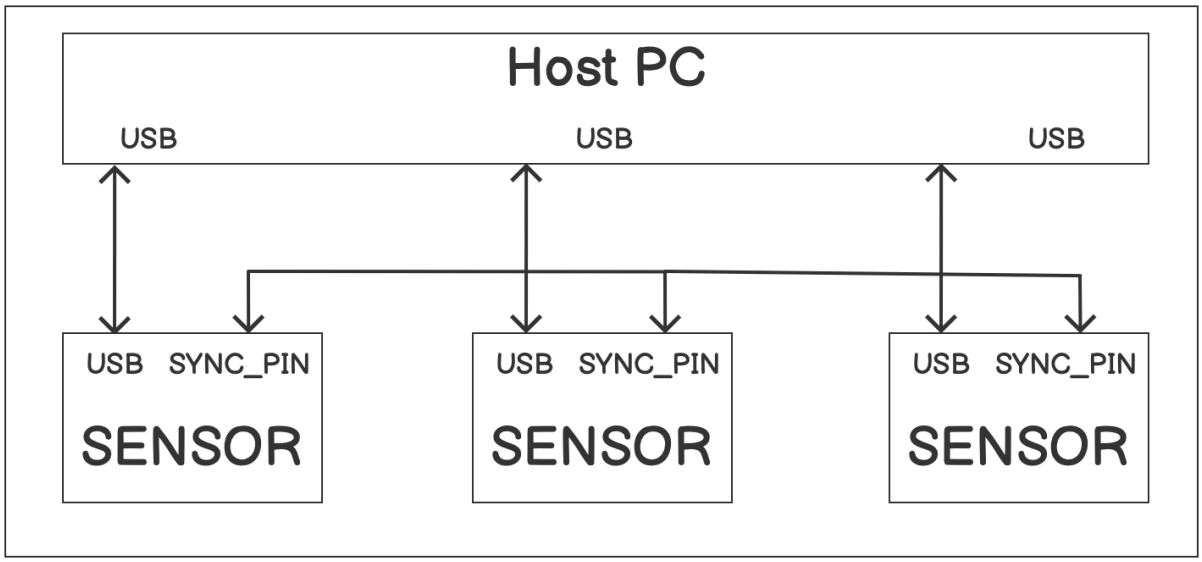

Figure **4**-6: Multi sensor synchronisation wiring example

4.4.2 Sensor syncronsisation 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

Run	Points	Host software runtime [ms]	Average Sensor communication time per reading [ms]	Communication jitter time [ms]	Comment
1	1	9453	1.44	0	
2	1	9864	1.5	0	
3	10	12984	1.22	0.9	
4	10	12673	1.13	1.1	

Run	Points	Host software runtime [ms]	Average Sensor communication time per reading [ms]	Communication jitter time [ms]	Comment
5	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 - debian bookworm x64,
- MRP5-library - version 1.4.1
- Unified Sensor Firmware4 - 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 software5. 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 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-pin**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

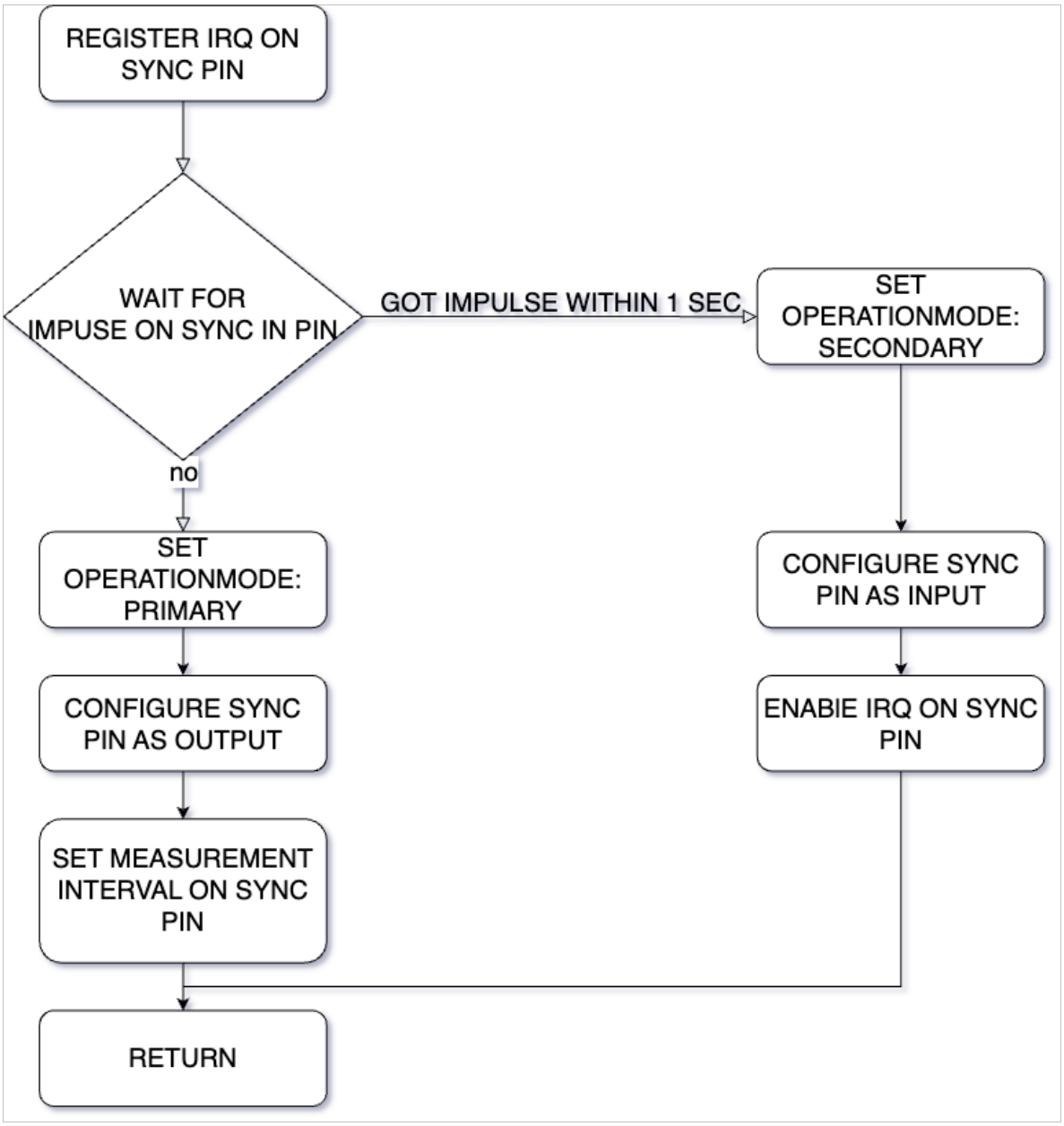

Figure **4**-7: Unified sensor firmware multi sensor synchronisation procedure

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 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 query. The synchronisation status can be queried via the user interface4.4.1 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 readout software5 by using the sensor identification number.

4.5 Example sensors

Two functional sensor platforms4.3 were built in order to create a solid test platform for later tests and for the development of the MRP 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)

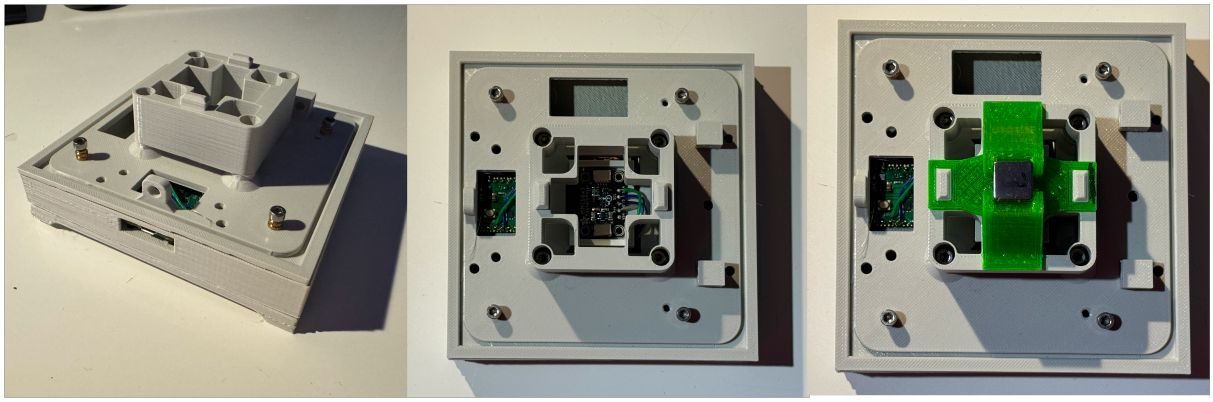

Figure **4**-9: 1D sensor construction with universal magnet mount

These cover all the required functions described in the Usecases3. 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 Firmware4.4 platform and also with the MRP-library5.

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 very limited measuring range, even small neodymium magnets already saturate its range, it is possible to use 3D-printed spacers above the sensor.

The standard magnet holder can be adapted for different magnet shapes and can be placed on the spacer without play in order to be able to perform a repeatable

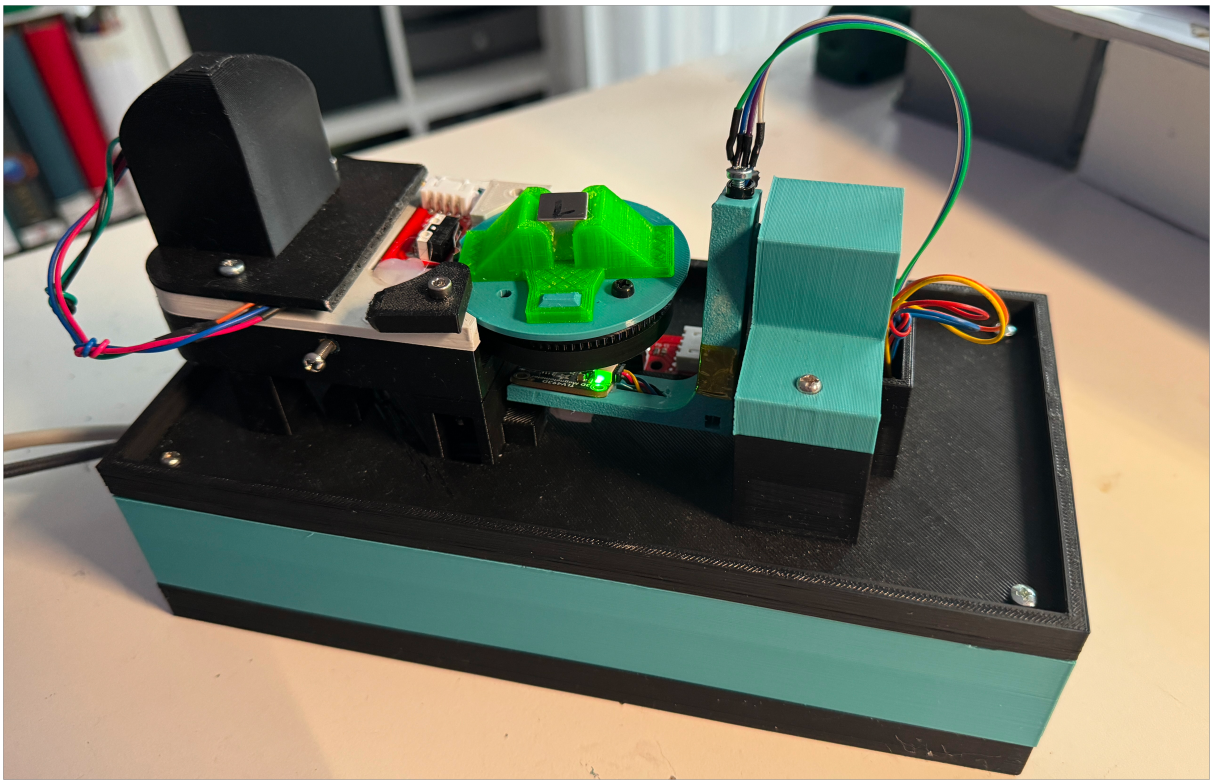

Figure **4**-10: Full-Sphere sensor implementation using two Nema17 stepper motors in a polar coordinate system

measurement without introducing measurement irregularities by mechanically changing the magnet.

4.5.2 3D: Fullsphere

The 3D Fullsphere**4**-10 sensor offers the possibility to create a 3D map**4**-11 of the magnets. 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 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

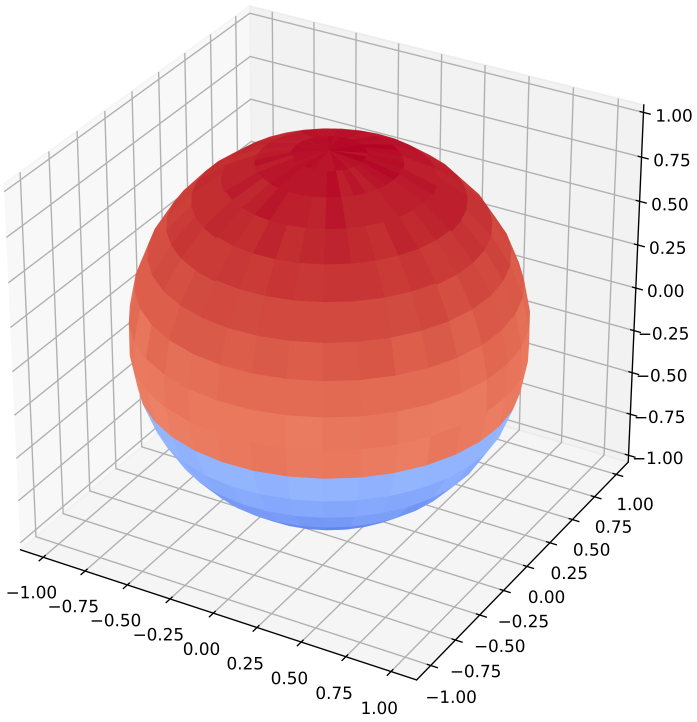

Figure **4**-11: 3D plot of an N45 12x12x12 magnet using the 3D fullsphere sensor

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. This is shown here using a temperature-compensated Voltcraft GM−70 telsameter**4**-12, which has a measuring range of 0−3T with a resolution of 0.1mT. It offers an RS232 interface with a documented protocol4.4 for connection to a PC. This connectivity makes it possible to make the device compatible with the MRP library using interface software [13] executable on the host PC. However, it does not offer the range of functions that the Unified Sensor Firmware4.4 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 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 Unified Sensor Firmware4.4 was adapted using a separate build configuration and the protocol of the measuring device was implemented.

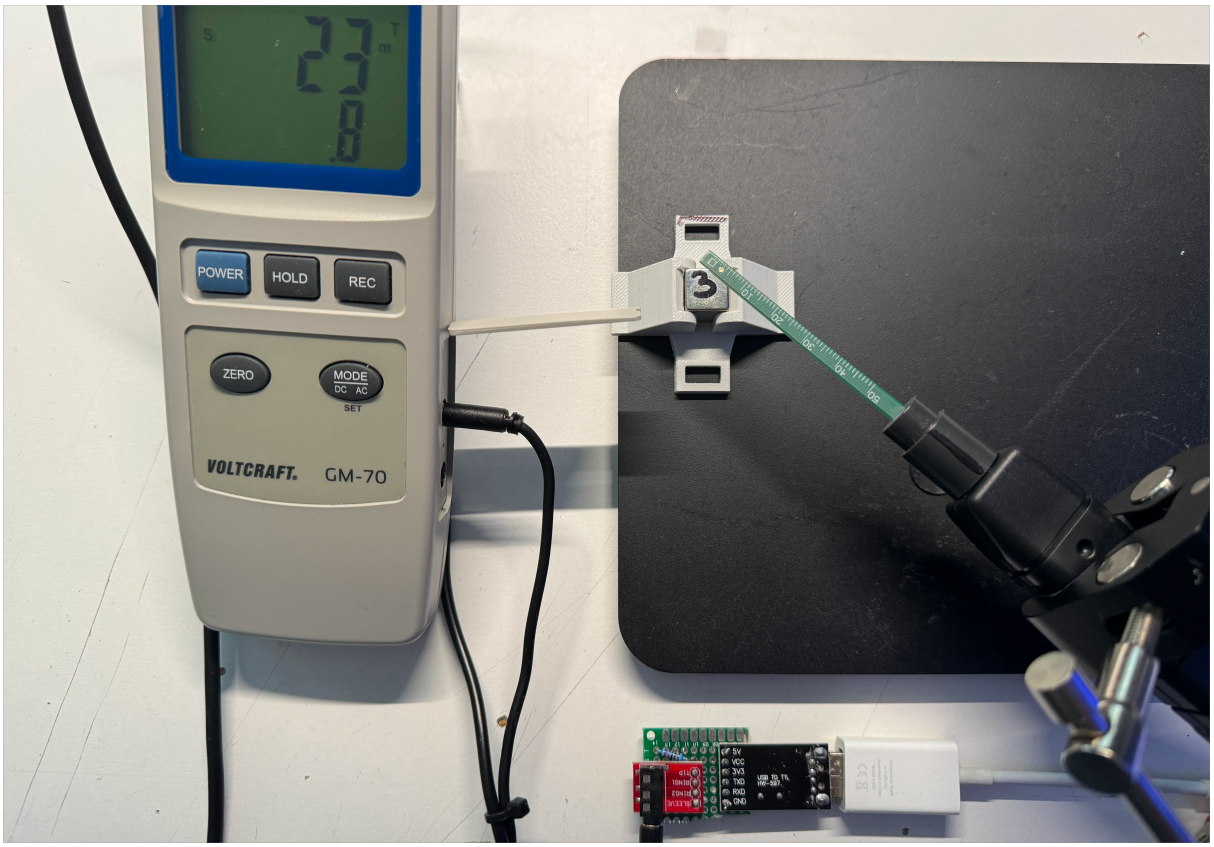

Figure **4**-12: Voltcraft GM70 teslameter with custom PC interface board

Tabelle 4.4: Voltcraft GM70 serial protocol

BYTE-INDEX	REPRESENTATION	VALUE
0	PREAMBLE	0x2
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

BYTE-INDEX	REPRESENTATION	VALUE
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

5.1 Library requirements

5.1.1 Concepts

- beispiele für projekte welche nur einzelne schnritte implementieren
- so kann man sich auf die implementierung

5.1.2 User interaction points

5.2 Extension Modules

Im folgenden werden die einzelnen module, welche vom user modifiziert und ersetzt werden können im details erleutert.

5.2.1 HAL

- aufbau hal im grunde wird nur ein die commandos an das sensor cli weitergegeben
- alle sensoren implementieren mehr oder weniger die gleichen befehle
- hal gibt nur weiter und ist “dumm”

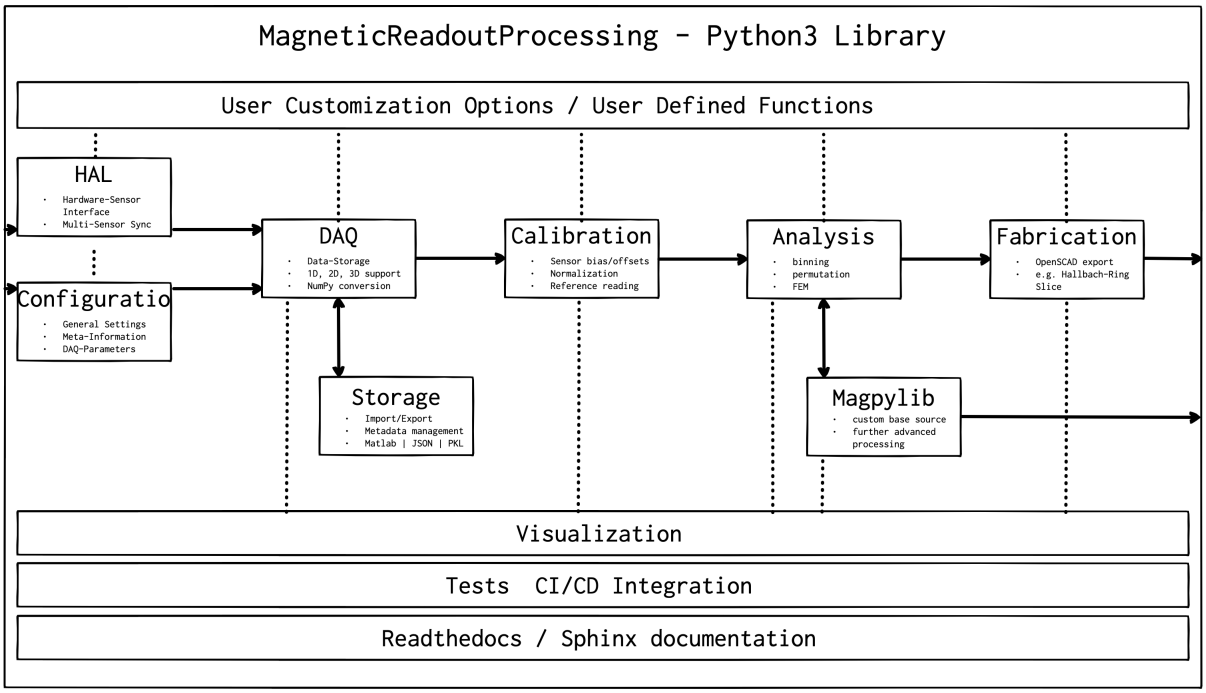

Figure **5**-1: MRPlib COMPLETE FLOW

5.2.2 Visualisation

5.2.3 Storage

- metadata management

5.2.4 Export

- format import export
- matlab

5.3 Multible sensor setup

At the current state of implementation, it is only possible to detect and use sensors that are directly connected to the PC with the MRP-library. It has the disadvantage that there must always be a physical connection. This can make it difficult to install

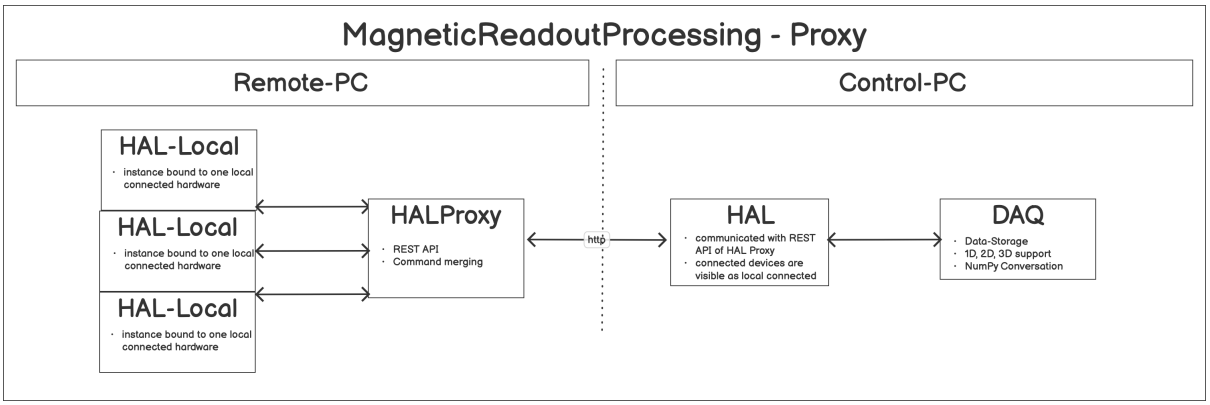

Figure **5**-2: MRPlib Proxy Module

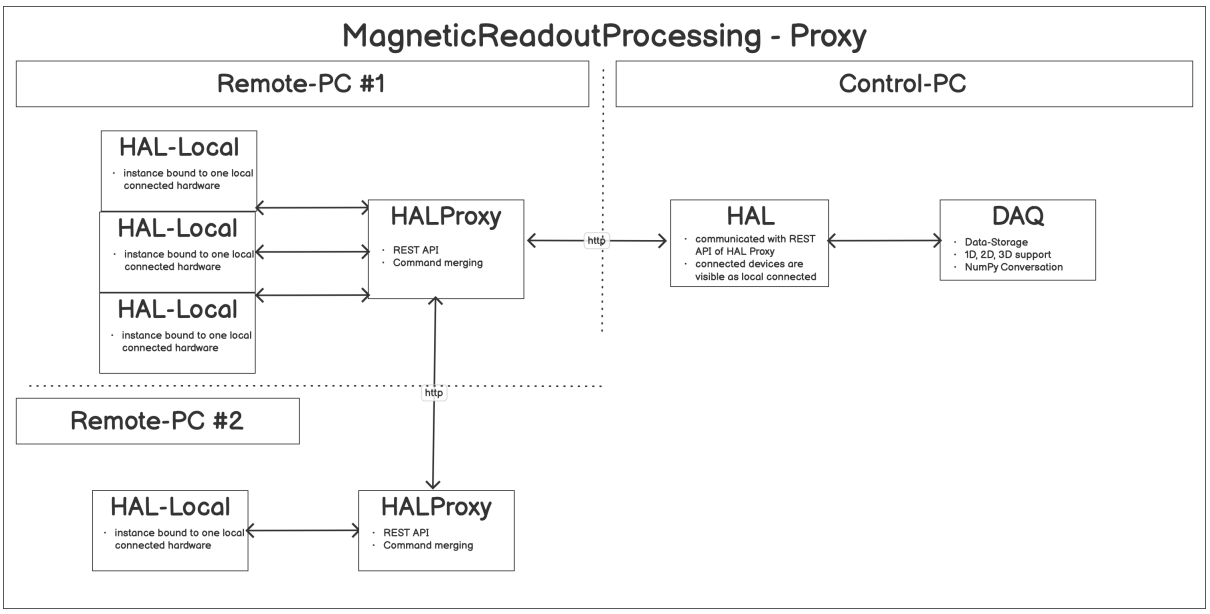

Figure **5**-3: mrp proxy multi

multiple sensors in measurement setups where space or cable routing options are limited. To make sensors connected to a small remote PC available on the network, the Proxy**5**-2 module has been developed. 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 approach of implementing this via a Representational State Transfer (REST) interface also offers the advantage that several measurements or experiments can be recorded at the same time with the sensors.

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 remote (+pc) #2 is connected to remote (+pc) #1. This is then visible as a sensor opposite the Control PC, even if there are several proxy instances behind it.

5.3.1 Network-Proxy

The figure **5**-2 shows the separation of the various Hardware Abstraction Layer (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 proxy application must always be started5.1 on the remote PC side.

```
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 5.1: 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:5.2

```
1 # GET PROXY STATUS
2 $ wget http://proxyinstance.local:5556/proxy/status
3 {
4 "capabilities":[
5   "static",
6   "axis_b",
7   "axis_x",
8   "axis_y",
9   "axis_z",
10  "axis_temp",
11  "axis_stimestamp"
12 ],
13 "commands":[
14  "status",
15  "initialize",
16  "disconnect",
```

```
17   "combinedsensorcnt",
18   "sensorcnt",
19   "readsensor",
20   "temp"
21 ]}
22 # RUN A SENSOR COMMAND AND GET THE TOTAL SENSOR COUNT
23 $ wget http://proxyinstance.local:5556/proxy/command?cmd=
     combinedsensorcnt
24 {
25 "output":[
26   "2"
27 ]}
```

Listing 5.2: MRPproxy REST enpoiint 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 other, only the Internet Protocol (IP) address or host name of the remote PC is required5.3.

```
1 # CONFIGURE MEASUREMENT JOB USING A PROXY INSTANCE
2 $ MRPcli config setupsensor testcfg --path http://proxyinstance.local
    :5556
3 > remote sensor connected: True using proxy connection:
4 > http://proxyinstance.local:5556 with 1 local sensor connected
```

Listing 5.3: 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**??**. this no longer works for long distances and the syncronisation must be made via the network connection.

If time-critical synchronisation over the network is required, Precision Time Protocol (PTP) and Puls Per Second (PPS) output functionality[9] can be used on many SBC, such as the Raspberry–Pi Compute Module 4.

5.3.3 Command-Router

- nummerieierung zuerst lokale sensoren dann weitere proxy sensoren
- commando templating
- einzelne sensor capabilites werden gemerged
- ids werden aufsuimmiert
- spätere sensor identifikation geschieht über die jeweilige sensor id, welche über diesen index abgefragt werden kann
- tablle mit mergin algorithmus

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 documentation6.4.1 contains further and more detailed examples. Many basic examples are also supplied in the form of test cases6.3.

5.4.1 MRPReading

```
1 from MRP import MRPReading, MRPMeasurementConfig
2 # [OPTIONAL] CONFIGURE READING USING MEASUREMENT CONFIG INSTANCE
3 config: MRPMeasurementConfig = MRPMeasurementConfig
4 config.sensor_distance_radius(40) # 40mm DISTANCE BETWEEN MAGNET AND
      SENSOR
5 config.magnet_type(N45_CUBIC_12x12x12) # CHECK MRPMagnetTypes.py FOR
      IMPLEMENTED TYPES
6 # CREATE READING
7 reading: MRPReading = MRPReading(config)
8 # ADD METADATA
9 reading.set_name("example reading")
10 ## ADD FURTHER DETAILS
11 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")
```

5.4.2 MRPHal

```
from MRP import MRPHalSerialPortInformation, MRPHal, MRPBaseSensor,
    MRPReadingSource
# SEARCH FOR CONNECTED SENSORS
system_ports: MRPHalSerialPortInformation = MRPHalSerialPortInformation
    .list_serial_ports()
sensor = MRPHal(system_ports[0])
# OR USE SPECIFIED SENSOR DEVICE
device_path: MRPHalSerialPortInformation = MRPHalSerialPortInformation(
    "/dev/serial/by-name/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: MRPReadingSource = MRPReadingSourceHelper.
    createReadingSourceInstance(sensor)
result_readings: [MRPReading] = reading_source.perform_measurement(
    _readings=1, _hwavg=1)
```

5.4.3 MRPSimulation

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

5.4.4 MRPAnalysis

Dieses beispiel zeigt

```
from MRP import MRPAnalysis, MRPDataVisualization, MRPReading
# CREATE EMPTY READING
reading = MRPReading.MRPReading()
# CREATE A SAMPLE MEASUREMENT
for i in range(1000):
    measurement = MRPReadingEntry.MRPReadingEntry()
    # readout sensor or use dummy data and assign result
    measurement.value = (random.random() -0.5) * 2
    reading.insert_reading_instance(measurement, False)
# PLOT MEAN VISUALIZATION
MRPDataVisualization.MRPDataVisualization.plot_error([reading])
# APPLY COMPENSATION
# Here the ``calculate_mean`` function is used
reading_mean_value = MRPAnalysis.MRPAnalysis.calculate_mean(reading)
# we want to subtract the mean value from all readings
```

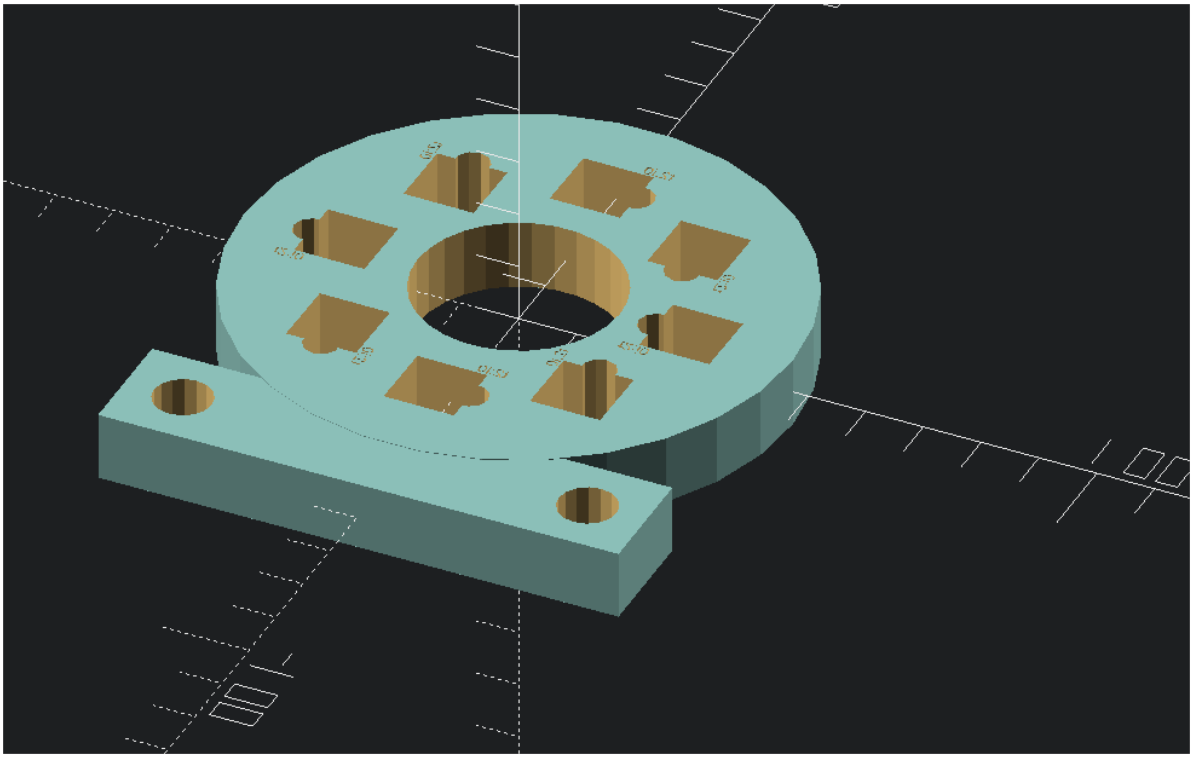

Figure **5**-4: Generated hallbach array with generated cutouts for eight magnets

```
16 reading_mean_value = -reading_mean_value
17 # modify measurement values
18 MRPAnalysis.MRPAnalysis.apply_global_offset_inplace(reading,
       reading_mean_value)
```

5.4.5 MRPVisualisation

5.4.6 MRPHallbachArrayGenerator

The following example code5.4, 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[14] using the MRPHallbachArrayGenerator module.

```
readings = []
for idx in range(8):
  # GENERATE EXAMPLE READINGS USING N45 CUBIC 15x15x15 MAGNETS
  readings.append(MRPSimulation.MRPSimulation.generate_reading(
    MRPMagnetTypes.MagnetType.N45_CUBIC_15x15x15))
## GENERATE HALLBACH
hallbach_array: MRPHallbachArrayGenerator.MRPHallbachArrayResult =
    MRPHallbachArrayGenerator.MRPHallbachArrayGenerator.
    generate_1k_hallbach_using_polarisation_direction(readings)
# EXPORT TO OPENSCAD
## 2D MODE DXF e.g. for lasercutting
MRPHallbachArrayGenerator.MRPHallbachArrayGenerator.
    generate_openscad_model([hallbach_array], "./2d_test.scad",
    _2d_object_code=True)
## 3D MODE e.g. for 3D printing
MRPHallbachArrayGenerator.MRPHallbachArrayGenerator.
    generate_openscad_model([hallbach_array], "./3d_test.scad",
    _2d_object_code=False)
```

Listing 5.4: 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[11] file, which contains the module generated. After computing the model using the OpenSCAD-CLI utility, the following model rendering**5**-4 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. Standardised naming conventions and well thought-out default values simplify the application. 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 community feedback and regular updates promotes the adaptability of the MRP-library to changing needs. 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 this 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

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

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 around this MRP-library. The (+mrp)-library-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 with the MRP-library and processed further. The following bash code6.1 shows the setup procedure in detail:

```
1 # CLI EXAMPLE FOR CONFIGURING A MEASUREMENT RUN
2 ## CONFIGURE THE SENSOR TO USE
3 $ MRPcli config setupsensor testcfg
4 > 0 - Unified Sensor 386731533439 - /dev/cu.usbmodem3867315334391
5 > Please select one of the found sensors [0]:
6 > sensor connected: True 1243455
7 ## CONFIGURE THE MEASUREMENT
8 $ MRPcli config setup testcfg
9 > CONFIGURE testcfg
10 > READING-NAME: [testreading]: testreading
11 > OUTPUT-FOLDER [/cli/reading]: /tmp/reading_folder_path
12 > NUMBER DATAPOINTS: [1]: 10
13 > NUMBER AVERAGE READINGS PER DATAPOINT: [1]: 100
14 # RUN THE CONFIGURED MEASUREMENT
15 $ MRPcli measure run
16 > STARTING MEASUREMENT RUN WITH FOLLOWING CONFIGS: ['testcfg']
17 > config-test: OK
18 > sensor-connection-test: OK
19 > START MEASUREMENT CYCLE
20 > sampling 10 datapoints with 100 average readings
21 > SID:0 DP:0 B:47.359mT TEMP:23.56
22 > ....
23 > dump_to_file testreading_ID:525771256544952_SID:0_MAG:
     N45_CUBIC_12x12x12.mag.json
```

Listing 6.1: CLI example for configuring a measurement run

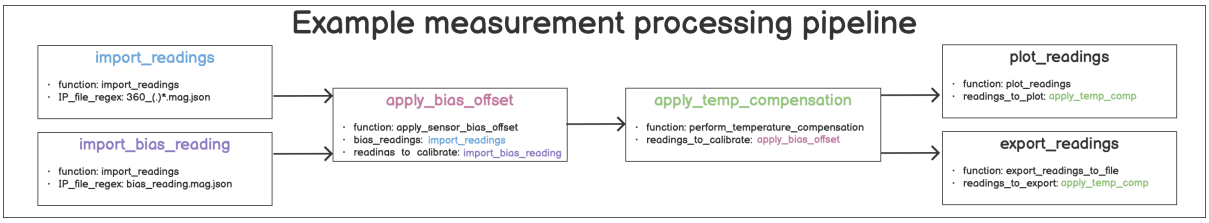

Figure **6**-2: example measurement analysis pipeline

6.2 Programmable data processing pipeline

After it is very 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 easy to understand and can also be easily edited with a text editor. Detailed examples can be found in the documentation[2]. 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[6]. This system makes almost all Python functions available to the user. To simplify use, a pre-defined list of tested MRP library functions for use in pipelines is listed in the documentation[2]. The following pipeline definition6.2 shows the previously defined steps**6**-2 as yaml syntax.

```
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 User Defined Processing Pipeline

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 > as the 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:

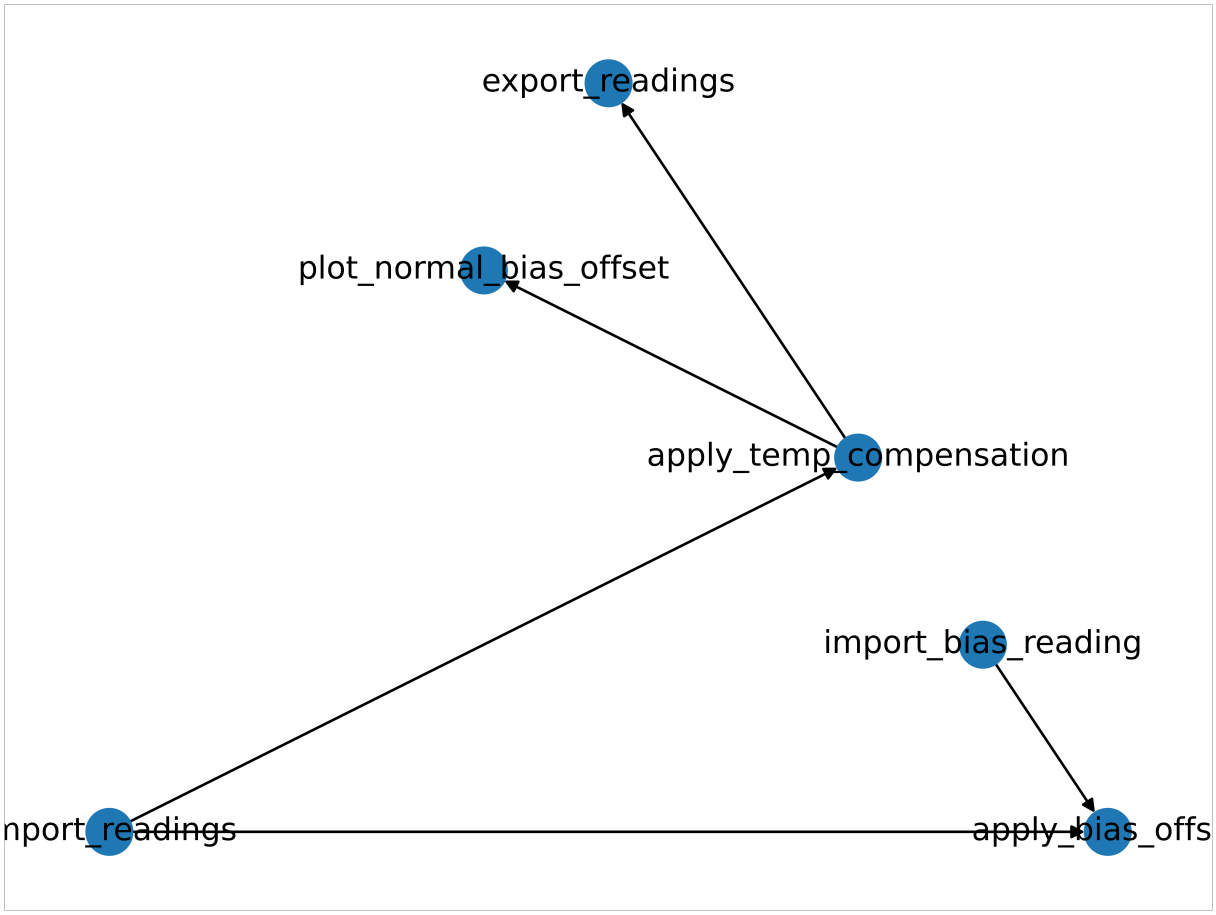

Figure **6**-3: Result step execution tree from user defined processing pipeline example

- 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.

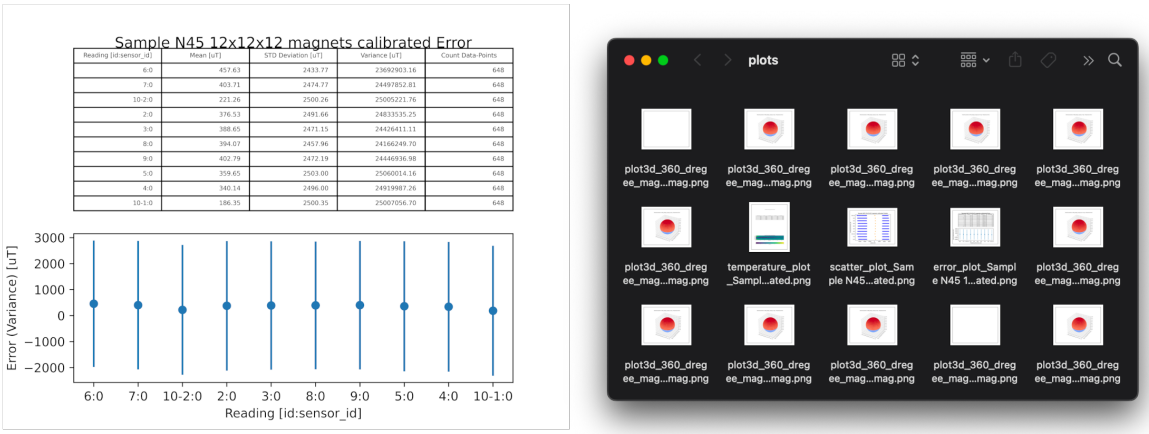

Figure **6**-4: pipeline output files after running example pipeline on a set of readings

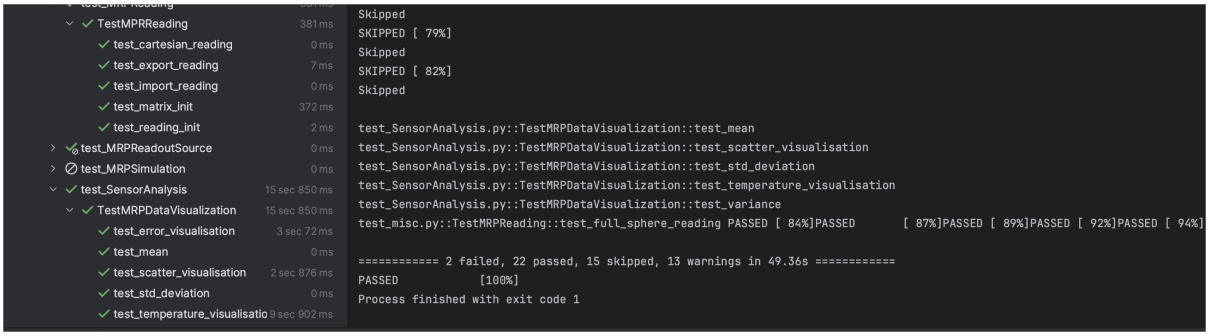

Figure **6**-5: MRP library test results for different submodules executed in PyCharm IDE

6.3 Tests

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 use cases. The test framework PyTest[12] 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 use cases were mapped using the test cases created, the code of the test cases could later be used in slightly simplified variants6.3 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[10]. 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

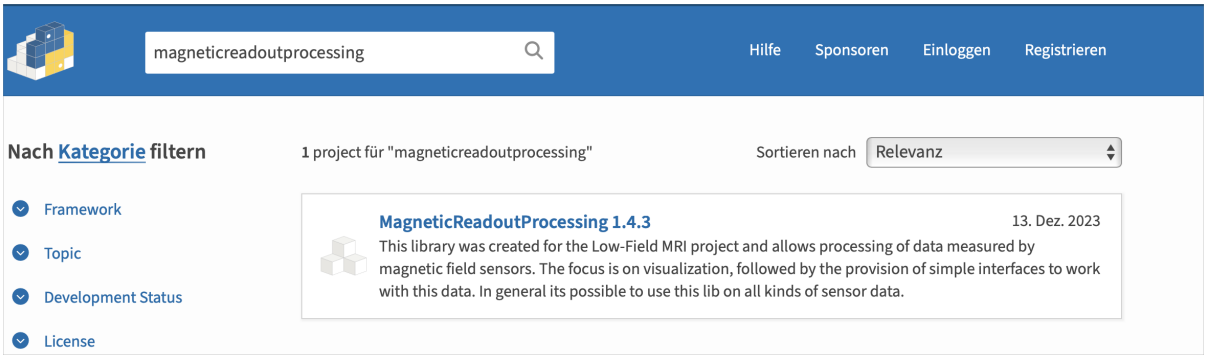

Figure **6**-6: MagneticReadoutProcessing library hosted on PyPi

software and executed via virtualisation on the systems provided by Github Actions[10].

6.4 Package distribution

One important point that improves usability for users is the simple installation of the MRP-library. As it was created in the Python programming language, there are several public package registry where users can provide their software modules. Here, PyPi [5]**6**-6[4] 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 (+mrp)-library using pip

To make the MRP-library 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 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 MRP-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[8] 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.**??**

```
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: Python docstring example

Since ‘docstrings’ only document the source code, but do not provide simple how-to-use instructions, the documentation framework Sphinx[1] 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

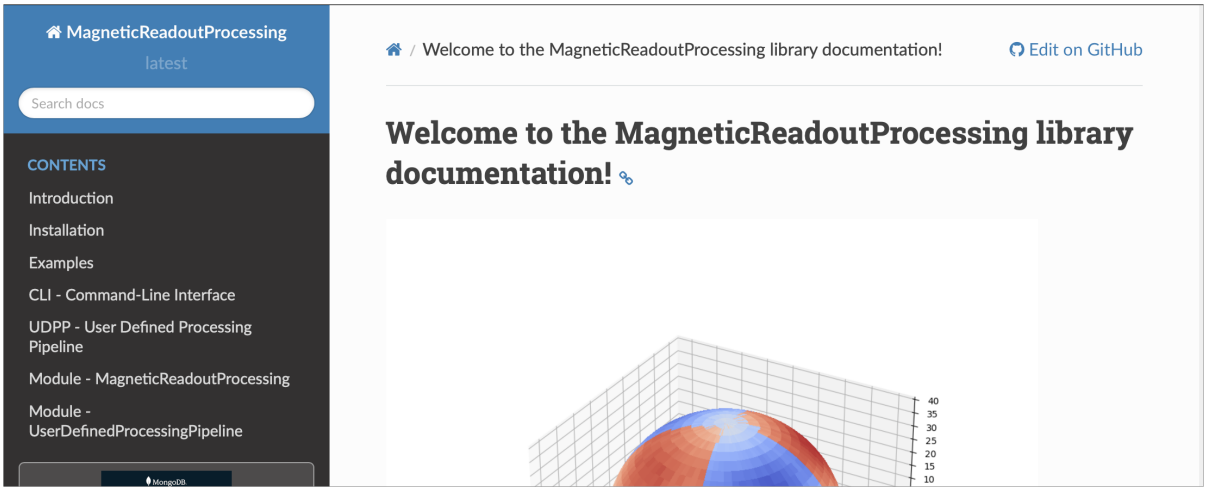

Figure **6**-7: MagneticReadoutProcessing documentation hosted on ReadTheDocs

services, which provide Python 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[3], where users can make documentation for typical software projects available to others.

The documentation has also been uploaded for ReadTheDocs[2] 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[10], 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 magnets3. 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.

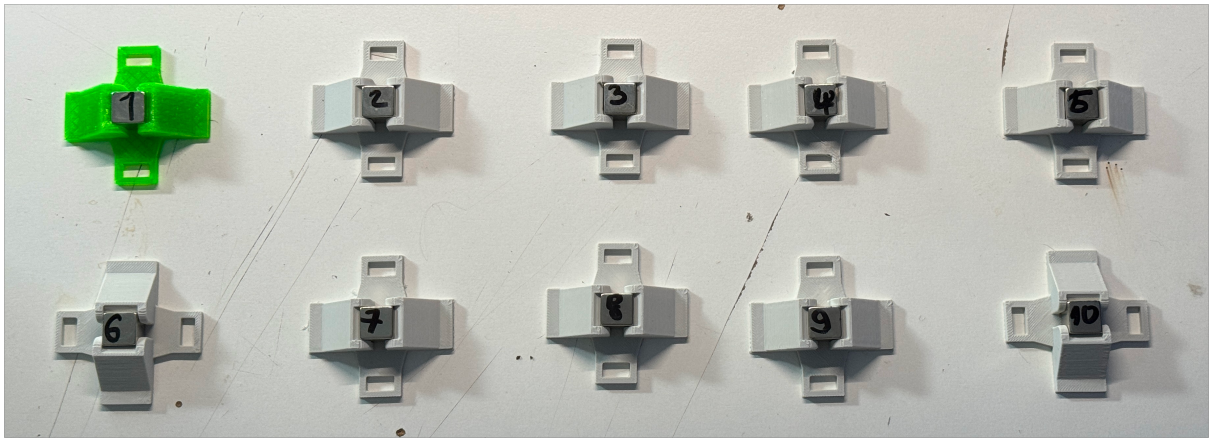

Figure **7**-1: testmagnets in holder

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 eval_measurement_config-CLI command. The measurement was then configured7.1.

```
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 carry out 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 algorithm7.2. 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 **import** 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[7] 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.

```
1  settings:
2    enabled: true
3    export_intermediate_results: false
4    name: pipeline_mrp_evaluation
5
6  stage import:
7    function: import_readings
8    parameters:
9      IP_input_folder: ./readings/evaluation/
10     IP_file_regex: 360_(.)*.mag.json
11
12 stage find_similar_values:
13   function: custom_find_similar_values_algorithm
14   parameters:
15     _readings: stage import # USE RESULTS FROM import STAGE
16     IP_return_count: 4 # RETURN BEST 4 of 10 READINGS
17
18 stage plot_filtered:
19   function: plot_readings
20   parameters:
21     readings_to_plot: stage find_similar_values # USE RESULTS FROM
       find_similar_values STAGE
22     IP_export_folder: ./readings/evaluation/plots/plot_filtered/
23     IP_plot_headline_prefix: MRP evaluation - filtered
```

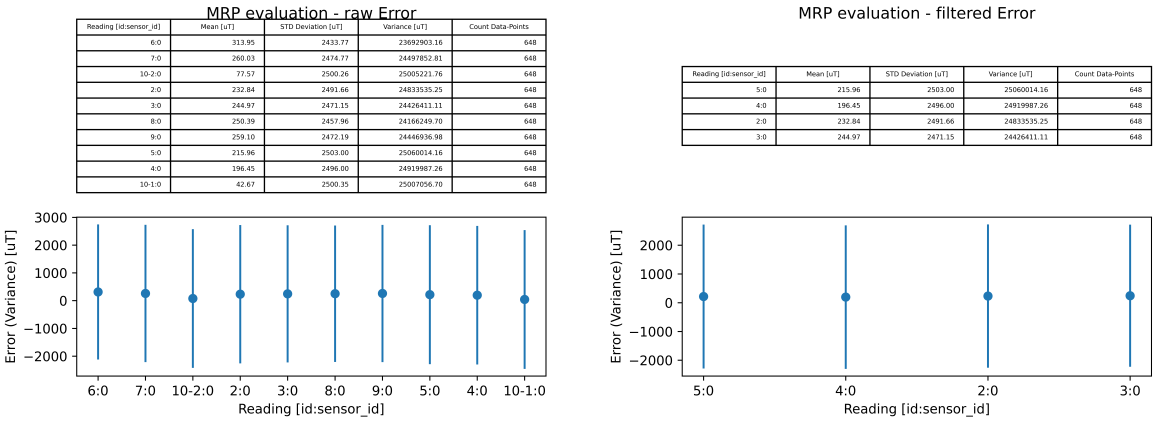

Figure **7**-2: MRP evaluation result

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 208.8 uT. The algorithm was thus successfully executed using the pipeline and the results could be validated.

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

In this version of the framework, a solid foundation has been laid that includes all the necessary features and is ready for use. During development, particular emphasis was placed on comprehensive documentation to make it easier to get started. Together with examples for various use cases, a user can quickly evaluate the framework.

However, it should be noted that the framework is still in its infancy. 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] DEVELOPERS, Sphinx: *Sphinx makes it easy to create intelligent and beautiful documentation.* https://www.sphinx-doc.org/en/master/. Version: 01.01.2024

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

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

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

[5] 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

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

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

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

[9] 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

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

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

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

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

[14] 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

List of Figures

List of Tables

Listings