

Helmholtz Test Bench Control Software Documentation

Author:

Martin Zietz

Leon Teichröb

Supervisors:

M.Sc. Markus T. Koller

M.Sc. Lukas-Maximilian Loidold

Institut für Raumfahrtssysteme, Universität Stuttgart
März 2021

Contents

List of Figures	ii
List of Tables	iii
List of Symbols	iv
1 Software Implementation	1
1.1 Program Structure	1
1.2 Program Files	2
1.2.1 Main Executable <code>main.py</code>	2
1.2.2 Global Variables File <code>globals.py</code>	2
1.2.3 Test Bench Control File <code>helmholtz_cage_device.py</code>	3
1.2.4 Graphical User Interface <code>User_Interface.py</code>	4
1.2.5 Sequence Execution File <code>csv_threading.py</code>	7
1.2.6 Calibration Procedures <code>calibration.py</code>	8
1.2.7 Configuration Handling File <code>config_handling.py</code>	8
1.2.8 Data Logging Handling File <code>csv_logging.py</code>	8
1.2.9 TCP Remote Control <code>socket_control.py</code>	9
1.2.10 Hardware Control Libraries	9
1.3 Conversion to Executable	10
2 Operating Instructions	12
2.1 Test Bench Assembly Instructions	12
2.1.1 Disassembly Procedure	12
2.1.2 Reassembly Procedure	14
2.2 Software Users Guide	15
2.2.1 Installation	15
2.2.2 User Interface Elements	15
2.2.3 Hardware Connections and Program Setup	26
2.2.4 TCP Remote Control	27
Bibliography	29
A Example Program Auxiliary Files	30

List of Figures

1.1	Software file layout and architecture	1
1.2	User interface (example state)	5
2.1	Test bench assembly drawings	12
2.2	Main connector (X-cables temp.)	13
2.3	Intermediate (dis-)assembly step	13
2.4	Manual input mode user interface	16
2.5	CSV sequence mode user interface	18
2.6	Ambient field and coil constant calibration view.	19
2.7	Magnetometer calibration view.	21
2.8	Data logging configuration page	22
2.9	Program settings page	24

List of Tables

2.1	Status display entries	15
2.2	Settable program constants	25
2.3	TCP Remote Control Commands	27

List of Symbols

Symbol	Description	Unit
B	Magnetic flux density	T
B_0	Ambient magnetic flux density	T
B_{max}	Maximum achievable magnetic flux density	T
B_{min}	Minimum achievable magnetic flux density	T
I	Current	A
I_{max}	Maximum current	A
K	Coil constant	$\frac{T}{A}$
R	Electrical resistance	Ω
U	Voltage	V
U_{max}	Maximum voltage	V

1 Software Implementation

1.1 Program Structure

To operate the test bench, a Python software with a graphical user interface (UI) was developed. It manages the test bench hardware, such as the PS2000B and QL355TP Power Supply Units (PSU), the polarity switch box, and magnetometers. This chapter focuses on the overall software implementation. Detailed information is provided in the form of comments in the source code, which is available on the IRS git server¹. For a user manual, please refer to Section ??.

Software development and testing were done in Windows 10 and Python 3.9. Some aspects may need to be adapted to use the software on a different operating system or Python version. The software in its current state (26. October 2021) has been fully tested with the complete hardware suite. For the complete system test, a FGM3D magnetometer, QL355TP PSU, and the polarity switcher were used and are thus recommended as a reference configuration.

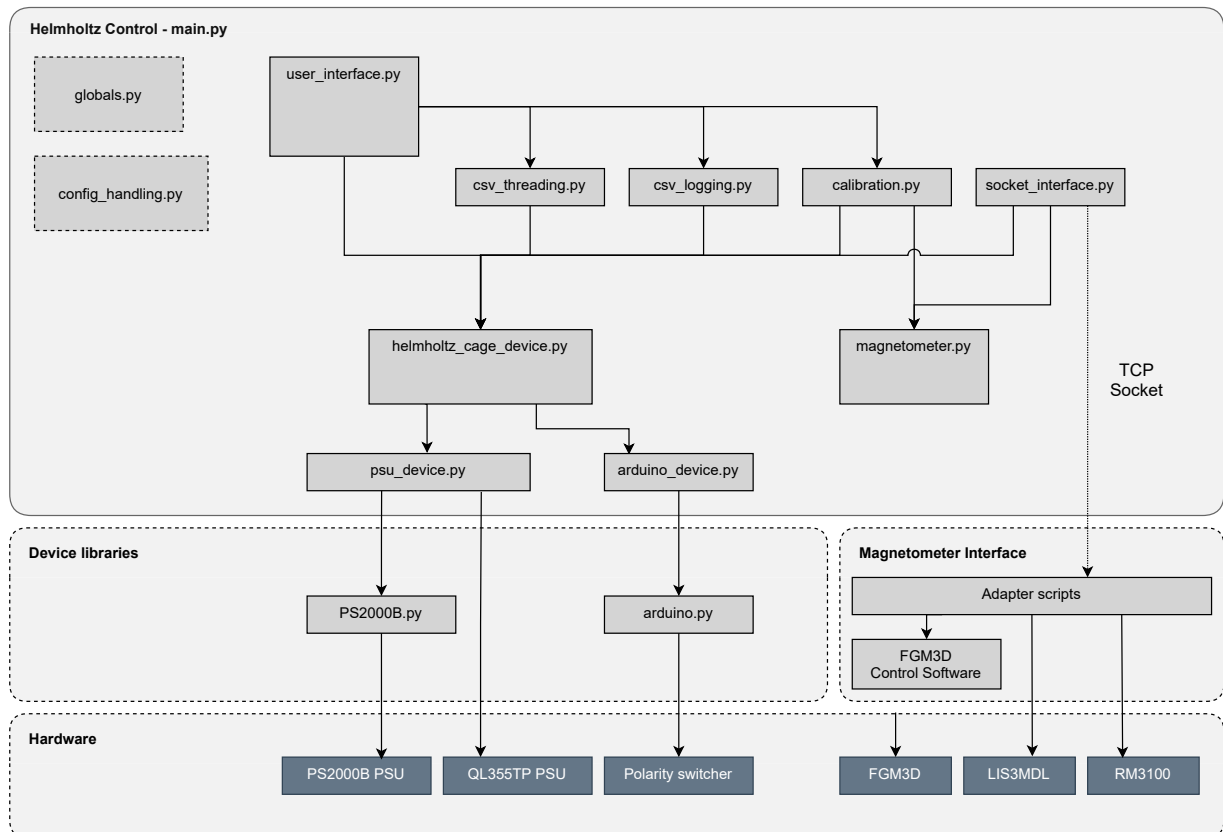


Figure 1.1: Software file layout and architecture

¹https://egit.irs.uni-stuttgart.de/eive/Helmholtz_Test_Bench.git

The program file architecture is shown in Figure ?? . This is meant to give an overview of the structure, therefore it does not show all interactions between the files.

Upon execution of `main.py`, a test bench proxy object is initialized that acts as a wrapper around the PSU and Arduino interfaces and calls. Next, the UI (controlled by `user_interface.py`) is set up and displayed. Advanced software functionality is split into separate files and components that are called from the UI code. This includes the CSV and calibration functionality for example. All program elements use the global `HelmholtzCageDevice` object instance defined in the `helmholtz_cage_device.py` file to control and read data from the test bench. Communication with some of the hardware is achieved via modified third-party libraries such as `PS2000B.py` and `Arduino.py`, which were taken from online sources [1, 2]. `globals.py` is used to easily pass frequently used variables between the different program files. It contains mainly constants and defaults as well as some global instances, that can be accessed and changed by the other program elements after importing it. The application is able to operate with one or all test bench devices disconnected. For example, if one PSU were to be disconnected, commands and fields could still be applied to the axes accessible on the other PSU. The program also provides error handling for a variety of device disconnects and user mistakes. This includes protection against the commanding of excessive currents and voltages, as well as the display of warnings when attempting to enter a potentially dangerous value in the settings.

1.2 Program Files

1.2.1 Main Executable `main.py`

The main executable file contains the code to start (`program_start`) and stop (`program_end`) the program. During program initialization the information from the default configuration file is read out to set the necessary constants. The startup defaults may be modified according to the currently used hardware setup. Afterwards, the global Helmholtz cage and magnetometer proxy device instances are created. These instances are requisite for the following UI initialization, but do not yet need to be connected to hardware devices. Afterwards, the graphical and TCP interfaces are created, and the hardware initialization is triggered in a non-blocking, asynchronous function call. Finally, the UI's `mainloop` is started.

The `program_end` function in this file ensures a safe shutdown of all equipment at the end of the program. It is immediately called when the user closes the window. The function stops any other operational threads (see Section ??) and powers down the PSUs and Arduino. It also asks the user to store possible unsaved log data and then destroys the application window.

For exceptions that were not caught by lower levels, the main file also includes application-level error handling. In such a case an error message is displayed and the `program_end` function called to ensure a safe shutdown, regardless of the current program state.

1.2.2 Global Variables File `globals.py`

This file holds global variables and constants that are shared between modules. Examples of these are the `CAGE_DEVICE` and `MAGNETOMETER` objects that represent hardware devices, but also status indicators like `exit_flag`, which signals the end of the program to all components. The file also contains the default values for all constants used to run the test bench, like coil constants, resistances, and maximum currents. These are stored in a hard-coded dictionary, which is also used to regenerate the default configuration. The dictionary also includes the minimum and

maximum safe value for each constant, which are used to warn users if they attempt to set a value that may damage equipment. In the future, it may be advantageous to store this information in a separate configuration file to allow easier modification.

1.2.3 Test Bench Control File `helmholtz_cage_device.py`

This file contains all classes directly related to the operation of the test bench. It includes the main control class `HelmholtzCageDevice`, its proxy `HelmholtzCageProxy`, as well as the `Axis` adapter object used by both. Magnetometer interfacing is handled in the separate `magnetometer.py` file. The application contains a single global `HelmholtzCageDevice` instance, contained in the `CAGE_DEVICE` variable.

Main Control Class `HelmholtzCageDevice`

This class provides the following important functionality:

- Asynchronous connection and disconnection of hardware devices (`connect_hardware_async` and `shutdown`). These functions may be called at any time during program execution, and are used as the primary means of reloading configuration variables and connecting new hardware at runtime.
- Set currents, raw fields, and compensated fields in the test bench coils. Uses `_set_signed_currents`, `_set_field_raw`, and `_set_field_compensated`. These functions are called internally when the corresponding command is received in the `_cmd_exec_thread`.
- Hardware status and control is mediated by an array of `Axis` adapter objects. Each axis stores its own specific configuration data and related status information.
- Components may provide callbacks to the `subscribe_status_updates` function, to receive periodic status information about the test bench. This is more sensible than polling, since this operation is slow.

Device access is limited to one application component at once by means of a proxy pattern. With the `request_proxy` and `release_proxy` functions, up to one `HelmholtzCageProxy` instance may be obtained if none are currently active. This proxy must be released with the latter function, to allow for another component to request a proxy instance. All commands to the test bench are set by means of the proxy object, except for startup and shutdown commands, which must always be available to the application.

The hardware connection sequence goes as follows:

- Read the current config variables (see Section ??).
- Attempt to connect to Arduino on specified com port.
- Select PSU object type based on configuration variable.
- Attempt to create PSU object instances.
- Axis specific config variables are reloaded.

- All PSU channels are zeroed and activated. The Arduino relay pins are set to their default “low” state.

The disconnection sequence:

- Zero and disable all PSU output channels. The PSU serial device is disconnected.
- The Arduino relay pins are set to low, and the serial device is closed.
- Information regarding the shutdown process is presented to the user.

Axis Adapter Class `Axis`

This class is an adapter object representing a single axis (X, Y, Z) of the test bench. It contains static and dynamic status information about the axis in its attributes and the means to command the axis in its methods. During program initialization an object of this class is created for each of the three axes and stored in the global `HelmholtzCageDevice` instance. The class contains the primary way to command the test bench hardware in form of the `set_signed_current` method. Its parameter is a positive or negative current value in Ampere. The method first checks if this value exceeds the safe limits defined in the program configuration. If not, the appropriate PSU channel and the switch box Arduino are commanded to supply the desired current and to actuate the polarity change relay as needed. To generate a specific magnetic field, the `set_field_compensated` method can be called. It takes a given field value B in Tesla and calculates the needed current:

$$I = \frac{B - B_0}{K} \quad (1.1)$$

B_0 is the background magnetic field in the measurement area for this axis in Tesla and K is the coil constant in $[\frac{T}{A}]$. The calculated value I is passed to the `set_signed_current` method to command the test bench. The `set_field_raw` method works the same way, but without subtracting the ambient field. Similarly the minimum and maximum achievable field values B_{min} and B_{max} are calculated in the class initialization, mainly in order to provide this information to the user:

$$B_{max} = B_0 + I_{max} * K \quad (1.2)$$

$$B_{min} = B_0 - I_{max} * K \quad (1.3)$$

Here I_{max} is the maximum allowed current, as defined in the configuration page of the UI.

Value checking function `value_in_limits`

This function checks whether a value is within the safe limits defined in `globals.py`.

1.2.4 Graphical User Interface `User_Interface.py`

This file is used to construct the UI, that the user interacts with to control the test bench. It contains several classes, each with code to build a specific section of the UI as well as methods to perform the actions that are triggered by different user inputs in that section.

The general layout of the application window is shown in Figure ?? . At the bottom left is a status display, that is constantly updated with the current state of all connected devices. To its right

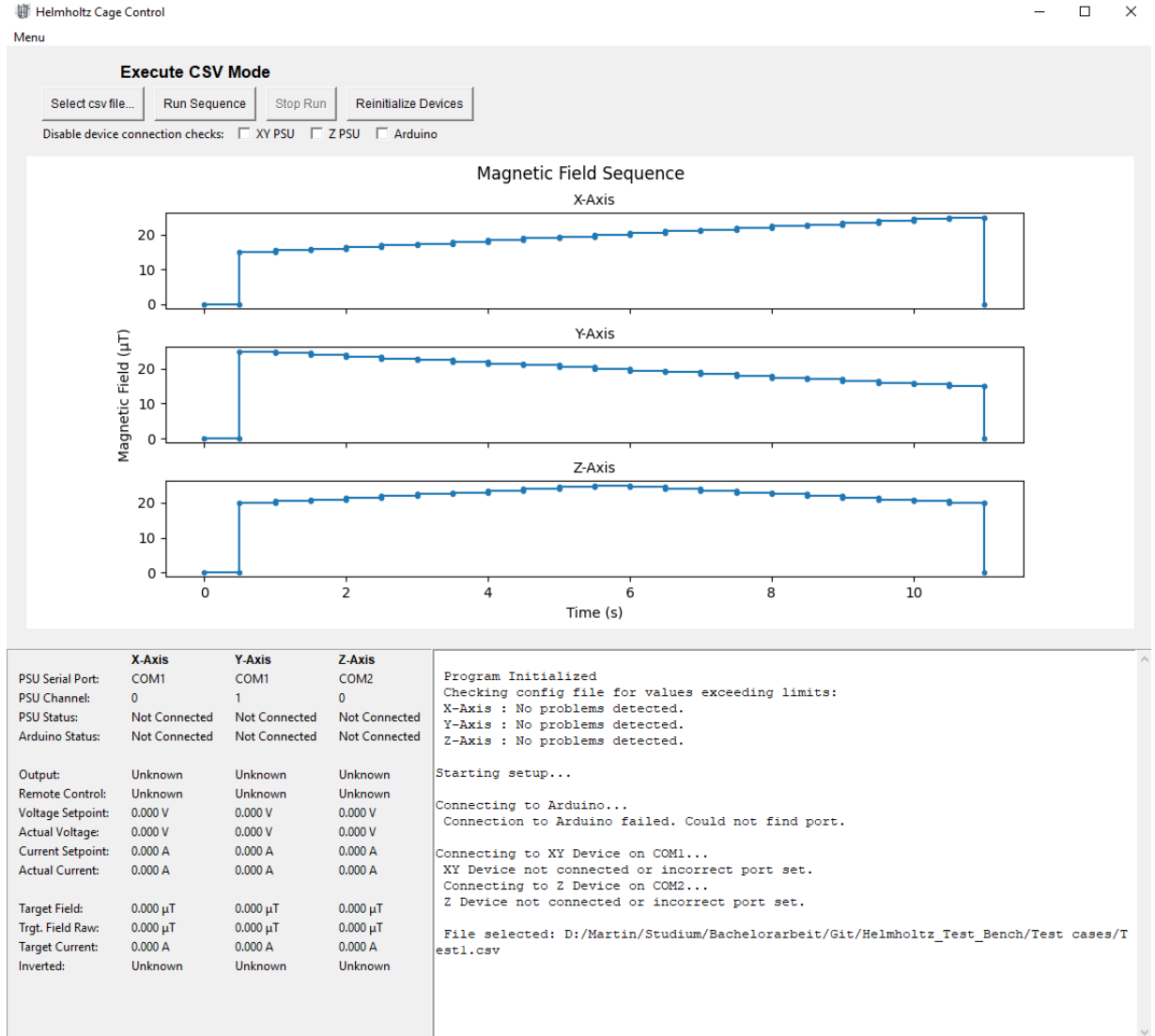


Figure 1.2: User interface (example state)

is a text area that can be used to print out information to the user, similar to a basic console output. The top half is the main area, used to display the interactive elements. Its content changes depending on which mode the user has selected. At the very top of the window a drop-down menu is used to switch between the different modes. For more detailed usage instructions please refer to Section ??.

The interface is setup through a central object of class `HelmholtzGUI`, which inherits the `Tk` main application class from the `tkinter` library. The other UI element classes (except the top menu) inherit from `tkinter.Frame` and are placed in the layout by the main object. This structure is based on a proposal by H. Kinsley [3]. Each class is briefly described below.

Main Application Class `HelmholtzGUI`

An instance of this class represents the application window. The program `mainloop` is running on this object. The major UI elements are initialized here and placed at their appropriate positions.

This class's only method is `show_frame`, which is used to switch between the different operating modes by displaying their respective frame in the main area.

Menu Bar Class `TopMenu`

The menu bar at the very top of the application window is constructed by this class. It contains methods to implement each option of the menu. At this time, it is only used to switch between modes, but more functionalities could be added in the future if needed.

Static Manual Input Mode Class `ManualMode`

The manual input mode interface is used to manually command static values of magnetic fields or currents on the test bench. It is placed in the main area of the application window, the layout can be seen in Figure ???. The methods provided in this class interface mainly with the `globals.CAGE_DEVICE` instance to command the devices.

CSV Sequence Execution Mode Class `ExecuteCSVMode`

This class constructs and operates the interface for executing magnetic field sequences from a Comma Separated Values (CSV) file. It is placed in the application's main area, its layout is shown in Figure ???. Buttons are provided to load and run CSV sequences. To execute the sequence without blocking the UI, a separate execution thread defined in `csv_threading.py` is created and managed by this class. Hardware control is acquired upon creation of the thread, or an exception is returned otherwise. More details on the execution thread is provided in Section ??.

Magnetometer Calibration Mode Class `CalibrateMagnetometer`

This class constructs magnetometer calibration interface. It is placed in the application's main area, its layout is shown in Figure ???. This class creates and manages a separate execution thread defined in `calibration.py` to perform the calibration without blocking the UI. Hardware control is acquired upon creation of the thread, or an exception is returned otherwise. More details on the calibration thread is provided in Section ???. Communication with the running thread is performed by means of the `view_mpi_queue` queue variable, which is regularly polled in `update_view`. This is where different Message Passing Interface (MPI) commands are delegated to their respective handlers. For example, the live completion status and final calibration results are handled here. It should be of note, that the calibration threads do not store any results or information, instead these are contained in variables defined near the top of `__init__`.

Ambient Field Calibration Mode Class `CalibrateAmbientField`

This class consists of an identical structure to the Magnetometer Calibration View Class. The only major difference is that this view provides an interface for two calibration procedures; The ambient field and the coil constants. All relevant calibration worker threads can also be found in `calibration.py`.

Settings Page Class `Configuration`

The program settings page is constructed in this class. It provides a UI to edit application settings and also controls the reading and writing of configuration files. Like the test bench control modes, the settings page is placed in the application's main area. Its layout can be seen in Figure ???. It interfaces with the `config_handling.py` file to store this information in a

`ConfigParser` object. The `ConfigParser` object can then also be read and written to and from configuration files (see Section ??). Individual entry fields are automatically highlighted when values exceed the safe limits defined in `globals.py` to notify the user. Usage instructions and additional information can be found in Section ??.

Data Logging Configuration Page Class `ConfigureLogging`

This class generates the page for configuring and controlling the logging of test bench data to a CSV file. Similarly to the settings page it is placed in the main area, its layout is shown in Figure ??. The page provides a number of checkboxes used to select what specific data is to be logged. This information is stored in a dictionary of log options. Whenever a row of data needs to be logged, a method in the class calls the `log_datapoint` function in `csv_logging.py`. To indicate what data to log, the config dictionary is passed as the function parameter (see Section ??).

There are two options to control when and how often data is logged: in regular intervals and on event. These can also be used at the same time. The information, which of these is enabled, is stored in two boolean attributes of the class object. For logging in regular intervals a method in the `ConfigureLogging` class object periodically calls itself. For logging on event, a data point is generated whenever a significant change on the test bench is commanded. As of now, this is simply done by inserting a piece of code in every function where it was seen as appropriate. However, this is somewhat inconsistent and carries the risk of missing some state changes. A better balance between logging consistency and excessive data generation should be devised in a future update.

Status Display Class `StatusDisplay`

The status display (bottom left in Figure ??) is used to monitor the state of the test bench devices. The individual values are displayed and updated through labels tied to variables of type `tkinter.StringVar()`, which are stored in a dictionary. `StatusDisplay` subscribes to status updates from `HelmholtzCageDevice`, by registering a callback that queues the new information. The `update_labels` method is called with the new status information and updates the label variables. A shared queue and regular polling for new status information in the main thread is used to avoid calling Tkinter code outside the main thread. Updates are received from the cage object at a polling rate of 1 s.

Output Console Class `OutputConsole`

The output console in the bottom right of the UI (see Figure ??) is generated in this class. The main body is a `tkinter.Text` widget. Printing of information to it is done via the `ui_print` function defined in `utility.py`, which can be used exactly like the built-in `print`.

1.2.5 Sequence Execution File `csv_threading.py`

This file contains code for executing a timed sequence of magnetic field vectors from a CSV file. To do this without interfering with the UI, it must run in a separate thread.

The main class for this is `ExecCSVThread`. It inherits the `Thread` class from the `threading` library, so each of its instances represents a unique thread. Its main method, apart from those needed to start and stop it, is `execute_sequence`. It takes the desired sequence in the form of a `numpy` array and commands the test bench at the appropriate times. The method also

continuously polls that all devices are still connected and that the run has not been aborted by user input or closing of the main application window.

Apart from the main class this file contains functions to read data from a CSV file to a `numpy` array and check that no values in it exceed the test bench limits. There is also a function to generate a plot of the data for display in the UI. The line plot is modified to visually reflect the discrete and nearly instantaneous change of fields in the test bench (see result in Figure ??).

1.2.6 Calibration Procedures `calibration.py`

This file contains the worker thread objects `AmbientFieldCalibration`, `CoilConstantCalibration`, `MagnetometerCalibrationSimple`, and `MagnetometerCalibrationComplete`. All of these threads start by checking for and acquiring hardware during instantiation, while still in the main thread. Then upon running, the main code in `calibration_procedure` is executed; This function may freely block and make use of sleep commands. Typically, calibration procedures should save two data detail levels: Processed data that is displayed as a final result, and detailed, raw data points that can be exported by the user to apply custom algorithms or verify the applications function.

The calibration procedures communicate with the main thread via a MPI queue. Depending on whether the procedure completes successfully or not, a `finished` or `failed` status signal is sent. In addition, a `calibration_data` message is sent together with “`finished`” to convey the final results to the main thread. Other signals may also be used, such as `progress` to indicate the current completion status to the main thread. Data that should be passed to the calibration procedure from the main thread, is done so through the constructor.

1.2.7 Configuration Handling File `config_handling.py`

This file contains functions and variables for reading and writing configuration files. The processing is done using the `configparser` library.

The current program configuration is stored in `CONFIG_OBJECT`, an instance of `ConfigParser`. It contains all configurable information and can be stored in its entirety in a `.ini` file. The `write_config_to_file` and `get_config_from_file` functions are used to read/write the entire `CONFIG_OBJECT` from/to such a file. An example is provided in Appendix ??.

To access or change specific values the `read_from_config` and `edit_config` functions are provided. The latter also checks if a value is within the safe limits defined in `globals.py`. If it is not, the user is warned and asked to confirm, before the value is written to `CONFIG_OBJECT`.

The `check_config` function does this check for all values of a provided configuration object. If excessive values are found, a warning message is displayed and the configuration UI page opened, where they are highlighted.

The last function of the file is `reset_config_to_default`, which overwrites the entire `CONFIG_OBJECT` with the default values defined in `globals.py`.

1.2.8 Data Logging Handling File `csv_logging.py`

This file handles the logging of test bench status data to a CSV file, using the `pandas` library. Within the program the logged data is stored in a `pandas.DataFrame` object, which represents a table with column headers and data rows.

A `logging_selection_options` dict, contains a key-value list of logging options and their display labels, which are used to generate the UI. The `ConfigureLogging` class of the UI then later

generates a list containing the keys belonging to the ticked checkboxes, which is then passed to the logging functions to indicate what data should be logged. To log a data point, this option dict is passed to the `log_datapoint` function. According to the options set, data is aggregated from the `CAGE_DEVICE` and `MAGNETOMETER` components and assembled into a key-value dictionary. Since no unified set of keys covering all of the logged parameters exist, new, understandable keys were selected. The dict of log values is appended to the end of a Python list that collects every data point. When the logging is finished, a CSV file is assembled and saved using the `write_to_file` function. Assembly consists of first prepending two headers to the data rows, one containing the dictionary keys, and one containing the user-readable long names. The long names can be obtained from the `get_long_column_header` function.

The file also provides functions to allow the user to choose a filename and clear the logged data.

1.2.9 TCP Remote Control `socket_control.py`

This class is responsible for enabling TCP control of the Helmholtz cage hardware. In addition, it is the sole data path for magnetometer data to ingress into the control software. As with all other software components, the command processing thread accesses the hardware using the global `g.CAGE_DEVICE` and `g.MAGNETOMETER` instances, and uses the proxy model to prevent control collisions.

The `socket_control.py` file defines the `SocketInterfaceThread` class, which provides a listener thread to accept incoming TCP connections. The listener thread instance is defined at application start-up in `main.py`. For each established connection, a `ClientConnectionThread` instance is created. All the main logic and command parsing is handled the client connection/session threads. The API is documented in Chapter ?? and also in the application source code.

Note: Following an application crash, the application may present an error in acquiring the TCP port. This may be fixed by waiting a few minutes, or restarting the computer. Perhaps this special case can be fixed with a more rigorous exit handler in the future.

1.2.10 Hardware Control Libraries

“PSUDevice” Abstract Class and PSU Drivers `psu_device.py` / `PS2000B.py`

An abstract class `PSUDevice` describing any compatible power supply object is used to allow for quick and easy addition of new device support. This improvement was implemented since power supplies may be replaced or upgraded on a moderately frequent basis. The abstract class exposes functions to set currents and voltages, and also to query the current device status. QL355TP devices are natively supported with the `PSUDeviceQL355TP` object, that is also defined in `psu_device.py`.

Since the code was at some point migrated to require a consistent interface for PSU objects, the original PS2000B library had to be put in a wrapper class (`PSUDevicePS2000B`). The external library provides the class `PS2000B` and some supporting functions, and was adapted from S. SpröBig [1] with only minor modifications. More information can be found in the read me file provided by the original author. The library objects provide effectively the same set of functions advertised by the abstract class, as such the wrapper is quite trivial. In the future, it may make sense to migrate the library code directly into the `PSUDevicePS2000B` class.

The main modification to the PS2000B library done as part of this thesis, was to implement independent commanding of both PSU channels. For this an additional parameter `channel`

(integer type, 0 or 1) was added to all methods of the PS2000B class that interact with the device. Additionally the properties PS2000B.output1, PS2000B.voltage1 and PS2000B.current1 were duplicated to support the second channel (e.g. PS2000B.output2).

Arduino Command API `arduino.py` / `arduino_device.py`

To control the Arduino microcontroller, an interface and wrapper class is used. First, the online library from [2] is integrated without major modifications to expose a hardware control API. It communicates with the provided `prototype.ino` Arduino program to allow toggling and control of its pins with methods that closely resemble the ones used in the standard Arduino programming language. More details can be found in the read me file provided by the original author [2].

The `Arduino` class is then wrapped in the `ArduinoDevice` class defined in `arduino_device.py`, to expose a more polarity-switcher-relevant API. This simplifies its use in the already complex `HelmholtzCageDevice` function.

Magnetometer Device `magnetometer.py`

The magnetometer object is hardware-agnostic, and merely mirrors the state of an external physical device. To enable quick and optimal compatibility with new magnetometer devices, data is ingested by means of the TCP port and the application functions purely as a listener. External magnetometer interface scripts will first push their data according to the API defined in `socket_control.py`, which will then be forwarded to the global `MagnetometerProxy` instance. This means the magnetometer proxy object is purely passive, and functions like a simple data container.

The magnetometer object mirrors the state of the TCP connection, which is presented as a connected/disconnected state.

1.3 Conversion to Executable

The program is compiled to a .exe executable file to enable simple use without a Python installation. For this, the "Auto Py to Exe" tool developed by B. Vollebregt [4] is used. The resulting files are distributed in a separate repository on the IRS git server.² When a new software version is ready for publication, the following procedure should be used to release it:

1. `pip install auto-py-to-exe` in the project's virtual environment.
2. Execute "auto-py-to-exe" in a console. To make sure the right version is executed, don't install auto-py-to-exe in your system environment.
3. In the Auto Py to Exe UI, select `main.py` file as "Script Location"
4. Select options "One File" and "Window Based"
5. Select file "Helmholtz.ico" as the "Icon"
6. Select "Releases" in the development repository as the output folder
7. Execute conversion
8. Rename resulting "main.exe" file to "Helmholtz Cage Control.exe"

²https://egit.irs.uni-stuttgart.de/zietzm/Helmholtz_Test_Bench_Releases

9. Verify correct program operation from created executable
10. Copy new executable to the release repository² and replace previous version
11. Commit, merge and create a new release in the git web interface
12. Record changes in release notes and changelog and update the documentation

2 Operating Instructions

2.1 Test Bench Assembly Instructions

Because of the limited space available in the IRS cleanroom, the test bench may need to be disassembled and reassembled in the future. Instructions for this are provided here. Mentioned position numbers relate to those shown in Figure ??.

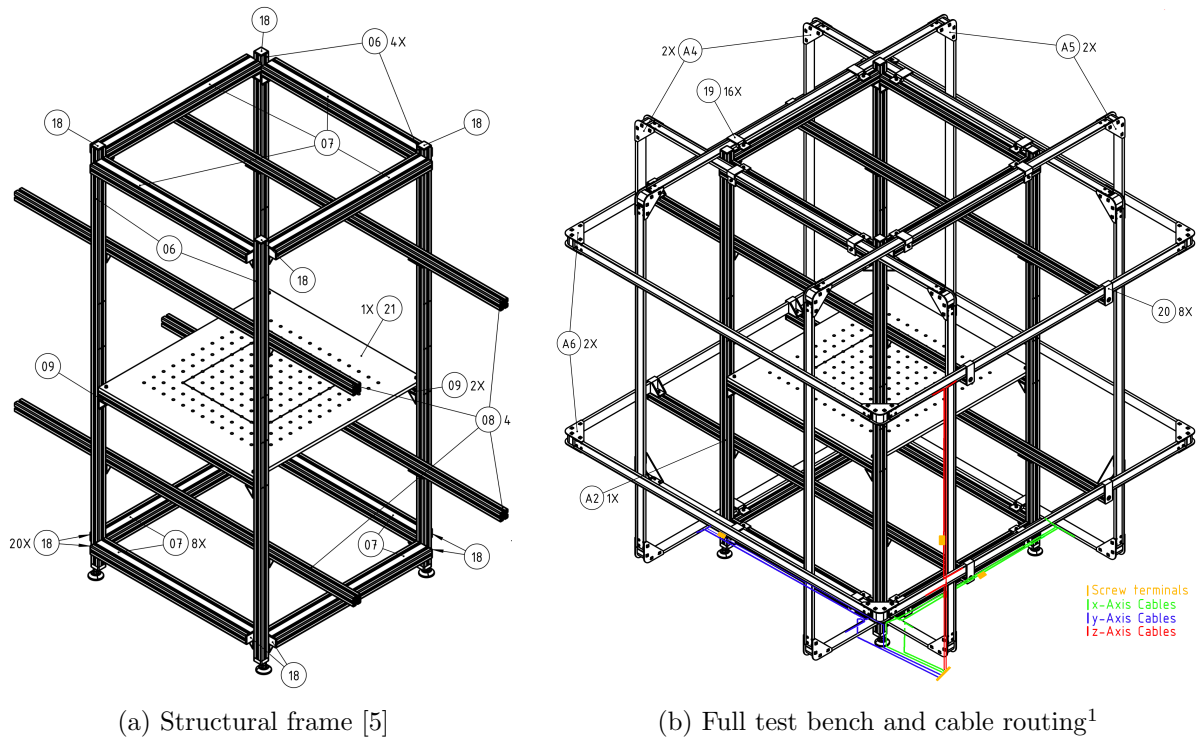


Figure 2.1: Test bench assembly drawings

2.1.1 Disassembly Procedure

Notes:

The coils must always be supported on two sides during handling to avoid bending.

The used slot nuts are wedged in the structural profiles and will stay in place even when their screw is removed. In most cases this is desirable, to mark the correct positions for reassembly. If one needs to be removed, loosen the attached screw and gently tap its head with a hammer.

All removed screws must be stored orderly, as they are needed for future reassembly!

¹Base drawing from [5]

Procedure:

1. Undo cabling
 - a) Disconnect switch box from main cable and PSUs
 - b) Disconnect coil cables from central screw terminal (Fig. ??)
 - c) Disconnect screw terminals between coils, leave terminal on one side
 - d) Untie wires from their guiding structures
2. Remove Z-axis coils
 - a) Unscrew upper coil brackets (20) from profiles (08), but leave them on the coil
 - b) Lift off upper coil (A6)
 - c) Remove upper coil support profiles (08), leave angle pieces on main structure
 - d) Repeat procedure for lower coil
3. Remove Y and X-axis coils
 - a) Unscrew coil brackets (19) for one Y-axis coil (A5) from frame profiles (07), but leave them on the coil
 - b) Lift off coil (A5)
 - c) Repeat procedure for second coil (A5) and X-axis coils (A4)
4. Remove mounting plate (21)
5. Lay remaining frame on its side, mounting plate profiles (09) should be on top and bottom (orientation shown in Figure ??)
6. Separate "upper" rectangular frame section (highlighted yellow in Figure ??)
 - a) Remove bolts of angles connecting rectangle to cross-member profiles (07), angle pieces should remain on "upper" rectangle
 - b) Lift off entire rectangle section
7. Remove cross-member profiles (07) from "lower" rectangle section, angles stay attached

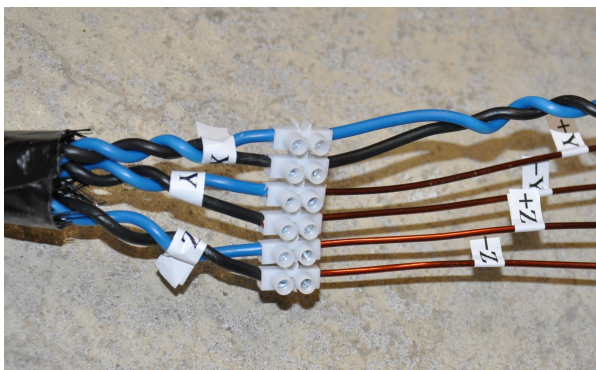


Figure 2.2: Main connector (X-cables temp.)

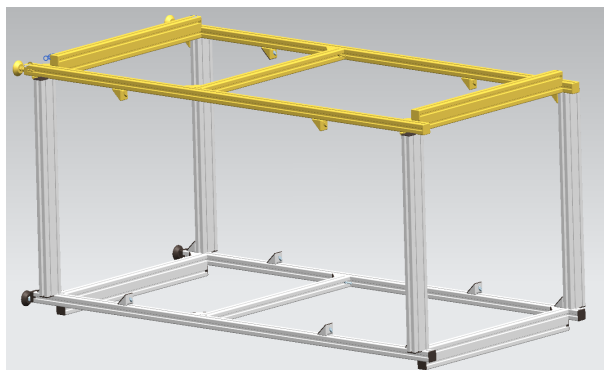


Figure 2.3: Intermediate (dis-)assembly step

2.1.2 Reassembly Procedure

Notes:

These instructions assume a disassembly according to Section ???. Most slot nuts should still be in place and simplify positioning the parts. However, the angle pieces do allow some tolerance, so care should still be taken to get the correct alignments.

Procedure:

1. Lay one main frame rectangle on the floor as shown in Figure ??
2. Attach cross-member profiles (07)
3. Attach second frame rectangle on top, state should now be as shown in Figure ??.
4. Lift frame upright and adjust foot heights for solid stand
5. Attach mounting plate (21)
6. Install X and Y-axis coils
 - a) Lift +X coil (A4) onto cross-member profiles (07); mind wire exit position (Fig. ??)
 - b) If needed, adjust cross-member profile height; coil should be supported equally on upper and lower profiles without bending
 - c) Centre coil and secure with 3D-printed brackets (19)
 - d) Repeat for -X (A4) and Y-axis coils (A5)
7. Install Z-axis coils (A6)
 - a) Attach lower coil support profiles (08)
 - b) Lift -Z coil over X/Y coils onto support profiles; mind wire exit position (Fig. ??)
 - c) Centre coil and secure with 3D-printed brackets (20)
 - d) Repeat for upper support profiles and +Z coil
8. Make electrical connections
 - a) Connect screw terminals between coils (unmarked wire ends)
 - b) Route and tie down wires as shown in Figure ??
 - c) Check that there are no short circuits between coil wires and frames
 - d) Connect main screw terminal as shown in Figure ??
 - e) Connect main cable to switch box
 - f) Verify contact on all axes (multimeter at switch box inputs), resistances approx. $3.1\ \Omega$
 - g) Connect switch box to PSUs: X-axis to PSU 1 channel 1, Y-axis to PSU 1 channel 2, Z-axis to PSU 2 channel 1
9. Setup laptop and initialize control program as described in Section ??
10. Verify correct polarity on all axes through magnetic field measurements at different currents
11. Calibrate test bench (exact procedure to be developed)

2.2 Software Users Guide

2.2.1 Installation

1. Download latest release: https://egit.irs.uni-stuttgart.de/eive/Helmholtz_Test_Bench/releases
2. Unpack ZIP-folder and run "Release\Helmholtz Control.exe"
3. Setup hardware and program according to Section ??

2.2.2 User Interface Elements

The general UI layout is shown in Figure ?. The upper area contains the interactive elements of each operating mode and settings page. Switching between these pages is done using the top "Menu" bar. The lower half of the UI contains the status display and a console output.

Status Display

The status display shows the current state of the test bench devices. Explanations of the different values are given in Table ?.

Table 2.1: Status display entries

Entry	Explanation
PSU Serial Port:	Serial COM port used to connect to the PSU for this axis.
PSU Channel:	PSU output channel for this axis (0→channel 1, 1→channel 2).
PSU Status:	Connection status of PSU for this axis. Possible states: <ul style="list-style-type: none"> • <i>Connected</i>: communication nominal • <i>Not Connected</i>: PSU not found during initialization • <i>Connection Error</i>: PSU was connected but then an error occurred, e.g. it was disconnected
Arduino Status:	Connection status of switch box Arduino (identical for all axes). Possible states: <ul style="list-style-type: none"> • <i>Connected</i>: communication nominal • <i>Not Connected</i>: Arduino not found during initialization • <i>Connection Error</i>: Arduino was connected but then an error occurred, e.g. it was disconnected
Output:	Status of PSU output channel. Possible states: <ul style="list-style-type: none"> • <i>Active</i>: set current or voltage is applied to the output jacks • <i>Inactive</i>: output jacks are unpowered • <i>Unknown</i>: no connection to PSU
Remote Control:	Status of PSU channel remote interface. Possible states: <ul style="list-style-type: none"> • <i>Active</i>: channel can be controlled remotely • <i>Inactive</i>: channel can not be controlled remotely • <i>Unknown</i>: no connection to PSU
Voltage Setpoint:	Maximum voltage PSU is set to supply.

Continued on next page

Table 2.1 – Continued from previous page

Entry	Explanation
Actual Voltage:	Voltage across PSU channel output jacks. Usually lower than "Voltage Setpoint", as voltage is throttled to achieve the desired current.
Current Setpoint:	Current the PSU output channel is set to supply
Actual Current:	Current flowing through the PSU output channel.
Target Field:	Desired magnetic flux density in the measurement area.
Trgt. Field Raw:	Flux density used to calculate needed current (after ambient field compensation).
Target Current:	Desired current to flow through the coils. Negative values mean reversed polarity.
Inverted:	Status of polarity change relay for this axis inside the switch box. Possible states: <ul style="list-style-type: none"> • <i>True</i>: (Arduino pin "HIGH"→relay switched→polarity inverted, status light-emitting diode (LED) should be illuminated) • <i>False</i>: pin "LOW"→opposite of "True" state • <i>Unknown</i>: no connection to Arduino

Manual Mode

The manual input mode is used to set static currents or magnetic fields on the test bench. Its layout is shown in Figure ???. The main UI elements are listed below.

Figure 2.4: Manual input mode user interface

- **"Select Input Mode" drop-down:** Switches between setting currents or magnetic fields
- **Value entry fields:** Enter values to be set on the test bench here
 - Entries must be numeric (decimal point)
 - Entries must be in indicated safe range (may be changed in the settings page)

- **"Compensate ambient field" checkbox:** When ticked, the ambient magnetic field (set in settings page) is subtracted from entered values before commanding the test bench (inactive for "Current" input mode)
- **"Execute" button:** Implements values from the entry fields
 - Commands currents on PSUs and polarity on switch box
 - If a device is not connected, the remaining ones are still commanded
 - Values beyond safe limits (indicated to the right of entry fields) are rejected
 - Device status and any errors are displayed in status display and console
- **"Power Down All" button:** "Panic button", sets currents on both PSUs to 0, deactivates outputs and sets switch box relay pins to inactive state
- **"Reinitialize" button:** Reruns program initialization
 - Reinitializes connection to PSUs and switch box Arduino
 - Press after (re)connecting a device to let program establish communications

To command a field vector or currents on the test bench:

1. Select needed input mode, using the drop-down menu
2. Enter desired values in entry fields
3. For magnetic fields, choose whether ambient field should be compensated by (un)tick the checkbox
4. Press "Execute" button, devices will now implement set values
5. Check console output to see if any errors occurred
6. Monitor behavior in status display and on devices
7. When finished, press "Power Down All" button to remove currents from the test bench

CSV Sequence Execution Mode

This mode is used to run timed sequences of magnetic fields. These have to be defined in a CSV file of the following form:

- *Column separator:* Semicolon (;)
- *Decimal:* Comma (,) or Period (.)
- *Line terminator:* Tested with Windows standard (`\r\n`), other options may work as well
- *Columns:* Time in seconds; X-axis, Y-axis and Z-axis flux density in Tesla

An example for the CSV file structure is given below:

```
Time (s);xField (T);yField (T);zField (T)
0,5;0,000015;0,000025;0,00002
1;0,0000155;0,0000245;0,0000205
```

When loading such a file, the application will check that there are no values exceeding the safe test bench limits in the sequence. If any are found, a warning message is shown and the limits displayed in the sequence graph. It is still possible to execute such a sequence, however the excessive values will not be commanded. For those time stamps, 0 A is set instead. The limits may be adjusted in the program settings page, but care should be taken to avoid equipment damage.

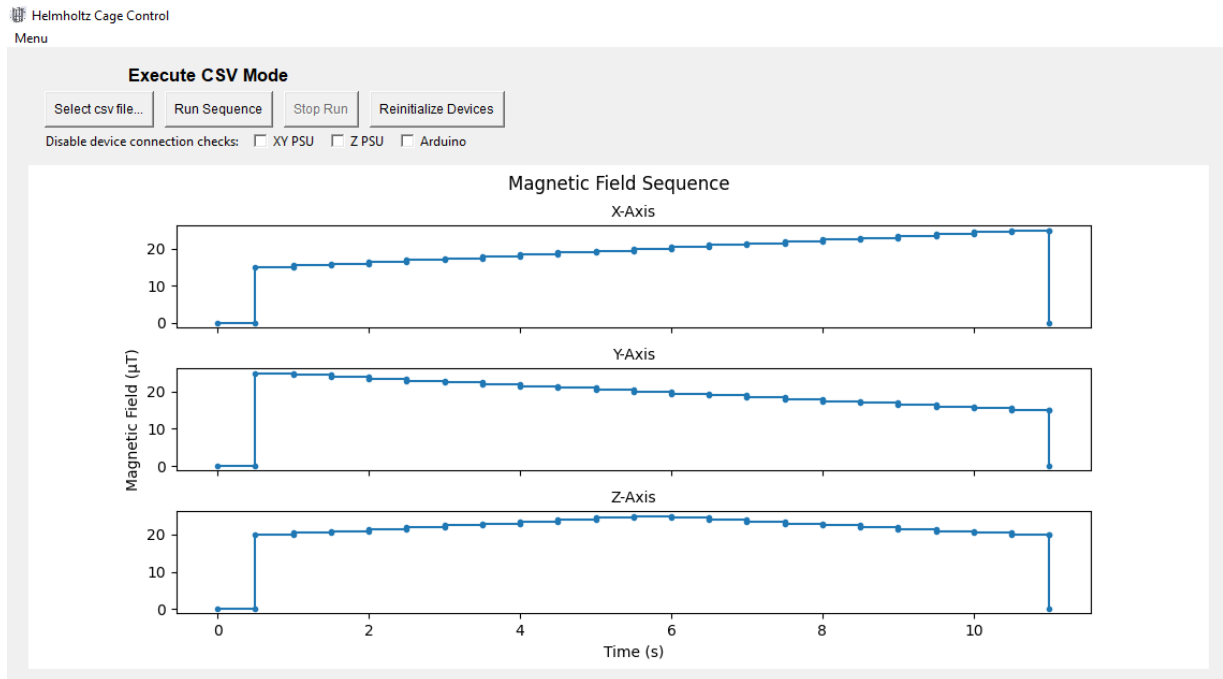


Figure 2.5: CSV sequence mode user interface

The UI layout is shown in Figure ??, its main elements are listed below.

- **“Select csv file...” button:** Opens a file dialog to let user choose a CSV file to execute
- **“Run Sequence” button** Starts executing the sequence from the chosen CSV file
- **“Stop Run” button:** Aborts sequence execution
- **“Reinitialize devices” button:** Reruns program initialization
 - Reinitializes connection to PSUs and switch box Arduino
 - Press after (re)connecting a device to let program establish communications
- **“Disable device connection checks” checkboxes:** Disable connection check for specific devices, allowing sequence execution with some equipment missing
- **Graph display:** Displays graph of field sequence, as it will be executed by the test bench

To execute a field sequence:

1. Ensure correct configuration of program and test bench in the settings page
2. Prepare sequence in CSV file with correct format (external program)
3. Press "Select csv file..." button, select prepared file and open
4. Check plots to confirm sequence was loaded correctly and no values exceed safe limits
5. If not all devices are used, check boxes of unused devices to disable connection check
6. Press "Run Sequence" button
7. Monitor execution in console, status display and on devices

Ambient Field and Coil Constant Calibration

The application offers calibration tools to determine the exact ambient field with the intention of cancelling it, as well for measuring the test bench's coil constants. These are both integrated into one application view, depicted in Figure ?? . All the calibration tools require access to the complete Helmholtz cage hardware, as well as a magnetometer. These must be connected before starting the test. It is important that the magnetometer is centered as well as possible to achieve accurate results. For these tests, where the magnetometer itself is not the Device Under Test (DUT), it is recommended to use the IRS's FGM3D reference magnetometer. Further, an adapter script (`fgm3d_adapter.py`) already exists for this sensor. For more details on writing adapter scripts and connecting magnetometers, please refer to Section ?? .

Ambient Field Calibration

Magnetometer state: **Not connected**

Field data:

X:	0.000	μT
Y:	0.000	μT
Z:	0.000	μT

Calibrate Ambient Field Calibrate Coil Constants

Progress:

Ambient Field Results

	X	Y	Z	
Ambient Field:	<input type="text"/>	<input type="text"/>	<input type="text"/>	A
Ambient Field:	<input type="text"/>	<input type="text"/>	<input type="text"/>	μT
Residual Field:	<input type="text"/>	<input type="text"/>	<input type="text"/>	μT

Save and apply Export raw CSV Copy to clipboard

Coil Constants

	X	Y	Z	
K:	<input type="text"/>	<input type="text"/>	<input type="text"/>	μT/A
K Std. Dev.:	<input type="text"/>	<input type="text"/>	<input type="text"/>	μT/A
Angles:	X-Y	Y-Z	X-Z	°

Save and apply Export raw CSV Copy to clipboard

Figure 2.6: Ambient field and coil constant calibration view.

After setting up the magnetometer, operation is simple: Click on the buttons to "Calibrate Ambient Field" or "Calibrate Coil Constants". This will cause the calibration procedure to start. The current status will be shown in the progress bar underneath. After completing, the results of the calibration procedure will be saved into the data fields on the right hand side of Figure ?? . At

this point, the buttons underneath the respective will become available. These two calibrations are primarily used to determine important application config parameters, namely the ambient field and coil constant as their names imply. To automatically apply the newly measured values, click “Save and apply”. The “Copy to clipboard” will put the results table as shown in the UI into the system clipboard, from which it can be pasted into software such as Microsoft Excel and LibreOffice Calc. If semi-raw experiment data is required to verify the program functioning or to apply custom algorithms, it can be exported with the “Export raw CSV” button.

Ambient Field Calibration Method This calibration uses a P-controller (implemented as PI-controller with $I = 0$) to attempt to reach zero as a set point for the magnetometer. Updates are preformed approximately every half second by default and the calibration executes for 45 seconds. The procedure consistently achieves zero offsets below 10 nT. The ambient field exported by this calibration has been processed by multiplying the current that was required with the current coil constants, thus it is strongly recommended to first preform the coil constant calibration if both are to be executed.

Coil Constant Calibration Method The coil constant calibration uses a linear distribution of currents in each axis individually to collect coil constant data. Each setpoint is held for 3 seconds before being measured, and by default 8 points distributed across -3 A to 3 A . To calculate the individual setpoints corresponding to each setpoint, first the field magnitude compared to the initial conditions is calculated, second the sign is estimated and reapplied, third the field is divided by the applied current. The final result is the average of all constants. In addition to the primary coil constant measurement, the maximum setpoints are also sampled once for each axis to calculate the angles between the coils. This is simply done by calculating the angle between the measured vectors.

To understand the calibration methods in all detail, it is recommended to look at the `calibration.py` source file.

Magnetometer Calibration

The helmholtz control software supports calibrating magnetometers. In preparation, a calibration of the ambient field using the reference magnetometer (FGM3D) should be conducted beforehand to achieve accurate results. Afterwards, the magnetometer must be replaced with the DUT. Since the software only supports one magnetometer at once, the reference magnetometer (meaning: its adapter script) should be disabled beforehand. Tip: To mount CubeSat magnetometers a purpose built PC104 holder can be found in accompaniment of the Helmholtz cage.

As the coordinate system of the Helmholtz Cage and the magnetometer may not match, there are two options that available. First, the calibration can be conducted as-is, and the calibration results will reflect the unexpected coordinate system by often yielding negative sensitivities and untypical sensor axis angles. These values are not invalid, but describe a sensor correction that would transform it into the Helmholtz coordinate system. With some work, the desired calibration parameters could be extracted mathematically by changing signs and adding or subtracting 90° increments. The second option would be to change the sensors orientation in software in the adapter script that is used. This yields a more expected set of calibration parameters, but these now do not describe the initial axes anymore. In this case, the parameters must be correlated back to their initial axes and angles must be negated if the axis was flipped. This common issue should be solved robustly in the control/calibration software in the future.

Figure 2.7: Magnetometer calibration view.

The procedure to start the magnetometer can be done as follows:

- Activate the FGMM3D box, start the *FGM3D TD Application* software and check if the device was recognised automatically (Port: FGM3D TD (000125). If the device was not recognized, click on *File -> Rescan devices*, else restart the computer.
- Click on *connect* and *start*. Telemetry should now be displayed in the application. Use the *Live Streaming* console and stream to **COM10**
- Start the *Helmholtz Control.exe* software and navigate to e.g. *Menu -> Ambient Filed Calibration* Start the *fgm3dcadapter.py* script that forwards the data live stream from *COM10* on the Sensys app to **COM11** of the Helmholtz application.
- The test bench should be ready to run e.g. the Magnetometer Calibrtation script.

The calibration method used (Zikmund [6]) relies on a simplified error model and the Helmholtz test bench. After either measuring the local geomagnetic field or cancelling it, the magnetometer-under-test runs through a sequence of magnetic fields generated by the Helmholtz coils, which supplies sufficient data to solve a system of equations containing the coefficients of interest. The non-linear system is constructed with the equation below on a per-axis basis, with one row for every sample.

$$B_{meas} = S (B_E \sin \alpha_E + B_x \cos \alpha \cos \beta + B_y \cos \alpha \sin \beta + B_z \sin \alpha)$$

The calibration procedure makes use of nearly equidistantly distributed vectors in all directions as test points. The number of these points, as well as the settle time before taking a measurement, can be set in the UI. Generally, a high number of points, such as larger than 8, is recommended to achieve both an accurate result and also to get useful residual data. The meaning of the individual calibration coefficients can be derived from Zikmund [6] or also the thesis accompanying the Helmholtz test bench [7].

After completing, the results of the calibration procedure will be saved into the data fields on the right hand side of Figure ???. This also enables the save buttons underneath. The “Copy

to clipboard” will put the results table as shown in the UI into the system clipboard, from which it can be pasted into software such as Microsoft Excel and LibreOffice Calc. If semi-raw experiment data is required to verify the program functioning or to apply custom algorithms, it can be exported with the “Export raw CSV” button.

Data Logging Configuration Page

The application has the ability to log test bench data to a CSV file. The data is temporarily stored internally and must be saved to an external file by user request. The logging output is highly customizable using the options shown in Figure ??.

An example of a log file is given in Appendix ?. Unless explicitly disabled, the first three columns are time stamps: date, system time and time since the start of logging in seconds. All dynamic values from the status display as well as the magnetometer status can be logged, see Table ?? for explanations. Each type of data is logged for all three axes by default, but this can also be specified. The units for numerical values are Volt, Ampere and Tesla.

There are two options as for when and how often data is logged. The first option logs a row of data in a regular time interval specified by the user. The second option logs, whenever a significant command is sent to the test bench, for example when a new field vector is commanded. Both options can be used simultaneously.

The logging configuration UI is shown in Figure ?. Its elements are listed below.

Figure 2.8: Data logging configuration page

- **“Start Logging” button:** Starts the logging of data, as configured in the other elements
- **“Stop Logging” button:** Stops the logging of data
- **“Write data to file” button:** Lets user save the logged data to a file
 - Opens dialogue to let user select a file path (chosen file name must be *.csv)

- Saves logged data to the selected file
- **"Clear logged data" button:** Deletes all data logged internally by the program, user will be asked if data should be saved to a file first
- **"Datapoints logged" counter:** Displays the current number of logged data rows
- **"Log in regular intervals" controls:** Enable checkbox to periodically log data, set interval (in seconds) in the entry field to the right
- **"Log whenever test bench is commanded" checkbox:** Enable, to log data on significant changes to the test bench (e.g. a new field vector is commanded)
- **Data selection checkboxes:** Select what data to log, explanations for most options are given in Table ???. In addition, the Log X/Y/Z-Axis Data checkboxes specify whether the selected logging variables will be included for the respective axis.

To collect and save log data:

1. Select when to log data (both options can be used simultaneously)
 - For regular logging, enable "Log in regular intervals" checkbox and set desired interval
 - For logging on test bench command, enable second checkbox
2. Select what data to log, using the lower checkboxes
3. Press "Start Logging" button
4. When finished, press "Stop Logging" button
5. Press "Write data to file" button
6. Choose file location and name (must be *.csv, e.g. my_log.csv), and press save
7. Press "Clear logged data" button (optional)

Settings Page

This is the main page for configuring the program. The settings can be stored to and loaded from *.ini configuration files.

The different program constants are set through a series of entry fields. All inputs must be numerical values (decimal points). Safe limits for each value are set inside the program. The constants and limits are listed in Table ?? . If a value exceeding those boundaries is entered, a warning message is displayed and the respective entry field marked in red. **It is not recommended to ignore these warnings, as incorrect settings can damage equipment on the test bench!**

The screenshot shows a 'Configuration Window' with the following elements:

- Buttons: 'Load config file...', 'Save current config', 'Save current config as...'.
- Serial Ports:
 - Arduino Serial Port: COM5 (e.g. COM10)
 - XY PSU Serial Port: COM7 (e.g. COM10)
 - Z PSU Serial Port: COM8 (e.g. COM10)
- Axis Settings (X-Axis, Y-Axis, Z-Axis):
 - Coil Constants: 38.83, 38.65, 37.3 (μT/A Field generated per applied current)
 - Ambient Field: 0.0, 0.0, 0.0 (μT Field to be compensated)
 - Resistances: 3.131, 3.107, 3.129 (Ω Resistance of coils + equipment)
 - Max. Current: 5.0, 5.0, 5.0 (A Max. allowed current)
 - Max. Voltage: 15.0, 15.0, 15.0 (V Max. allowed voltage, must not exceed 16V!)
 - Arduino Pins: 15, 16, 17 (- Should be 15, 16, 17)
- Buttons: 'Update and Reinitialize', 'Restore Defaults'.

Figure 2.9: Program settings page

Figure ?? shows a screenshot of the UI layout with the default settings. The main elements are listed below.

- **"Load config file..." button:** Imports an existing configuration file
 - Opens file dialogue for selecting configuration file
 - Loads settings from selected file
 - Checks if any settings exceed safe limits and, if so, displays warning messages
 - Reinitializes test bench devices with new settings
- **"Save current config" button:** Writes settings to the currently selected file and reinitializes test bench devices with new settings
- **"Save current config as..." button:** Writes settings to a new file
 - Opens dialogue to let user choose new file path and name (must be *.ini)
 - Reinitializes test bench devices with current settings

- **"Serial Port" entries:** Input COM ports for both PSUs and the switch box here
 - Use Windows device manager to find correct port names (connect devices separately to differentiate between devices)
 - Test bench X- and Y-axes need to be connected to channel 1 and 2 of one PSU, Z-axis to channel 1 of the other
- **Program constant entry fields:** Set constants here, details are listed in Table ??
- **"Update and Reinitialize" button:** Implements any changed settings in the program and on test bench devices, needs to be pressed for changes to take effect
- **"Restore Defaults" button:** Restores default settings

Table 2.2: Settable program constants

Entry	Limits	Explanation
Coil Constants:	0 to 50 $\frac{\mu\text{T}}{\text{A}}$	Magnetic field generated per applied current <ul style="list-style-type: none"> • Used to calculate current needed to achieve desired field • Must be measured and tuned before test campaigns
Ambient Field:	-200 to 200 μT	Background magnetic field in the measurement area <ul style="list-style-type: none"> • Subtracted from desired field to compensate ambient field • Must be measured and tuned before test campaigns
Resistances:	1 to 5 Ω	Electrical resistance of each axis, measure from PSU connectors
Max. Current:	0 to 6 A	Current limit for each axis <ul style="list-style-type: none"> • Program will block commanding of higher values on PSU • Maximum safe permanent current: $I_{max} = 5.5 \text{ A}$ [5]
Max. Voltage:	0 to 16 V	Voltage limit for each axis <ul style="list-style-type: none"> • Program will block commanding of higher values on PSU • Maximum safe voltage, limited by diodes inside switch box: $U_{max} = 16 \text{ V}$
Arduino Pins:	15,16,17	Output pins on switch box Arduino for inverting polarity on each axis (hard wired inside switch box, should not need to be changed)

2.2.3 Hardware Connections and Program Setup

1. Connect hardware
 - a) Connect switch box to test bench main cable bundle (connector on the back)
 - b) Connect switch box to PSUs: **X-axis to PSU 1 channel 1, Y-axis to PSU 1 channel 2, Z-axis to PSU 2 channel 1**
 - c) Connect switch box power supply (12 V DC)
 - d) Connect switch box USB port to PC
 - e) Connect PSU USB ports to PC
2. Configure interfaces to hardware
 - a) Start program (run "Helmholtz Cage Control.exe")
 - b) Go to settings page (Menu→Settings...)
 - If program has not been configured before:
 - c) Use Windows device manager to find correct serial COM ports for PSUs and Arduino (connect /disconnect in turn to differentiate between devices)
 - d) Enter COM port names in application (switch box should be found automatically)
 - e) Press "Update and Reinitialize" button
 - If a previous configuration exists:
 - c) Press "Load config file..." button
 - d) Select configuration file (e.g. "myconfig.ini") and open
 - e) Check that the settings were loaded correctly and no values violate the safe limits (fields highlighted in red)
 - f) Check console print to see if all devices were found, otherwise check physical connections and COM port settings
3. If using a magnetometer, the listening port is open and the adapter script can now be started
4. Test configuration
 - a) Go to manual mode (Menu→Static Manual Input)
 - b) Switch input mode to "Current" (see Section ??)
 - c) Set different currents and check device response:
 - Current should be activated on correct PSU channel
 - For negative currents, corresponding status LED on switch box should light up and relay actuation be audible as clicking sound
 - d) If using a magnetometer: Check the device output in either of the calibration-views
5. Go back to the settings page (Menu→Settings...)
6. Change program constants as needed (e.g. enter measured ambient field), see Section ??

7. Save configuration:

- Press "Save current config" button to update the current configuration file
- Press "Save current config as..." button to save to a new configuration file, set new file name in file dialogue (e.g. "myconfig.ini")

2.2.4 TCP Remote Control

The Helmholtz Control Software offers a TCP automation interface that is opened upon application start-up. To find out which port is being listened on, look for the corresponding information in the application console, but typically, it will be port 6677. The commands that are accepted by the TCP interface are documented in Table ?? and also in the `socket_control.py` source file. All field commands will implicitly preform the usual safety checks to ensure safe operation. The commands that are shown must all be terminated with a single `\n` (newline) char. Commands may be split across multiple packets if desired. Important Note: Before useful commands can be sent, `declare_api_version` must be called.

The `tools` folder in the application git repository contains an example implementation (`fgm3d_adapter.py`) of a magnetometer interface using the TCP socket.

Table 2.3: TCP Remote Control Commands

Command	Description
<code>set_raw_field [X] [Y] [Z]</code>	Returns: 0 or 1 for success. Accepts decimal point formatted floats, with or without scientific notation. The <code>float()</code> cast must understand it. The field units are Tesla. This causes an additional field of the given strength to be generated, without regard for the pre-existing geomagnetic/external fields.
<code>set_compensated_field [X] [Y] [Z]</code>	Returns: 0 or 1 for success. Accepts decimal point formatted floats, with or without scientific notation. The <code>float()</code> cast must understand it. The field units are Tesla. This causes a field of exactly the given magnitude to be generated by compensating external factors such as the geomagnetic field.
<code>set_coil_currents [X] [Y] [Z]</code>	Returns: 0 or 1 for success. Accepts decimal point formatted floats, with or without scientific notation. The <code>float()</code> cast must understand it. The field units are Ampere. This establishes the requested current in the individual coils.
<code>magnetometer_field [X] [Y] [Z]</code>	Returns: 1. Accepts decimal point formatted floats, with or without scientific notation. The <code>float()</code> cast must understand it. The field units are Tesla. Sets the state of a virtual magnetometer object which mirrors a physical sensor by means of this command.
<code>get_api_version</code>	Returns: a string uniquely identifying each API version. This function can be called before <code>declare_api_version</code> .

<code>declare_api_version</code> [version]	Returns: 0 or 1. Declare the API version the client application was programmed for. It must be compatible with the current API version. This prevents unexpected behavior by forcing programmers to specify which API they are expecting. This function must be called before sending HW commands.
--	--

Bibliography

- [1] SPRÖSSIG, Sören. *Python PS2000B Library* [online]. [visited on 2020-11-18]. Available from: <https://github.com/ssproessig/Python-PS2000B>.
- [2] *Arduino Python 3 Command API* [online]. [visited on 2020-12-09]. Available from: <https://github.com/mkals/Arduino-Python3-Command-API>.
- [3] KINSLEY, Harrison. *Object Oriented Programming Crash Course with Tkinter* [online]. [visited on 2021-01-19]. Available from: <https://pythonprogramming.net/object-oriented-programming-crash-course-tkinter/>.
- [4] VOLLEBREGT, Brent. *Auto Py to Exe* [online]. [visited on 2020-03-02]. Available from: <https://github.com/brentvollebregt/auto-py-to-exe>.
- [5] BLESSING, Steffen. *Design of a CubeSat Magnetic Field Cage for the Verification of CubeSats Attitude Control Systems*. Stuttgart, 2020. Institute for Space Systems (IRS), University of Stuttgart.
- [6] ZIKMUND, A.; JANOSEK, Michal. Calibration procedure for triaxial magnetometers without a compensating system or moving parts. In: 2014, pp. 473–476. ISBN 9781467363860. Available from DOI: 10.1109/I2MTC.2014.6860790.
- [7] TEICHRÖB, Leon. *Mapping and Calibration of a Helmholtz Magnetic Field Cage and Test of the EIVE Attitude Control System*. Stuttgart, 2021. Institute for Space Systems (IRS), University of Stuttgart.

A Example Program Auxiliary Files

Example Configuration File

```
[X-Axis]
coil_const = 3.883e-05
ambient_field = 0.0
resistance = 3.131
max_volts = 15.0
max_amps = 5.0
relay_pin = 15
```

```
[Y-Axis]
coil_const = 3.865e-05
ambient_field = 0.0
resistance = 3.107
max_volts = 15.0
max_amps = 5.0
relay_pin = 16
```

```
[Z-Axis]
coil_const = 3.73e-05
ambient_field = 0.0
resistance = 3.129
max_volts = 15.0
max_amps = 5.0
relay_pin = 17
```

```
[Supplies]
supply_model = ql355tp
arduino_port = COM1
xy_port = COM2
z_port = COM3
```

Example Log File

```
Date;Time;t (s);X Target Field;Y Target Field;Z Target Field
2021-03-08;16:18:34;583986;0,0;0,0;0,0,0,0
2021-03-08;16:18:39;595109;5,011123;0,0;0,0,0,0
2021-03-08;16:18:44;217410;9,633424;-4,499999996e-05;-4,999999996e-05;0,0
2021-03-08;16:18:44;597378;10,013392;-4,499999996e-05;-4,999999996e-05;0,0
2021-03-08;16:18:49;604153;15,020167;-4,499999996e-05;-4,999999996e-05;0,0
2021-03-08;16:18:54;619011;20,035025;-4,499999996e-05;-4,999999996e-05;0,0
2021-03-08;16:18:57;713520;23,129534;-4,499999996e-05;4,999999996e-05;0,0
2021-03-08;16:18:59;635428;25,051442;-4,499999996e-05;4,999999996e-05;0,0
2021-03-08;16:19:00;759706;26,17572;0,0;0,0,0,0
```