Calibration of 3D-magnetometer in Rust for embedded applications

Mini-project in Sensors & Systems

Peter Krull (20194717)
pkrull19@student.aau.dk

Submitted
27-05-2023

AALBORG UNIVERSITET

The Technical Faculty of IT and Design

Frederik Bajers Vej 7
9220 Aalborg Øst, Denmark
http://www.es.aau.dk

Contents

Project description 1

Sensors and Systems at ES8
Spring 2023

Title	Magnetometer calibration in Rust
Participants	Peter Krull (20194717)
Description	This code implements an algorithm for calibrating a magnetometer, by determining the offset and scale values that turn the offset ellipsoid of measurements into the unit sphere centered in the origin. This is done using least squares approximation on a set of N collected measurements. New measurements are added to the collection based on the criterion that the mean distance to the k nearest neighbors should be larger than the smallest mean distance in the set. If the condition is met, the new measurements replaces the measurements with the lowest mean distance. This ensures that the N measurements are well distributed on the sphere. The least squares method can be called once the total mean distance exceeds some user-provided limit. If the calculation does not fail (if data is extremely noise, causing ill-conditioning), the offset and scale values are safely returned to the user, wrapped in an `Option<>` type
Target user	Someone who intends to get useful data out of a magnetometer. It can for example be implemented to work as a basic compass, or in a sensor fusion algorithm, to give AHRS information to a flight controller or similar. It should also be possible to modify the algorithm to do continuous online calibration, with some additional low-pass filtering.
IDE	Visual Studio Code with Rust-Analyzer
Sensors	Magnetometer inside ICM20948 (AK09916)
Lectures	Lecture 4 (magnetometer),
Code	`https://github.com/peterkrull/mag-calibrator-rs`

Sensor description 2

The sensor used for the project is the ICM20948 9-DOF inertial measurement unit (IMU). The 9-DOF consist of a 3-axis accelerometer, 3-axis gyroscope and a 3-axis magnetometer. The board used for this is manufactured by SparkFun, and provides access to the I2C interface of the ICM20948.

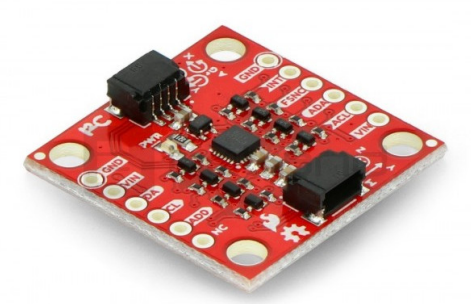

Figure 2.1. SparkFun ICM20948 module. Credit: SparkFun

The ICM20948 is manufactured as a multi-chip module (MCM) which in this case means the magnetometer of the model AK09916 is on a separate die inside the chip package. Special consideration is therefore necessary when implementing drivers for the ICM20948 since the AK09916 magnetometer, connects to the main-die inside the ICM20948 via an internal I^2C interface. This means that any configuration of, or reading from, the AK09916 module requires first configuring the ICM20948 correctly to either act as a pass-through or to buffer readings. These considerations are not documented here, but the driver used was developed by the author of this project and can be found at the link below:

`https://github.com/peterkrull/icm20948-async/`

For this project the magnetometer is the main focus, meaning that the accelerometer and gyroscope of the ICM20948 are not going to be used.

2.1 Workings of magnetometer

The `AK09916` module uses 3 hall-effect magnetic sensors, each orthogonal to the others. (**SSEQ A1**) The hall effect is a phenomenon where if a magnetic field passes through a magnetic plate, and a current is running through the plate perpendicular to the field, the magnetic field will cause the current to deflect, which charges the plate, and causes a voltage potential to appear perpendicular to both the current and magnetic field. This is illustrated in figure 2.2

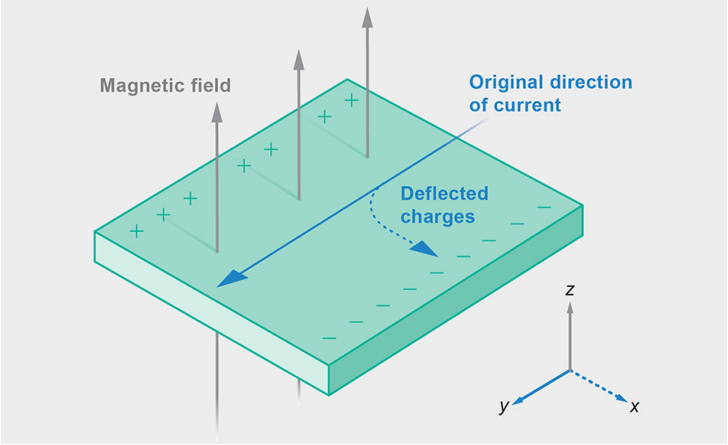

Figure 2.2. Hall effect illustration. Credit: Lucy Reading-Ikkanda / newsakmi.com

(**SSEQ A2**) Measuring the voltage across the plate, for a given current, indicates the magnitude (strength) of the magnetic field passing through the plate. By including 3 of these hall effect sensors, each perpendicular to each other, allows for measuring both the direction and magnitude of the magnetic field. Such an arrangement means the sensor can be referred to as a 3D-magnetometer, which is the case for the `AK09916`.

(**SSEQ A3**) While physical property being measured is a voltage, the desired signal is the magnetic field strength. (**SSEQ B4**) The relation between measured hall-voltage V_H and the field strength B_z for some current I_x can typically be considered linear given the formula 2.1. The constant K is dependent of the physical properties of the plate.

$$V_H = \frac{I_x B_z}{K} \tag{2.1}$$

(**SSEQ B1**) For the `AK09916` the measurements are delivered to the user at a rate up to 100 Hz as signed 16-bit integers with a full-scale resolution of $\pm 4\,900\,\mu T$, which results in a per-bit resolution of $0.15\,\mu T/LSB$. (**SSEQ B4**) The measurement can therefore be converted to the correct unit (µT) by type-casting to a float, and multiplying with the per-bit resolution $0.15\,\mu T/LSB$.

(**SSEQ C2 & C3**) The datasheet for the `AK09916` nor the `ICM20948` makes any mention of noise characteristics, accuracy or precision for the magnetometer. It has not been possible to setup an environment where testing the accuracy of precision was possible, and the noise characteristics are not measured for this project.

Calibration method 3

Commonly, the purpose of the 3D-magnetometer is to sense the direction and magnitude of the earths magnetic field lines, usually to use it as a heading reference. It is however easily disturbed by external factors such as the position of external ferrous metals or magnetic sources. Even mounting the sensor to a PCB, or the PCB to other hardware will disturb the readings. It is therefore necessary to calibrate the magnetometer, preferably at every power-on.

For a well-calibrated magnetic sensor, a measurement consists of a 3D-vector pointing in the same direction as the geomagnetic field lines, with a magnitude equal to the strength of the earths magnetic field. For a set of samples (measurements), where the orientation of the sensor is randomized, one would expect to see sphere centered in the origin, with the radius being the strength of the earths magnetic field at that location, typically 25 µT to 65 µT. However, the disturbed sensor is likely to measure an ellipsoid not centered in the origin.

3.1 Choosing a method

A method to calibration the magnetometer, such that the ellipsoid is transformed into a sphere centered in the origin, is to do a least-squares fit of a set of samples to the unit-sphere. This will not preserve the magnitude of the magnetic field, but for AHRS purposes, which this library was developed for, the magnitude is not important. The transformation from a set of noisy uncalibrated measurements to calibrated measurements will look like figure 3.1.

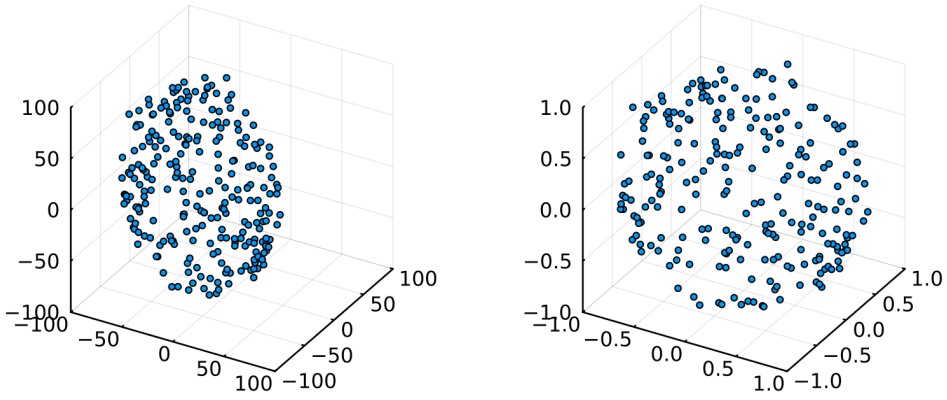

Figure 3.1. Comparison of uncalibrated (left) and calibrated (right) 3D-magnetometer.

The disturbances can be classified as either a translation and warping along the axes, caused by so-called hard-iron disturbances, and covariant (off-axis) warping, caused by soft-iron distur-

bances. Most importantly for AHRS is getting rid of the hard-iron effects, such that the point cloud is centered in the origin and mostly spherical. It is therefore chosen to go forward with such an approach.

The algorithm for the least squares calibration will be based on the Matlab code posted by user `geometrikal` on stackexchange: `https://electronics.stackexchange.com`. The code is shown below. This code was used to generate the calibrated plot (right) on figure 3.1 based on the uncalibrated samples (left), to verify its overall functionality.

```
H = [mag(:,1), mag(:,2), mag(:,3), - mag(:,2).^2, - mag(:,3).^2, ones(size(mag(:,1)))];
w = mag(:,1).^2;
X = (H'*H)\H'*w;
offX = X(1)/2;
offY = X(2)/(2*X(4));
offZ = X(3)/(2*X(5));
temp = X(6) + offX^2 + X(4)*offY^2 + X(5)*offZ^2;
scaleX = sqrt(temp);
scaleY = sqrt(temp / X(4));
scaleZ= sqrt(temp / X(5));
```

Hardware description 4

The library developed for this project is made to run on highly embedded systems, which is why a dual-core Cortex-M0+ RP2040 microprocessor is used as a target for development. The `ICM20948` is connected using the first I^2C bus running in fast-mode at 400 kHz and is configured to use pins 10 (SDA) and 11 (SCL) on the RP2040, with connections to 3V3 and GND as well, as shown in table 4.1.

IMU	SDA	SCL	GND	3V3
MCU	Pin 10	Pin 11	GND	3V3

Table 4.1. Connections between `ICM20948` (IMU) and RP2040 (MCU)

A photography of the `ICM20948` and micro controller mounted on a drone is shown in figure 4.1

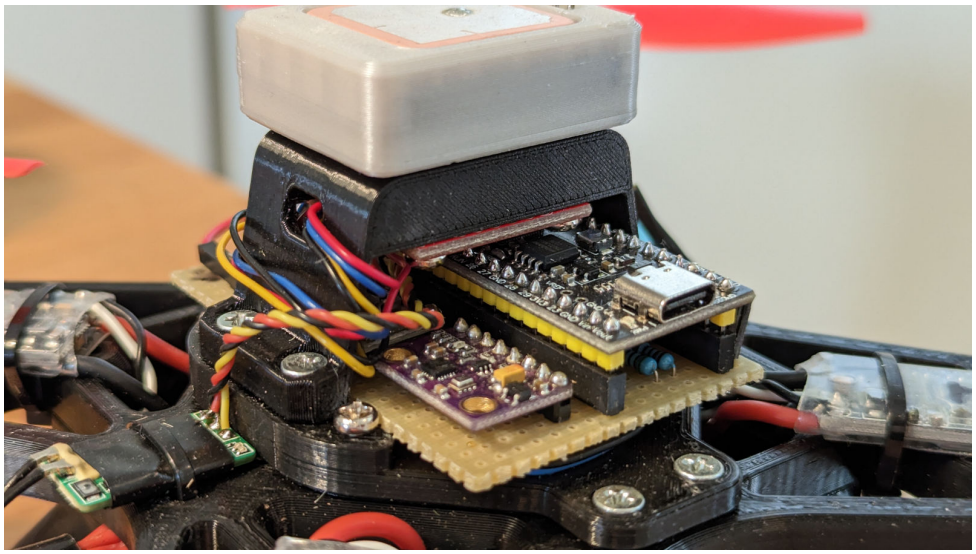

Figure 4.1. Photo of `ICM20948` (red PCB on the bottom of the black plastic piece) and micro controller mounted to drone.

The remaining hardware on the drone is not relevant for this project. (**SSEQ D1**) However, the hard-mounting of all hardware, especially the magnetometer, is good for keeping the magnetic disturbances mostly constant, which is essential to get a useful calibration.

Code description 5

The Rust code for this project can be found at the url:
`https://github.com/peterkrull/mag-calibrator-rs`

The purpose of this library is to perform 3D-magnetometer calibration on embedded devices such as memory and compute constrained micro controllers like the RP2040. In order to perform a calibration without having to dynamically allocate memory for an unknown number of samples, a constant-size buffer in necessary. The library should therefore have two main functionalities, as listed.

1. Collect a set of N samples which are highly unique, replacing the least unique sample every time a more useful sample is measured.
2. Do least-squares fit of collected samples to the unit sphere while lowering the risk of ill-conditioning.

5.0.1 External libraries

To do the the calibration described in section 3 requires matrix operations, including inversion and transposition. The Rust community has developed a freely available library (known as a Crate) called `nalgebra` which provides common matrix functionality with compile-time checks on dimensions. This library also supports compiling to embedded targets, with statically sized matrices. For this project `nalgebra` is used to do matrix operations.

5.1 System overview

Since it is necessary to collect and store a set of N measurements, means that it makes sense to implement the library in an object oriented fashion. In the Rust language, there are no traditional classes. To obtain similar functionality, Rust allows for data types, such as structs or enums, to have *implementations* using the `impl` keyword, which attaches methods to them.

This library is based on a struct named `MagCalibrator` to store various internal values. The struct itself is made public, which exposes it to external users, but none of its contents are made public. The struct uses a generic constant N to define the size of the buffer matrix, which is of the statically sized `nalgebra` type `SMatix`. The generic constant allows for the size of the entire `MagCalibrator` struct to be determined at compile-time, which means no dynamic allocation is required.

```rust
pub struct MagCalibrator<const N: usize> {
    matrix: SMatrix<f32, N, 6>,
    matrix_filled: usize,
    mean_distance: f32,
    pre_scaler: f32,
    k: usize,
}
```

The `MagCalibrator` struct contains the following entries.

1. `matrix` : $N \times 6$ matrix buffer to store samples and for least-squares fitting.
2. `matrix_filled` : Keeps track of how many samples have been collected so far.
3. `mean_distance` : Mean distance between all point to their `k`-nearest neighbors.
4. `pre_scaler` : Scalar applied to all samples to pre-conditioning for least-squares fit.
5. `k` : Number of neighboring points to evaluate samples against.

Apart from the object constructor, which is realized using a builder pattern, the public methods (exposed to the user of the library) are the following.

1. `get_mean_distance` : Returns the `mean_distance` entry from the object struct.
2. `evaluate_sample_vec` : Evaluates if a sample is should be added to `matrix` buffer.
3. `perform_calibration` : Does least squares fit of data in `matrix` buffer to get calibration.

Additionally, there are some private methods, that are used internally in the library. These are the following.

1. `mean_distance_from_single` : Calculate mean distance to `k`-nearest neighbors in the `matrix` buffer for a single sample.
2. `mean_distance_from_all` : Calculate mean distance to `k`-nearest neighbors for all samples samples in the `matrix` buffer.
3. `lowest_mean_distance_by_index` : Returns row-index of the sample with lowest mean distance to `k`-nearest neighbors in the `matrix` buffer.
4. `add_sample_at` : Inserts a sample at certain row in the `matrix` buffer with `pre_scalar` applied.

For the end-user, the intended use case may look like the following.

```rust
// Object for magnetometer/imu (bring your own)
let imu = ImuDriver::new(i2c);

// Create a calibration object with 20-sample buffer
let mag_calibrator = MagCalibrator::<20>::new()
    .pre_scaler(50.) // Approx geomagnetic field strength in DK

// Feed new samples to the evaluator while mean distance is low
while mag_calibrator.get_mean_distance() < 0.035 {
    let mag_reading: Vector3<f32> = imu.read_mag().into();
    mag_calibrator.evaluate_sample_vec(mag_reading);

    // Optional async (or blocking) delay
    Timer::after(Duration::from_ms(100)).await;
}

// Get the final calibrated values with pattern matching
if let Some((offset,scale)) = mag_calibrator.get_calibration() {
    // Do stuff with the calibration values
}
```

5.1.1 Collection of N unique samples

In order to determine the "uniqueness" of a sample, a distance to the k-nearest neighbors approach is used. The high-level logic behind this approach is illustrated in figure 5.1. This logic is defined in the `evaluate_sample_vec` method.

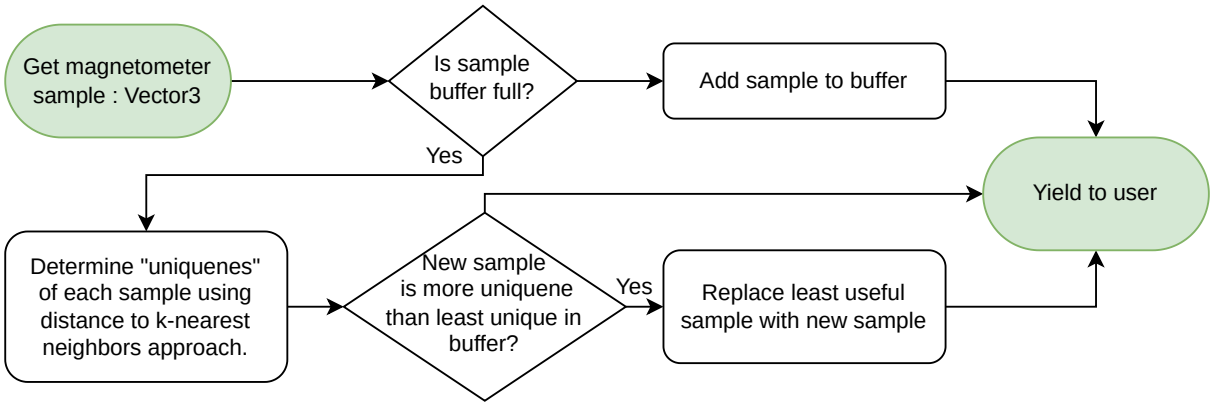

Figure 5.1. Flow diagram of function to evalueate sample.

The rust-code for `evaluate_sample_vec` is the following.

```
/// Add a sample if it is deemed more useful than the least useful sample
pub fn evaluate_sample_vec(&mut self, x: Vector3<f32>) {
    // Check if buffer is not yet "initialized" with real measurements
    if self.matrix_filled < N {
        self.add_sample_at(self.matrix_filled, x);
        self.matrix_filled += 1;
    }
    // Otherwise check which sample may be best to replace
    else {
        let (low_index, low_mean_dist) = self.lowest_mean_distance_by_index();
        let sample_mean_dist = self.mean_distance_from_single(x.transpose());
        if low_mean_dist < sample_mean_dist {
            self.add_sample_at(low_index, x);
        }
    }
}
```

Here a couple of private methods are used. Both of these are not marked as public, namely `lowest_mean_distance_by_index` and `mean_distance_from_single`. These are not described in detail, since they contain no logic and purely implement math. These other methods make heavy use of iterators and closures, which are a feature of Rust that can in some cases be simpler to work with than traditional for-loops, especially for somewhat complicated operations.

5.1.2 Performing the calibration

When the user has called the `evaluate_sample_vec` method enough times to obtain a well-distributed set of measurements, they should call the method `perform_calibration` to perform the calibration according to the description in chapter 3. This method returns an `Option<>` type, which means that the calibration can fail, and that the user of the library must take this into consideration in their code. The calibration may fail if it is not possible to compute the pseudo inverse, or if some of the calibrated offsets are not finite values, which can be the case for very noisy measurements, or if the measurements are not well distributed.

The flow diagram implementing the calibration is shown in figure 5.2.

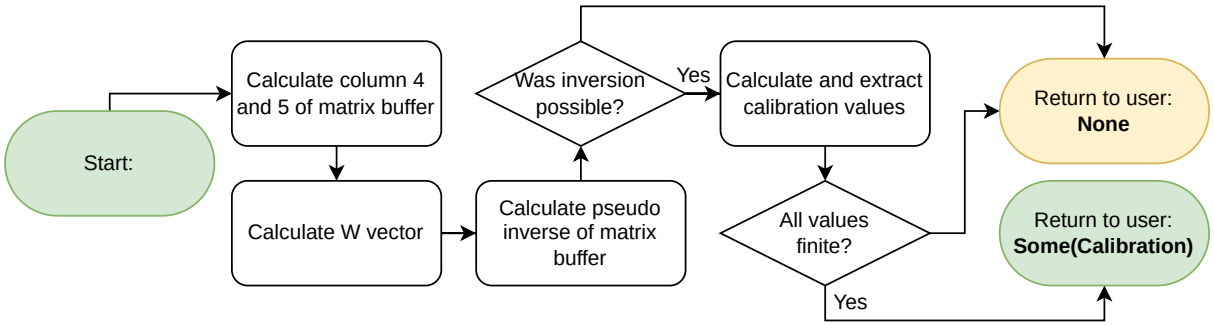

Figure 5.2. Flow diagram of function to perform calibration calculation.

The code which implements this functionality is the following.

```
/// Try to calculate calibration offset and scale values.  Returns None if
/// it was not possible to calculate the pseudo inverse, or if some of the
/// parameters are ‘NaN‘.  The tuple contains (offset , scale)
pub fn perform_calibration(&mut self) -> Option<([f32; 3], [f32; 3])> {
    // Calculate column 4 and 5 of H matrix
    self.matrix.row_iter_mut().for_each(|mut mag| {
        mag[3] = -mag[1] * mag[1];
        mag[4] = -mag[2] * mag[2];
    });

    // Calculate W vector
    let mut w: SMatrix<f32, N, 1> = SMatrix::from_element(0.0);
    self.matrix
        .row_iter()
        .enumerate()
        .for_each(|(i, row)| w[i] = row[0] * row[0]);

    // Perform least squares using pseudo inverse
    let x =
        (self.matrix.transpose() * self.matrix).try_inverse()? * self.matrix.transpose() * w;

    // Calculate offsets and scale factors
    let off = [x[0] / 2., x[1] / (2. * x[3]), x[2] / (2. * x[4])];
    let temp = x[5] + (off[0] * off[0]) + x[3] * (off[1] * off[1]) + x[4] * (off[2] * off[2]);
    let scale = [temp.sqrt(), (temp / x[3]).sqrt(), (temp / x[4]).sqrt()];

    // Check that off and scale vectors contain valid values
    for x in off.iter().chain(scale.iter()) {
        if !x.is_finite() {
            return None;
        }
    }

    Some((off,scale))
}
```

Test and discussion 6

6.1 Test

In order to verify the functionality of the code, a set of 2500 raw measurements are taken from the sensor where the sensor is moved around randomly, such that there is good coverage. For easy of testing, these measurements are sent to the computer via a serial connection, where they are saved in a CSV-formatted file. This makes it possible to do all testing of the Rust code offline, however the code has been tested to work on the RP2040 micro controller as well.

The `MagCalibrator` struct is defined as having a buffer size of 50 samples. All 2500 measurements are one-by-one evaluated using the `evaluate_sample_vec` method. At last, the `perform_calibration` method is called to get the calibration scale and offset values. These calibration values are then be applied to the samples in the buffer, to visually verify the result. Both the raw data and the calibrated data is shown in figure 6.1

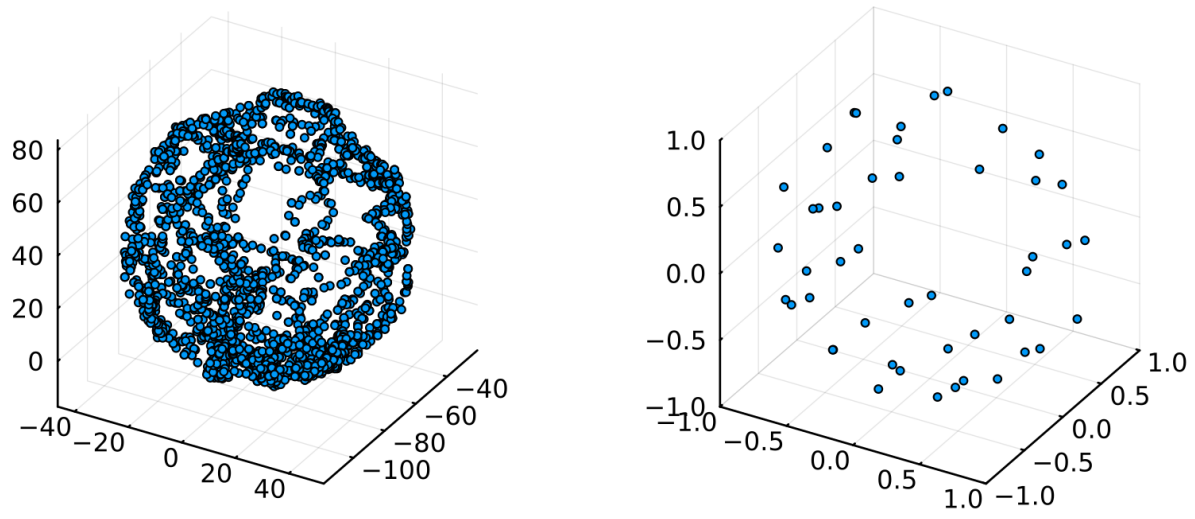

Figure 6.1. Plots of set of 2500 uncalibrated measurements (left) and set of 50 calibrated measurements (right)

It can be seen from the figure (through more clear on an interactive 3D-plot) that the points on the right plot, while much fewer than the left plot, are still well distributed across the surface of the sphere. The calibration performed on this small number of points also results in the sphere of measurements having been turned into the unit sphere. These results show the algorithms, both for collecting the measurements and for doing the fitting, function as intended.

6.2 Discussion

Currently, to obtain good results using 32-bit floating point values, the user is expected to supply a pre-scalar for the best fitting performance, which may require some trial and error to get right for a given sensor. It would be beneficial for a general purpose library to automatically determine a good pre-scalar value. The method for this is being investigated, but nothing has been implemented yet.

It is currently necessary for the user to call the `get_mean_distance()` method and decide themselves when to call the `perform_calibration()` method to get the calibration values. It would be preferred if the `perform_calibration()` would simply refuse to complete the calibration of the collected points are not adequately unique. This would require an additional algorithm, which is being investigated, but nothing has been implemented yet.

If the environment around the sensors is changing, like it may be for an aircraft flying to different locations, it would be beneficial to do continuous "on-the-fly" calibration. This is not currently supported, but some small changes to the library and some filtering could allow for this.

The calibration process normalizes the length of all measurements. As described, this is not a problem for AHRS purposes, but may be a problem for other use cases. A different approach could be taken where the strength of the measured field is preserved after calibration. For example by scaling the calibration values with the expected field strength at the current location.

No special consideration is taken to outliers caused by eg. noise. Due to how points are deemed "unique" by their distance to other points, outliers are more likely to persist in the measurement buffer. It would be possible to do a post-fit removal of outlier points, then re-fit to the remaining points, or just continue collecting more new points.