Cairo University - Faculty Of Engineering
Computer Engineering Department
Embedded Systems - Spring 2025

Calibration and Denoising of MPU6050 Accelerometer Sensor

Assignment 1

Name	ID
Asmaa Mohamed	9220154
Ahmed Hamdy	9220032
Ahmed Hamed	9220027
Somia Saad	9202666
Shehab Khaled	9220393

Delivered to
Dr. Mostafa Ghoneem

Contents

List of Figures

1 Abstract

This report presents a method for calibrating and denoising MPU6050 accelerometer measurements. The proposed approach corrects systematic errors using Ferraris' method, which requires only six fixed orientations. Experimental results demonstrate an improvement in measurement accuracy, making the sensor more reliable for robotics applications.

2 Introduction

Inertial Measurement Units (IMUs) integrate an accelerometer, a gyroscope, and sometimes a magnetometer within a single sensor package, leveraging MEMS (micro-electro-mechanical systems) technology. These sensors are widely used in navigation systems, attitude estimation, and consumer electronics such as smartphones. Their affordability makes them attractive for various applications, but this comes at the cost of high cross-sensitivity, significant offset biases, and drifts.
Ideally, an IMU's tri-axial sensor clusters should share the same three orthogonal sensitivity axes with fixed and known scale factors, ensuring accurate conversion of digital measurements into physical quantities like acceleration and angular velocity. However, low-cost MEMS-based IMUs, such as the MPU6050, often suffer from scaling errors, sensor axis misalignment, cross-axis sensitivities, and non-zero biases. These inaccuracies can degrade measurement reliability in motion tracking and control applications.
IMU calibration aims to estimate and correct these errors, improving measurement accuracy. While professional calibration tools exist, they are often costly. In this report, we explore Ferraris' calibration method, a cost-effective approach that corrects systematic errors using only six fixed orientations, enhancing the reliability of low-cost IMUs.

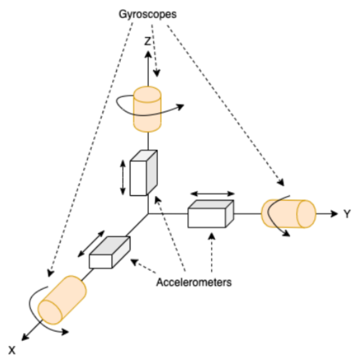


Figure 1: Simplified model of an IMU

3 Related Work

The calibration of Inertial Measurement Units (IMUs) has been a crucial area of research due to the impact it has on the accuracy of measurements in various applications such as navigation systems, robotics, and aerospace. Traditionally, IMU calibration has been performed using mechanical platforms that move the sensor through a series of precisely controlled orientations with known rotational velocities. These methods often involve using a robotic manipulator or other specialized hardware [1], [2], [3]. Although these platforms can provide high accuracy, their cost is typically very high, making them impractical for many applications, especially when calibration needs to be repeated multiple times.

In recent years, there has been a growing interest in alternative IMU calibration methods that do not rely on costly mechanical platforms. One such method is the multi-position approach, where the sensor is adjusted at various angles in a controlled environment. The multi-position approach typically involves adjusting the sensor at different orientations and recording the sensor's output at each position. From this data, errors such as misalignments, scaling factors, and biases can be computed.

A robust and easy-to-implement method for IMU calibration without the need for external equipment was proposed in [4]. This approach incorporates an error model that accounts for axes misalignment, scaling factors, and biases, significantly reducing the need for specialized calibration setups. The proposed model is particularly attractive for applications where cost is a concern, and external calibration equipment is unavailable.

Another notable work in this area is Ferraris (1994) [5], whose method builds on similar principles by proposing an error model that compensates for the common issues encountered during IMU calibration, including misalignments and biases. This paper provides a foundation for our approach and serves as the basis for the IMU calibration method we intend to implement in our report.

4 Accelerometer Model

The accelerometer model consists of two key components: a single-axis model for individual sensors and a three-axis model that accounts for interactions between multiple sensors in an inertial measurement unit (IMU).

4.1 Single-Axis Model

For an individual accelerometer, the output u_a along its sensitivity axis can be modeled as:

$$u_a = k_a a_s + b_a \tag{1}$$

where:

- u_a is the sensor output (in volts or digital units)
- a_s is the true acceleration along the sensor's sensitivity axis (in m/s^2)
- k_a is the scale factor (in units of output/acceleration)
- b_a is the bias (output when acceleration is zero)

This linear model requires calibration of two parameters per axis: the scale factor k_a and bias b_a.

4.2 Three-Axis Model

In practical IMUs containing three orthogonally-mounted accelerometers, additional considerations are necessary due to manufacturing imperfections:

$$\mathbf{u}_a = \mathbf{K}_a \mathbf{R}_a \mathbf{a} + \mathbf{b}_a \tag{2}$$

where:

- $\mathbf{u}_a = [u_{a,\sigma}\ u_{a,\mu}\ u_{a,\tau}]^\top$ is the output vector
- $\mathbf{a} = [a_\sigma\ a_\mu\ a_\tau]^\top$ is the true acceleration vector in the reference frame $\Sigma = (\sigma, \mu, \tau)$
- $\mathbf{K}_a$ is the diagonal scale factor matrix:

$$\mathbf{K}_a = \begin{bmatrix} k_{a,\sigma} & 0 & 0 \\ 0 & k_{a,\mu} & 0 \\ 0 & 0 & k_{a,\tau} \end{bmatrix}$$

- $\mathbf{R}_a$ is the orientation matrix accounting for axes misalignment:

$$\mathbf{R}_a = \begin{bmatrix} r_{a,\sigma\sigma} & r_{a,\sigma\mu} & r_{a,\sigma\tau} \\ r_{a,\mu\sigma} & r_{a,\mu\mu} & r_{a,\mu\tau} \\ r_{a,\tau\sigma} & r_{a,\tau\mu} & r_{a,\tau\tau} \end{bmatrix}$$

- $\mathbf{b}_a = [b_{a,\sigma}\ b_{a,\mu}\ b_{a,\tau}]^\top$ is the bias vector

4.3 Parameter Estimation

The true acceleration can be estimated from sensor outputs by inverting the model:

$$\hat{\mathbf{a}} = \mathbf{R}_a^{-1} \mathbf{K}_a^{-1} (\mathbf{u}_a - \mathbf{b}_a) \tag{3}$$

Full calibration of a three-axis accelerometer requires estimation of 15 parameters:

- 3 bias terms ($\mathbf{b}_a$)
- 3 scale factors (diagonal of $\mathbf{K}_a$)
- 9 orientation parameters ($\mathbf{R}_a$)

This comprehensive model accounts for individual sensor characteristics, cross-axis sensitivity, and manufacturing misalignments, enabling accurate acceleration measurements from raw sensor outputs.

5 Accelerometer Calibration Algorithm

This section presents a complete calibration method for determining the bias, scale factors, and orientation matrix of a three-axis accelerometer using gravity as a reference.

5.1 Bias Determination

The bias vector $\mathbf{b}_a$ can be estimated by comparing measurements taken in opposite orientations:

$$\begin{aligned} \mathbf{u}_{a1} &= \mathbf{K}_a\mathbf{R}_a\mathbf{g} + \mathbf{b}_a \\ \mathbf{u}_{a2} &= \mathbf{K}_a\mathbf{R}_a(-\mathbf{g}) + \mathbf{b}_a \end{aligned} \tag{4}$$

where $\mathbf{g}$ is the gravity vector in the reference frame Σ. The bias is obtained by averaging:

$$\mathbf{b}_a = \frac{\mathbf{u}_{a1} + \mathbf{u}_{a2}}{2} \tag{5}$$

5.2 Complete Six-Position Calibration

For full calibration, measurements are taken in all six orthogonal orientations:

$$\begin{aligned} \mathbf{U}_{a+} &= \mathbf{K}_a\mathbf{R}_a\mathbf{G} + \mathbf{B}_a \\ \mathbf{U}_{a-} &= \mathbf{K}_a\mathbf{R}_a(-\mathbf{G}) + \mathbf{B}_a \end{aligned} \tag{6}$$

where:

- $\mathbf{U}_{a+}$, $\mathbf{U}_{a-}$ are 3×3 measurement matrices for positive and negative orientations
- $\mathbf{G} = [\mathbf{g}\ \mathbf{g}\ \mathbf{g}]$ is the gravity vector replicated for three axes
- $\mathbf{B}_a = [\mathbf{b}_a\ \mathbf{b}_a\ \mathbf{b}_a]$ is the bias matrix

5.3 Scale Factor and Orientation Estimation

5.3.1 Difference Matrix Calculation

The difference matrix eliminates bias effects:

$$\mathbf{U}_{aD} = \mathbf{U}_{a+} - \mathbf{U}_{a-} = 2\mathbf{K}_a\mathbf{R}_a\mathbf{G} \tag{7}$$

5.3.2 Scale Factor Determination

The scale factors are obtained from the diagonal elements of:

$$\begin{bmatrix} k_{a,\sigma}^2 \\ k_{a,\mu}^2 \\ k_{a,\tau}^2 \end{bmatrix} = \operatorname{diag}\left[\left(\frac{1}{2}\mathbf{U}_{aD}\mathbf{G}^+\right)\left(\frac{1}{2}\mathbf{U}_{aD}\mathbf{G}^+\right)^\top\right] \tag{8}$$

where $\mathbf{G}^+$ is the pseudoinverse of $\mathbf{G}$.

5.3.3 Orientation Matrix Calculation

The orientation matrix is computed by normalizing the rows:

$$\mathbf{R}_a = \mathbf{K}_a^{-1}\left(\frac{1}{2}\mathbf{U}_{aD}\mathbf{G}^+\right) \tag{9}$$

Table 1: Calibration Parameter Summary

Parameter	Number of Values
Bias vector $\mathbf{b}_a$	3
Scale factors $\mathbf{K}_a$	3
Orientation matrix $\mathbf{R}_a$	9
Total	15

5.4 Implementation Considerations

- The rotation axis for 180° rotations should be approximately horizontal
- Small deviations from ideal conditions can be corrected numerically
- The method is insensitive to the initial orientation of the reference frame
- All three accelerometers are calibrated simultaneously

6 Implementation

The calibration and denoising of the MPU6050 accelerometer measurements were performed through the following steps:

6.1 Sensor Orientation and Data Collection

- **Sensor Mounting**: The MPU6050 sensor was securely attached to a rigid surface using double-sided tape to ensure consistent orientation during measurements.
- **Orientation Positions**: The sensor was positioned in six distinct orientations by aligning each axis (x, y, z) both parallel and antiparallel to the gravitational vector.
- **Data Collection**: For each orientation, multiple measurements were recorded to capture the sensor's response under different conditions, ensuring comprehensive data for calibration.

6.2 Data Acquisition

- **Library Utilization**: The Adafruit MPU6050 library was employed to interface the sensor with Arduino Uno, facilitating data transmission over the USB port.
- **Data Logging**: A Python script was developed to read the sensor's output and log the data into CSV files for subsequent analysis.

6.3 Data Recording

- **Session Structure**: Six separate data collection sessions were conducted, each corresponding to one of the six predefined orientations.
- **Measurement Conditions**: During each session, the sensor remained stationary to minimize dynamic effects and obtain stable readings.

6.4 Calibration Parameter Estimation

- **Data Processing**: The recorded data was processed using Python with our implementation and imucal library which also implements Ferraris Calibration

7 Calibration Results

7.1 Estimated Parameters

The calibrated parameters for the accelerometer are presented in Table **??**:

Parameter	**Matrix Representation**
Bias Vector	$\begin{bmatrix} 0.38395 \\ -0.13130 \\ 0.43695 \end{bmatrix}$
Scale Matrix	$\begin{bmatrix} 1.00173649 & 0 & 0 \\ 0 & 1.00529907 & 0 \\ 0 & 0 & 1.01810694 \end{bmatrix}$
Rotation Matrix	$\begin{bmatrix} 0.99895341 & -0.00842065 & 0.04495761 \\ 0.01155954 & 0.99992051 & -0.00503448 \\ -0.05108319 & 0.00527653 & 0.99868046 \end{bmatrix}$

Table 2: Matrix Form of Calibration Parameters

7.2 Calibration Validation Results

7.2.1 Error Analysis

The calibration performance was evaluated using six static orientation tests with expected gravity vectors. Key error metrics were computed as follows:

Table 3: Error Metrics Summary

Metric	**Value (m/s^2)**
RMS Error	0.0342
Maximum Error	0.0866
Mean Error Norm	0.0547

7.2.2 Scale Factor Accuracy

The scale factor deviations from ideal unity gain were:

$$\text{Scale Deviations} = \begin{bmatrix} 0.17\% & 0.00\% & 0.00\% \\ 0.00\% & 0.53\% & 0.00\% \\ 0.00\% & 0.00\% & 1.81\% \end{bmatrix} \tag{10}$$

7.2.3 Per-Position Performance

Table **??** shows detailed performance for each test orientation:

Pos.	**Raw (m/s^2)**	**Calibrated (m/s^2)**	**Error Norm**
1	$[-9.43, -0.27, 0.92]$	$[-9.81, -0.05, 0.03]$	0.0630
2	$[10.20, -0.04, -0.10]$	$[9.82, 0.01, -0.09]$	0.0871
3	$[0.46, -9.99, 0.39]$	$[-0.04, -9.81, 0.01]$	0.0359
4	$[0.29, 9.73, 0.49]$	$[0.02, 9.81, -0.00]$	0.0171
5	$[-0.05, -0.08, -9.54]$	$[0.07, 0.00, -9.81]$	0.0686
6	$[0.83, -0.18, 10.42]$	$[-0.06, -0.00, 9.81]$	0.0567

Table 4: Per-Position Calibration Results

8 Position Estimation Accuracy Assessment

8.1 Assessment Methodology

To validate the accelerometer calibration, we performed position estimation tests using double integration of calibrated acceleration data. The assessment pipeline consisted of:

1. Collecting accelerometer data during controlled motion profiles
2. Applying the calibration transform:
$$\mathbf{a}_{\text{cal}} = \mathbf{R}_a^{-1}\mathbf{K}_a^{-1}(\mathbf{a}_{\text{raw}} - \mathbf{b}_a) \tag{11}$$
3. Removing gravity components
4. Double-integrating acceleration to obtain position:
$$\mathbf{p}(t) = \iint_0^t \mathbf{a}_{\text{motion}}(\tau)\, d\tau^2 + \mathbf{v}_0 t + \mathbf{p}_0 \tag{12}$$
5. Comparing with ground truth reference positions and Uncalibrated readings

8.2 Position Calculation

Position will be estimated through double numerical integration of acceleration data using:

8.2.1 Velocity Calculation

Integrated acceleration using the trapezoidal rule:

$$v[n] = v[n-1] + \frac{1}{2}(a_{\text{cal}}[n-1] + a_{\text{cal}}[n]) \cdot \Delta t \tag{13}$$

8.2.2 Position Calculation

Integrated velocity using the trapezoidal rule:

$$p[n] = p[n-1] + \frac{1}{2}(v[n-1] + v[n]) \cdot \Delta t \tag{14}$$

where:

- a is acceleration (m/s^2)
- Δt is sampling period (0.01 s for 100 Hz)
- Initial conditions $v[0] = p[0] = 0$

8.3 Experiment One: Linear Motion Validation

To validate the effectiveness of our calibration model in dynamic conditions, we conducted a controlled linear motion experiment along a custom-designed trajectory. The objective was to assess how well the calibration reduces drift and bias during motion.

8.3.1 Test Setup and Procedure

We prepared a straight track with a slight offset in the x-direction to introduce a small lateral movement. The sensor was initially kept stationary to allow for baseline data collection and bias estimation, minimizing the impact of human handling errors.

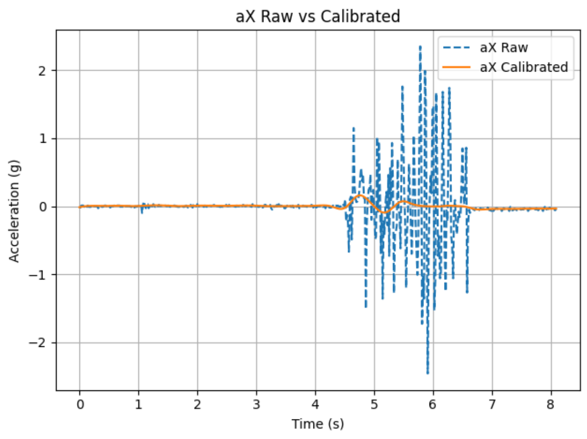
1. **Initial Calibration and Preprocessing**
    - Computed the full calibration transformation:

$$\mathbf{a}_{\text{cal}} = \mathbf{R}_a^{-1}\mathbf{K}a^{-1}(\mathbf{a}_{\text{raw}} - \mathbf{b}_a) \tag{15}$$

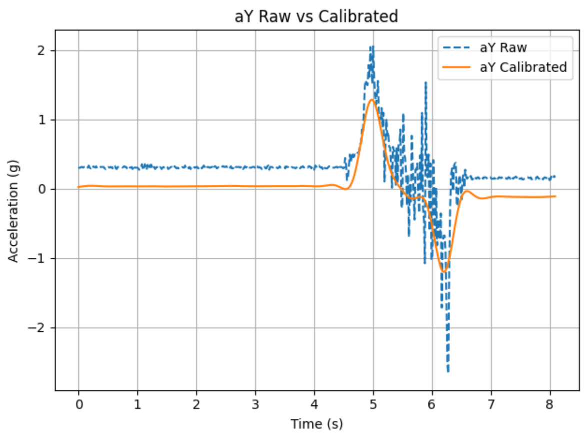
    - Collected 500 samples during the still period (5 seconds at 100Hz) to estimate sensor bias
    - Applied a low-pass filter to suppress high-frequency noise
    - Applied the calibration model to the raw acceleration data
2. **Motion Execution and Data Collection**
    - Moved the IMU along the track, aiming for linear motion with a slight x-axis deviation
    - Attempted to maintain a constant velocity during the motion phase
    - Ended with another still period to observe residual drift
3. **Integration and Analysis**
    - Integrated acceleration to compute velocity and position using the trapezoidal rule
    - Compared calibrated vs. raw data for acceleration, velocity, and trajectory

Figure 2: Estimated position during the linear motion test (calibrated)

8.3.2 Results and Analysis

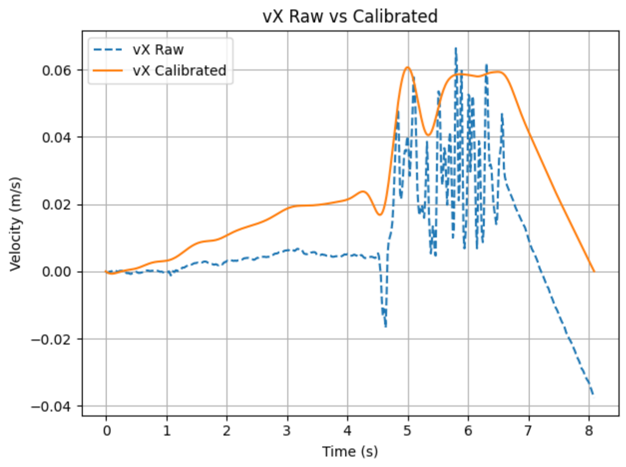


(a) X-axis acceleration (raw vs calibrated)

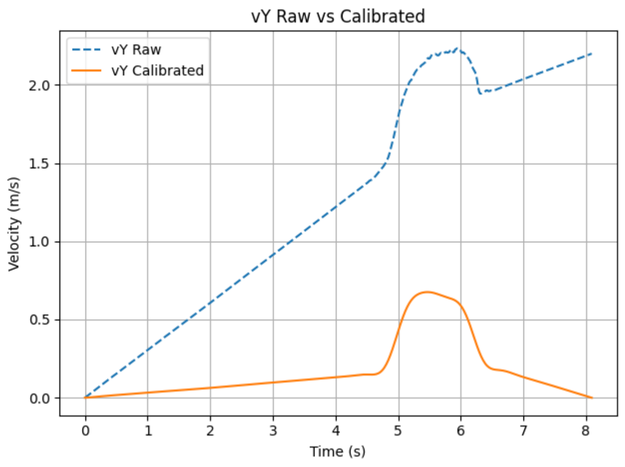


(b) Y-axis acceleration (raw vs calibrated)

Figure 3: Acceleration data before and after calibration



(a) X-axis velocity from integration



(b) Y-axis velocity from integration

Figure 4: Velocity profiles derived from calibrated data

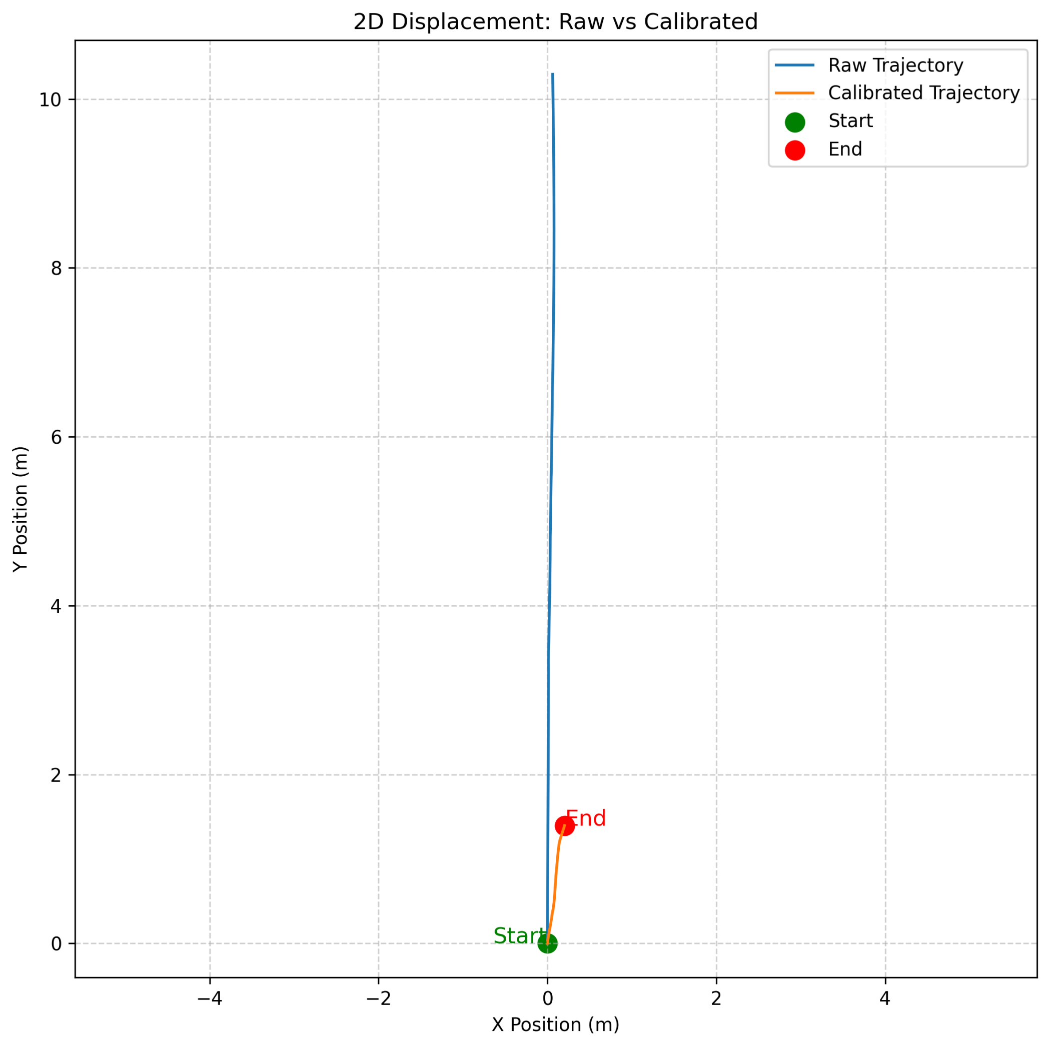


Figure 5: Position trajectory comparison: raw and calibrated (linear test)

Key Observations:

- **Acceleration Data (Fig. 3):**
  - Raw acceleration data exhibited clear bias, especially during stationary intervals.
  - The calibration corrected the baseline, centering the readings around zero and enhancing impulse clarity.
- **Trajectory Comparison (Fig. 6):**
  - The raw trajectory showed **severe drift in the Y direction**, falsely indicating significant movement on a short 0.7-meter track.
  - In the X direction, a **slight drift to the left** occurred, which effectively **canceled out the actual lateral motion** observed during the experiment.
  - This explains why the small X-axis movement was not clearly visible in the raw trajectory—it was masked by the directional bias.
  - After applying bias removal and calibration, the estimated trajectory **closely followed the intended linear path**, remaining stable with minimal drift, which confirms the effectiveness of the calibration in dynamic conditions.

8.4 Experiment Two: Static Sensor

To further validate the calibration and investigate long-term integration effects, a second experiment was conducted with the sensor held completely stationary. The goal was to analyze the drift behavior

8.4.1 Test Setup and Procedure

The sensor was firmly mounted on a stable surface with minimal environmental disturbance. Data was collected for 10 seconds at 100 Hz to capture any internal drift effects.

1. **Still Condition**: The sensor remained completely stationary during the entire capture period.
2. **Calibration and Processing**:
   - Applied the same calibration model used in the linear motion experiment.
   - Removed gravity and any residual bias.
   - Performed numerical integration of acceleration to estimate velocity and position.

8.4.2 Results and Analysis

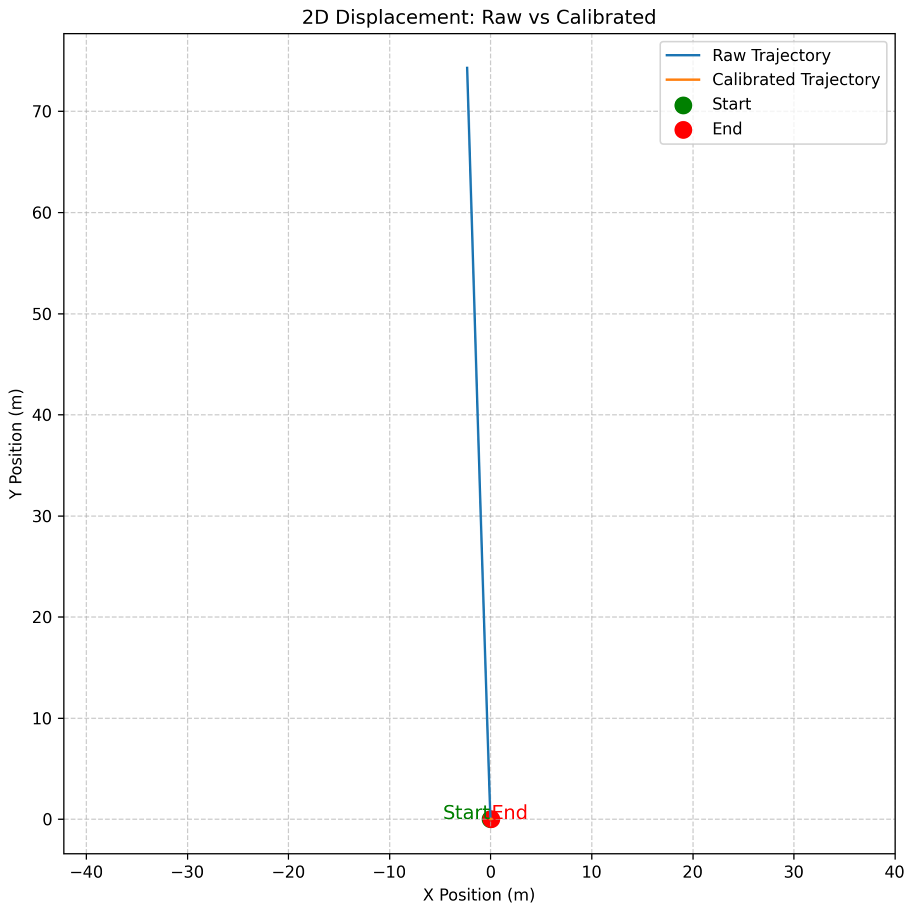


Figure 6: Position trajectory comparison: raw and calibrated

Key Observations:

- **Acceleration Data (Fig. 3):**
  - Raw acceleration data exhibited clear bias.
  - Calibration effectively corrected these biases, centering the readings around zero and enhancing impulse clarity.
- **Trajectory Comparison (Fig. 6):**
  - The raw trajectory exhibited **severe drift in the Y direction**
  - In the X direction, the trajectory drifted **slightly to the left**, which aligns with the linear motion test, where the X drift masked the true lateral displacement."
  - This slight X drift explains why the small lateral movement in the **first experiment** appeared less noticeable—it was effectively canceled out by the natural bias in that direction.
  - After applying bias removal and calibration, the estimated trajectory remained nearly stationary, indicating that the sensor was effectively corrected and exhibited minimal residual drift.

9 References

[1] R. Rogers, *Applied Mathematics in Integrated Navigation Systems*, ser. AIAA Education Series. American Institute of Aeronautics and Astronautics, 2003.

[2] A. B. Chatfield, *Fundamentals of High Accuracy Inertial Navigation*, ser. Progress in Astronautics and Aeronautics. Reston, VA: American Institute of Aeronautics and Astronautics, Inc., 1997.

[3] J. J. Hall and R. L. Williams II, “Inertial Measurement Unit Calibration Platform,” *Journal of Robotic Systems*, vol. 17, no. 11, pp. 623–632, 1998.

[4] D. Tedaldi, A. Pretto and E. Menegatti, ”A robust and easy to implement method for IMU calibration without external equipments,” 2014 IEEE International Conference on Robotics and Automation (ICRA), Hong Kong, China, 2014, pp. 3042-3049, doi: 10.1109/ICRA.2014.6907297.

[5] F. Ferraris, I. Gorini, U. Grimaldi, and M. Parvis, ”Calibration of three-axial rate gyros without angular velocity standards,” *Sensors and Actuators A-Physical*, vol. 42, pp. 446-449, 1994, doi: 10.1016/0924-4247(94)80031-6.

A IMU Data Logging Script

Filename: `imu_data_logger.py`
Purpose: Logs real-time IMU data from an Arduino over serial, based on axis input, and writes it to a CSV file. Pausing and resuming is controlled via the SPACE key.

```python
import serial
import time
import sys
import csv
import keyboard

if len(sys.argv) != 2 or sys.argv[1] not in ['x', 'y', 'z']:
    print("Usage: python script.py <axis>")
    sys.exit(1)

axis = sys.argv[1]
arduino = serial.Serial('COM13', 115200)
time.sleep(2)

filename = f"{axis}_axis.csv"
with open(filename, 'w', newline='') as csvfile:
    csv_writer = csv.writer(csvfile)
    csv_writer.writerow(["Timestamp", "aX (g)", "aY (g)", "aZ (g)",
                         "Temp (C)", "gX (  /s)", "gY (  /s)", "gZ (  /s)"])
    read_count = 0
    collecting = True
    print("Press SPACE to pause/resume data collection.")

    while True:
        if keyboard.is_pressed("space"):
            collecting = not collecting
            print("\nResumed\n" if collecting else "\nPaused\n")
            if collecting:
                for _ in range(3):
                    csv_writer.writerow([])
            while keyboard.is_pressed("space"):
                time.sleep(0.2)

        if collecting and arduino.in_waiting > 0:
            data = arduino.readline().decode('utf-8').strip()
            values = data.split(',')
            if len(values) == 8:
                csv_writer.writerow(values)
                read_count += 1
                print(f"Received: {data}")
```

Listing 1: IMU Data Logging Script

B Ferraris Calibration Script

Filename: imu_ferraris_calibration.py
Purpose: Processes static IMU data from six known orientations and computes a calibration model using the Ferraris method.

```python
from imucal import FerrarisCalibration, ferraris_regions_from_df
import pandas as pd
import numpy as np

folder = "data_folder"
file_paths = {
    "x_a": f"{folder}/x_axis_neg.csv",
    "x_p": f"{folder}/x_axis_pos.csv",
    "y_a": f"{folder}/y_axis_neg.csv",
    "y_p": f"{folder}/y_axis_pos.csv",
    "z_a": f"{folder}/z_axis_neg.csv",
    "z_p": f"{folder}/z_axis_pos.csv"
}

data_chunks = []
for part, file in file_paths.items():
    df_part = pd.read_csv(file)[:100]
    df_part["part"] = part
    data_chunks.append(df_part)

big_df = pd.concat(data_chunks, ignore_index=True)
big_df = big_df.rename(columns={
    "aX (g)": "acc_x", "aY (g)": "acc_y", "aZ (g)": "acc_z",
    "gX": "gyr_x", "gY": "gyr_y", "gZ": "gyr_z"
})
big_df = big_df.drop(['Timestamp', 'Temp (C)'], axis=1)

rot_parts = ["x_rot", "y_rot", "z_rot"]
num_rows = 1
zero_data = np.zeros((num_rows * len(rot_parts), big_df.shape[1]))
rot_df = pd.DataFrame(zero_data, columns=big_df.columns)
rot_df["part"] = np.tile(rot_parts, num_rows)

big_df = pd.concat([big_df, rot_df])
big_df.set_index("part", inplace=True)

regions = ferraris_regions_from_df(big_df)
cal = FerrarisCalibration()
info = cal.compute(regions, 100, 'm/s^2', 'deg/s')
info.to_json_file("results.json")
```

Listing 2: Ferraris Calibration Script

C Linear Motion Analysis Script

Filename: motion_analysis.py
Purpose: Applies smoothing, calibration, and integration to IMU data collected during linear motion to assess drift and trajectory accuracy.

```
# motion_analysis.py

"""
Appendix: IMU Motion Processing and Calibration Script

This script processes raw IMU acceleration data to validate the calibration model
through smoothing, calibration, and numerical integration (trapezoidal rule).
It outputs comparative plots for acceleration, velocity, and position.
"""

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import savgol_filter
from mpl_toolkits.mplot3d import Axes3D

# === Step 1: Load and Prepare Data ===
df = pd.read_csv("x_axis.csv")
df['Time'] = (df['Timestamp'] - df['Timestamp'].iloc[0]) / 1000  # Convert ms to seconds

# === Step 2: Calibration ===
bias = np.array([0.38395, -0.1313, 0.43695])
rotation_matrix = np.array([
    [0.99895341, -0.00842065,  0.04495761],
    [0.01155954,  0.99992051, -0.00503448],
    [-0.05108319, 0.00527653,  0.99868046]
])
scale_matrix = np.array([
    [1.00173649, 0, 0],
    [0, 1.00529907, 0],
    [0, 0, 1.01810694]
])

inv_rotation = np.linalg.inv(rotation_matrix)
inv_scale = np.linalg.inv(scale_matrix)

# Apply calibration transformation to each row
calibrated = []
for _, row in df.iterrows():
    raw = np.array([row['aX (g)'], row['aY (g)'], row['aZ (g)']])
    corrected = inv_scale @ inv_rotation @ (raw - bias)
    calibrated.append(corrected)

calibrated = np.array(calibrated)
df['aX_cal'], df['aY_cal'], df['aZ_cal'] = calibrated[:, 0], calibrated[:, 1], calibrated[:, 2]

# Remove remaining bias using initial still samples
df['aX_cal'] -= df['aX_cal'][:500].mean()
df['aY_cal'] -= df['aY_cal'][:500].mean()
df['aZ_cal'] -= df['aZ_cal'][:500].mean()

# === Step 3: Smoothing (Savitzky-Golay filter) ===
df['aX_raw_smooth'] = savgol_filter(df['aX (g)'], 5, 2)
df['aY_raw_smooth'] = savgol_filter(df['aY (g)'], 5, 2)
df['aX_cal_smooth'] = savgol_filter(df['aX_cal'], 5, 2)
df['aY_cal_smooth'] = savgol_filter(df['aY_cal'], 5, 2)

# === Step 4: Acceleration Plot ===
plt.figure()
plt.plot(df['Time'], df['aX_raw_smooth'], '--', label='aX Raw')
plt.plot(df['Time'], df['aX_cal_smooth'], label='aX Calibrated')
plt.xlabel('Time (s)')
plt.ylabel('Acceleration (g)')
plt.title('aX Raw vs Calibrated')
plt.legend()
```

```python
plt.grid(True)
plt.tight_layout()
plt.show()

plt.figure()
plt.plot(df['Time'], df['aY_raw_smooth'], '--', label='aY Raw')
plt.plot(df['Time'], df['aY_cal_smooth'], label='aY Calibrated')
plt.xlabel('Time (s)')
plt.ylabel('Acceleration (g)')
plt.title('aY Raw vs Calibrated')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# === Step 5: Convert Acceleration to m/s  ===
df['aX_raw_mps2'] = df['aX_raw_smooth']
df['aY_raw_mps2'] = df['aY_raw_smooth']
df['aX_cal_mps2'] = df['aX_cal_smooth']
df['aY_cal_mps2'] = df['aY_cal_smooth']

# === Step 6: Velocity Integration ===
vx_raw, vy_raw = [0], [0]
vx_cal, vy_cal = [0], [0]

for i in range(1, len(df)):
    dt = df['Time'].iloc[i] - df['Time'].iloc[i - 1]

    # Trapezoidal integration
    ax_avg_raw = 0.5 * (df['aX_raw_mps2'].iloc[i] + df['aX_raw_mps2'].iloc[i - 1])
    ay_avg_raw = 0.5 * (df['aY_raw_mps2'].iloc[i] + df['aY_raw_mps2'].iloc[i - 1])
    vx_raw.append(vx_raw[-1] + ax_avg_raw * dt)
    vy_raw.append(vy_raw[-1] + ay_avg_raw * dt)

    ax_avg_cal = 0.5 * (df['aX_cal_mps2'].iloc[i] + df['aX_cal_mps2'].iloc[i - 1])
    ay_avg_cal = 0.5 * (df['aY_cal_mps2'].iloc[i] + df['aY_cal_mps2'].iloc[i - 1])
    vx_cal.append(vx_cal[-1] + ax_avg_cal * dt)
    vy_cal.append(vy_cal[-1] + ay_avg_cal * dt)

df['vX_raw'], df['vY_raw'] = vx_raw, vy_raw
df['vX_cal'], df['vY_cal'] = vx_cal, vy_cal

# === Step 7: Velocity Plot ===
plt.figure()
plt.plot(df['Time'], df['vX_raw'], '--', label='vX Raw')
plt.plot(df['Time'], df['vX_cal'], label='vX Calibrated')
plt.xlabel('Time (s)')
plt.ylabel('Velocity (m/s)')
plt.title('vX Raw vs Calibrated')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

plt.figure()
plt.plot(df['Time'], df['vY_raw'], '--', label='vY Raw')
plt.plot(df['Time'], df['vY_cal'], label='vY Calibrated')
plt.xlabel('Time (s)')
plt.ylabel('Velocity (m/s)')
plt.title('vY Raw vs Calibrated')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# === Step 8: Position Integration ===
px_raw, py_raw = [0], [0]
px_cal, py_cal = [0], [0]

for i in range(1, len(df)):
    dt = df['Time'].iloc[i] - df['Time'].iloc[i - 1]

    vx_avg_raw = 0.5 * (df['vX_raw'].iloc[i] + df['vX_raw'].iloc[i - 1])
    vy_avg_raw = 0.5 * (df['vY_raw'].iloc[i] + df['vY_raw'].iloc[i - 1])
    px_raw.append(px_raw[-1] + vx_avg_raw * dt)
    py_raw.append(py_raw[-1] + vy_avg_raw * dt)

    vx_avg_cal = 0.5 * (df['vX_cal'].iloc[i] + df['vX_cal'].iloc[i - 1])
    vy_avg_cal = 0.5 * (df['vY_cal'].iloc[i] + df['vY_cal'].iloc[i - 1])
    px_cal.append(px_cal[-1] + vx_avg_cal * dt)
    py_cal.append(py_cal[-1] + vy_avg_cal * dt)

df['posX_raw'], df['posY_raw'] = px_raw, py_raw
df['posX_cal'], df['posY_cal'] = px_cal, py_cal

# === Step 9: Position Plot (2D) ===
plt.figure(figsize=(8, 8))
plt.plot(df['posX_raw'], df['posY_raw'], label='Raw Trajectory')
plt.plot(df['posX_cal'], df['posY_cal'], label='Calibrated Trajectory')

plt.scatter(df['posX_cal'].iloc[0], df['posY_cal'].iloc[0], color='green', s=100, label='Start')
plt.text(df['posX_cal'].iloc[0], df['posY_cal'].iloc[0], 'Start', fontsize=12, color='green', ha='
    right')

plt.scatter(df['posX_cal'].iloc[-1], df['posY_cal'].iloc[-1], color='red', s=100, label='End')
plt.text(df['posX_cal'].iloc[-1], df['posY_cal'].iloc[-1], 'End', fontsize=12, color='red', ha='
    left')

plt.xlabel('X Position (m)')
plt.ylabel('Y Position (m)')
plt.title('2D Displacement: Raw vs Calibrated')
plt.axis('equal')
plt.grid(True, linestyle='--', alpha=0.6)
plt.legend()
plt.tight_layout()
plt.show()

# === Step 10: 3D Plot (Position vs Time) ===
df['posZ_raw'] = 0
df['posZ_cal'] = 0

fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111, projection='3d')
ax.plot(df['Time'], df['posX_raw'], df['posY_raw'], '--', label='Raw Position', color='gray')
ax.plot(df['Time'], df['posX_cal'], df['posY_cal'], label='Calibrated Position', color='blue')

ax.set_xlabel('Time (s)')
ax.set_ylabel('X Position (m)')
ax.set_zlabel('Y Position (m)')
ax.set_title('3D Position over Time (Raw vs Calibrated)')
ax.legend()
plt.tight_layout()
plt.show()
```

Listing 3: Motion Analysis Script