Function and characteristics of an Inertial Sensor Cluster / I2C bus

Celestine Machuca (2570138)
Soodeh Mousaviasl (2571713)

Instructor: Prof.Dr.Rasmus Rettig

Date: 26.04.2023

Table of content:

Used materials:

- WEMOS D1 MINI
- Breadboard for prototyping
- MPU_6050 Inertial sensor
- USB-Mini Kabel
- 4 Male-Male Connectors
- Processing (https://processing.org)
- PlatformIO for code compilation and uploading
- Jupiter Notebook running Python For histograms
- Oscilloscope
- Personal PC

Setup:

The MPU6050 inertial sensor cluster has been connected to Wemos D1 mini using the I2C protocol for communication. Only two wires were needed for connection, and two wires for the power. See tables below:

MPU-6050	Wemos D1 mini
VCC	3.3V
GND	GND
SCL	D1
SDA	D2

Hardware setup can be seen in the figure 1:

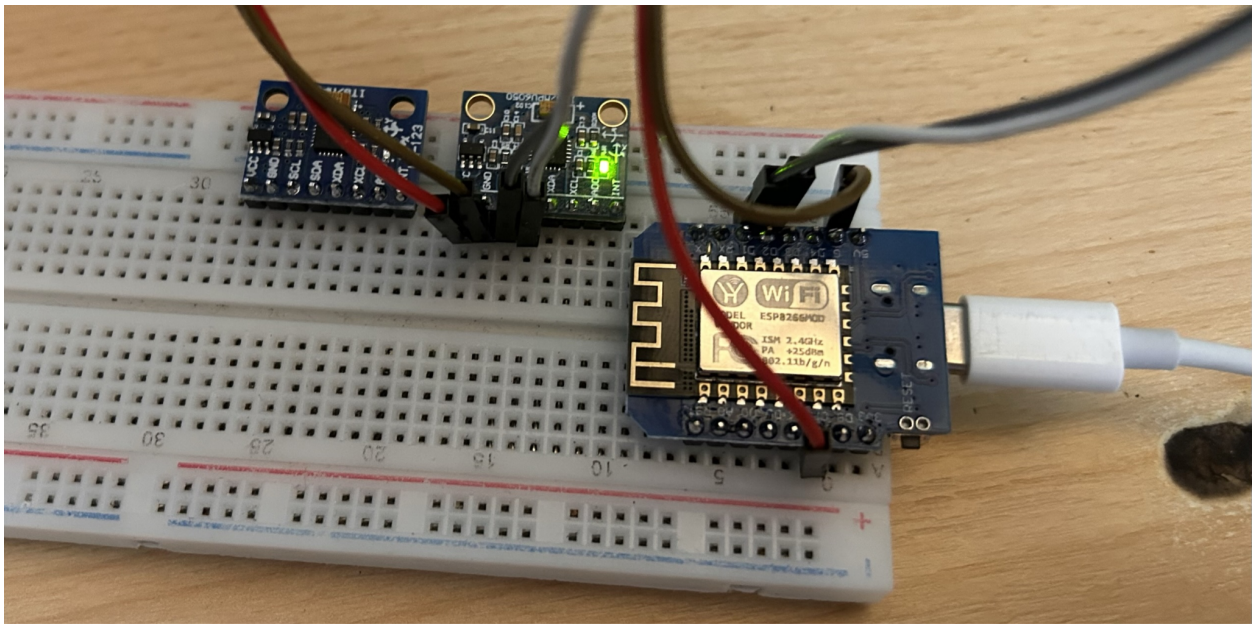


Fig 1: Hardware setup

Next, the given program (**BasisMPU6050**) was compiled and transferred to the WEMOS D1 mini. After that the "**Basisprogramm_ReadUSB_MPU"** program was loaded into the editor Jupiter Notebook running Python.

To investigate the data flow, both programs were analyzed.

The MPU6050 sends data in the form of *16 bit* values from its gyro, accelerometer, and temperature sensor to the Arduino via the I2C bus. The Arduino processes the received data and formats it for transmission. The Arduino sends the formatted data to the laptop via a USB cable using serial communication. A serial terminal or custom software on the laptop reads and interprets the received data, allowing you to view and analyze the measurements.

Note: MPU6050 sensor is an analog device, which produces continuous analog signals. Then these raw signals will be converted into digital values through the ADC inside the sensor.

From the code the data being recorded is:

```
// Accelerometer
int16_t acc_X = (((int16_t)result[0]) << 8) | result[1];
int16_t acc_Y = (((int16_t)result[2]) << 8) | result[3];
```

```
int16_t acc_Z = (((int16_t)result[4]) << 8) | result[5];

// Temperature sensor
int16_t temp = (((int16_t)result[6]) << 8) | result[7];
int16_t tempC = temp / 340 + 36.53;

// Gyroscope
int16_t gyr_X = (((int16_t)result[8]) << 8) | result[9];
int16_t gyr_Y = (((int16_t)result[10]) << 8) | result[11];
int16_t gyr_Z = (((int16_t)result[12]) << 8) | result[13];
```

Example output:

```
{"acc_X":4210,"acc_Y":15,"acc_Z":-1416,"temp":-4341,"tempC":24,"gyr_X":-83,"gyr_Y":-34,"gyr_Z":36}
{"acc_X":4212,"acc_Y":-2,"acc_Z":-1401,"temp":-4312,"tempC":24,"gyr_X":-85,"gyr_Y":-34,"gyr_Z":38}
{"acc_X":4214,"acc_Y":-4,"acc_Z":-1416,"temp":-4335,"tempC":24,"gyr_X":-85,"gyr_Y":-37,"gyr_Z":34}
{"acc_X":4207,"acc_Y":7,"acc_Z":-1409,"temp":-4332,"tempC":24,"gyr_X":-88,"gyr_Y":-37,"gyr_Z":33}
{"acc_X":4194,"acc_Y":1,"acc_Z":-1385,"temp":-4331,"tempC":24,"gyr_X":-84,"gyr_Y":-36,"gyr_Z":34}
{"acc_X":4216,"acc_Y":13,"acc_Z":-1412,"temp":-4332,"tempC":24,"gyr_X":-87,"gyr_Y":-37,"gyr_Z":33}
{"acc_X":4208,"acc_Y":15,"acc_Z":-1405,"temp":-4330,"tempC":24,"gyr_X":-90,"gyr_Y":-37,"gyr_Z":35}
{"acc_X":4209,"acc_Y":7,"acc_Z":-1389,"temp":-4343,"tempC":24,"gyr_X":-91,"gyr_Y":-40,"gyr_Z":34}
{"acc_X":4211,"acc_Y":10,"acc_Z":-1393,"temp":-4332,"tempC":24,"gyr_X":-88,"gyr_Y":-35,"gyr_Z":33}
{"acc_X":4194,"acc_Y":-1,"acc_Z":-1399,"temp":-4331,"tempC":24,"gyr_X":-86,"gyr_Y":-38,"gyr_Z":35}
```

The following diagram is showing how the data flow of the measurement data from the sensor then to the microcontroller then to the laptop:

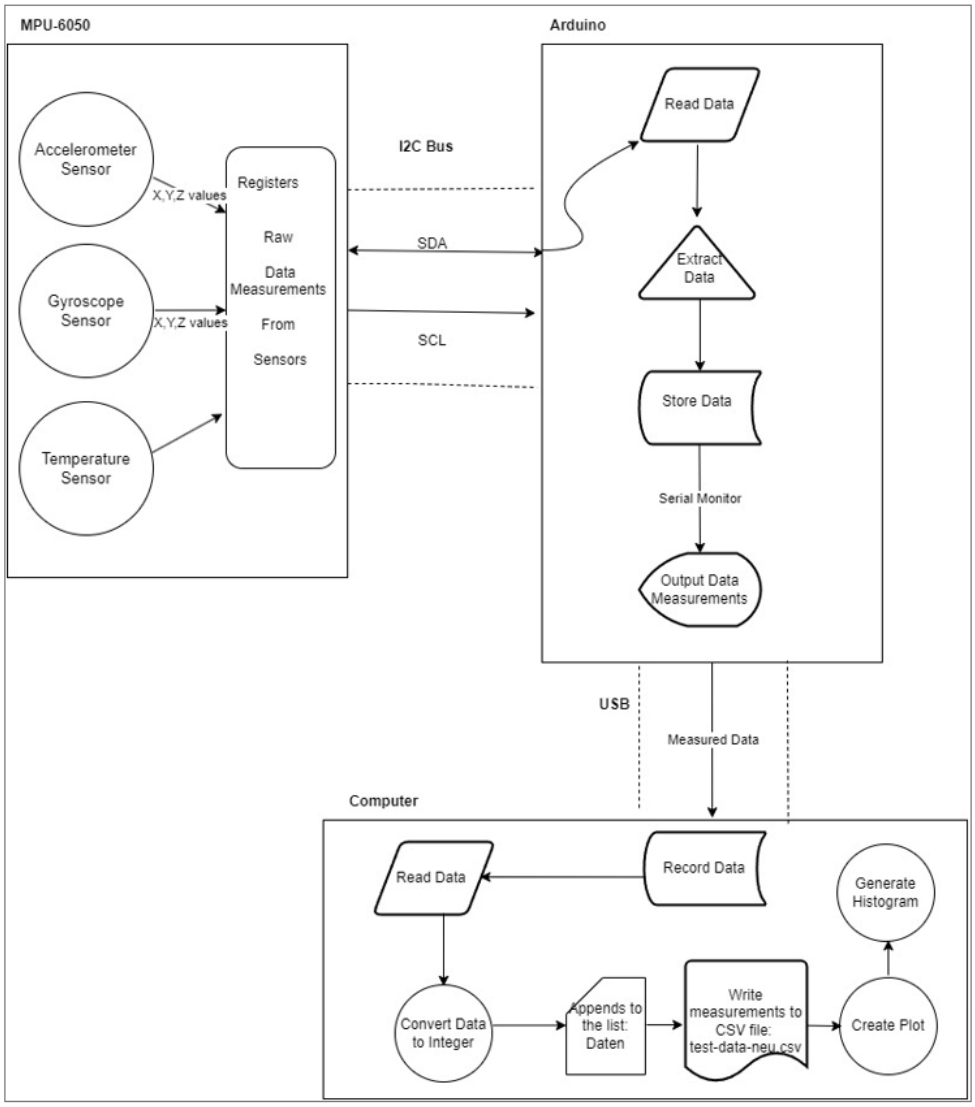


Fig 2: Data flow from sensor to microcontroller to the computer

Part 1: Configuration

Using the datasheet, the sensor cluster can be set to different bandwidth and measurements through modifying the values of specific registers in the MPU-6050's register map.
The 0x1A register – CONFIG – is used to configure the DLPF (Digital Low-Pass Filter) and the sampling rate divider (SMPLRT_DIV). The 0x1B register - GYRO_CONFIG – is for configuring the Gyroscope's full-scale range. And the 0x1C register - ACCEL_CONFIG – is for configuring the full-scale range of the accelerometer.
Hence, to set different bandwidth the DLPF_CFG bits of 0x1A register were modified. The following table from the datasheet represents the available options (https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Register-Map1.pdf):

DLPF_CFG	Accelerometer (F_s = 1kHz)		Gyroscope		
	Bandwidth (Hz)	Delay (ms)	Bandwidth (Hz)	Delay (ms)	Fs (kHz)
0	260	0	256	0.98	8
1	184	2.0	188	1.9	1
2	94	3.0	98	2.8	1
3	44	4.9	42	4.8	1
4	21	8.5	20	8.3	1
5	10	13.8	10	13.4	1
6	5	19.0	5	18.6	1
7	RESERVED		RESERVED		8

Modifying the FS_SEL bits of the 0x1B register was used to set the full-scale range of the gyroscope according to the table below from the same source:

FS_SEL	Full Scale Range	LSB Sensitivity
0	± 250 °/s	131 LSB/°/s
1	± 500 °/s	65.5 LSB/°/s
2	± 1000 °/s	32.8 LSB/°/s
3	± 2000 °/s	16.4 LSB/°/s

And by modifying the AFS_SEL bits of the 0x1C register the full-scale range of the accelerometer was set according to the table below from the same source:

AFS_SEL	Full Scale Range	LSB Sensitivity
0	±2*g*	16384 LSB/*g*
1	±4*g*	8192 LSB/*g*
2	±8*g*	4096 LSB/*g*
3	±16*g*	2048 LSB/*g*

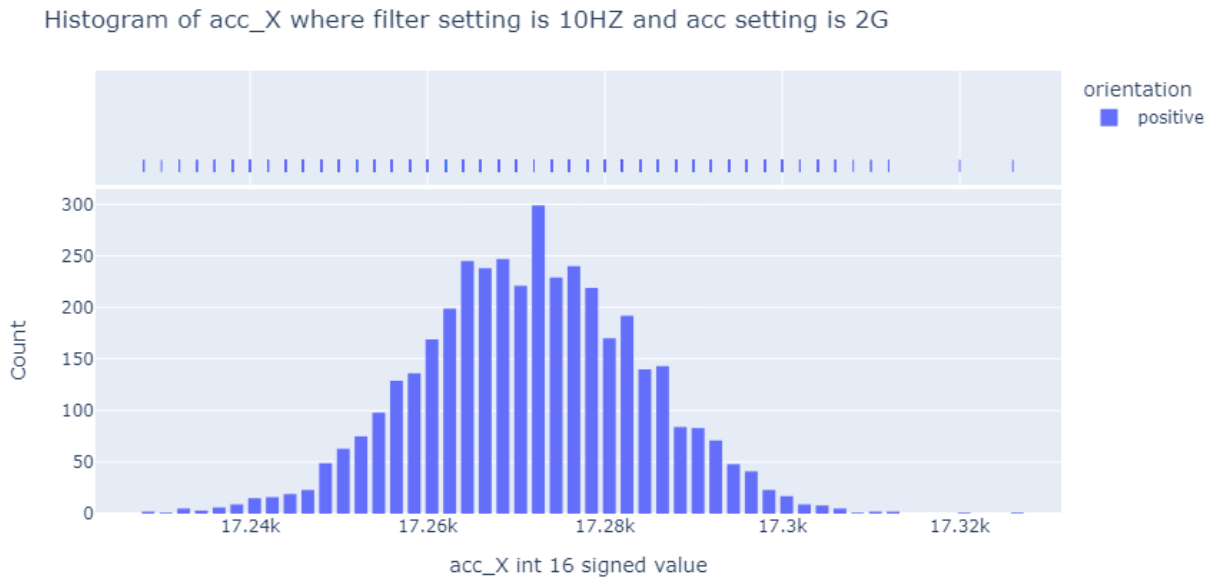
In order to set the bandwidth the following bits of register 0x1A can be selected: 0, 1, and 2. Then, for the measurement range, in order to set in different ranges, the bits 3 and 4 in register 0x1B (gyroscope) can be set. And the same bits for the 0x1C (accelerometer) register can be set.

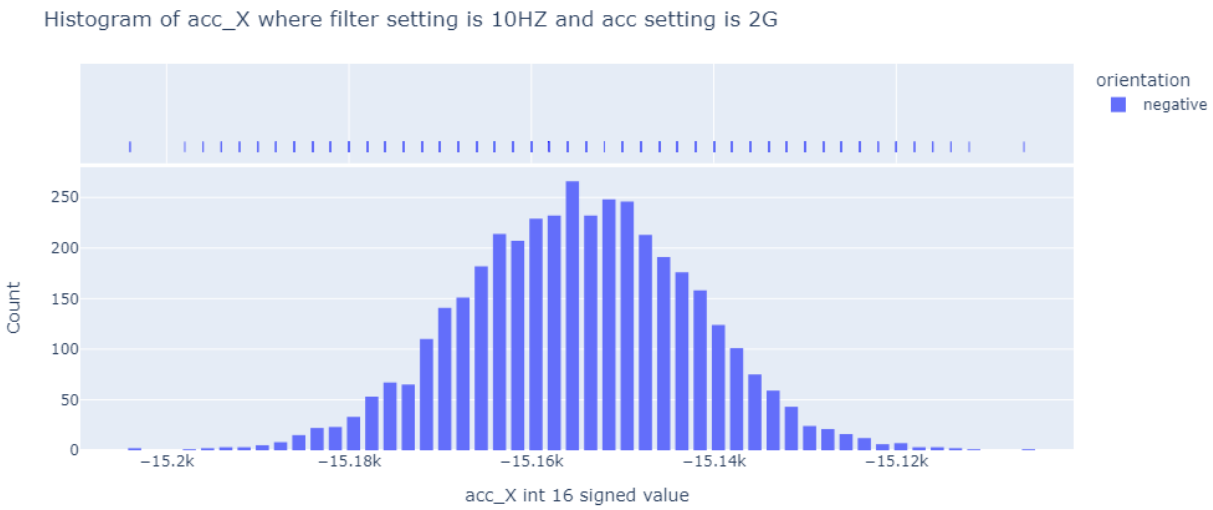
To investigate the different bandwidths and measurements the X-axis of the accelerometer was chosen to change between -1g, 0g, and +1g. The test was performed over ±2g, and ±16g against the earth gravity on the X-axis of the accelerometer. The results are depicted in the following histograms:

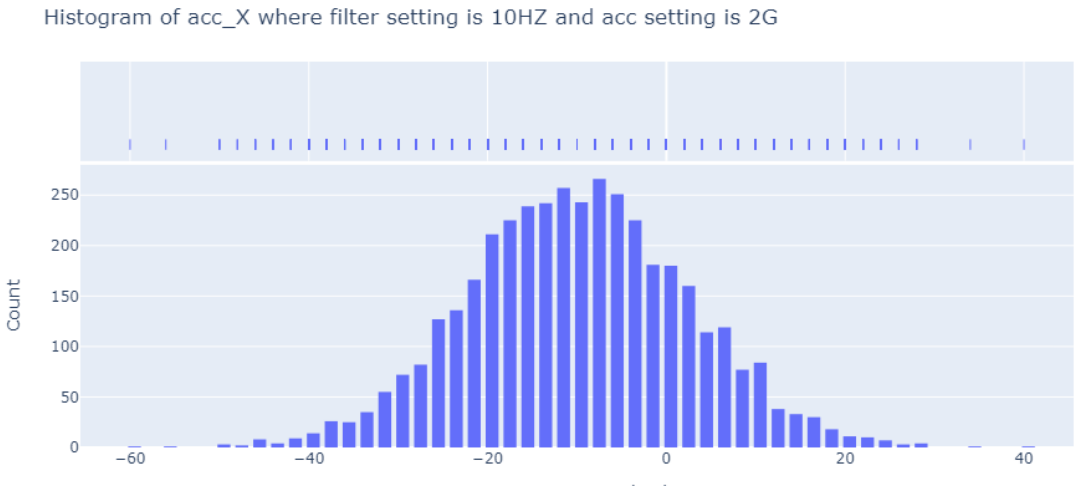
Note:

We added extra histograms from the additional data recorded in our dataset with the same setup as proof.

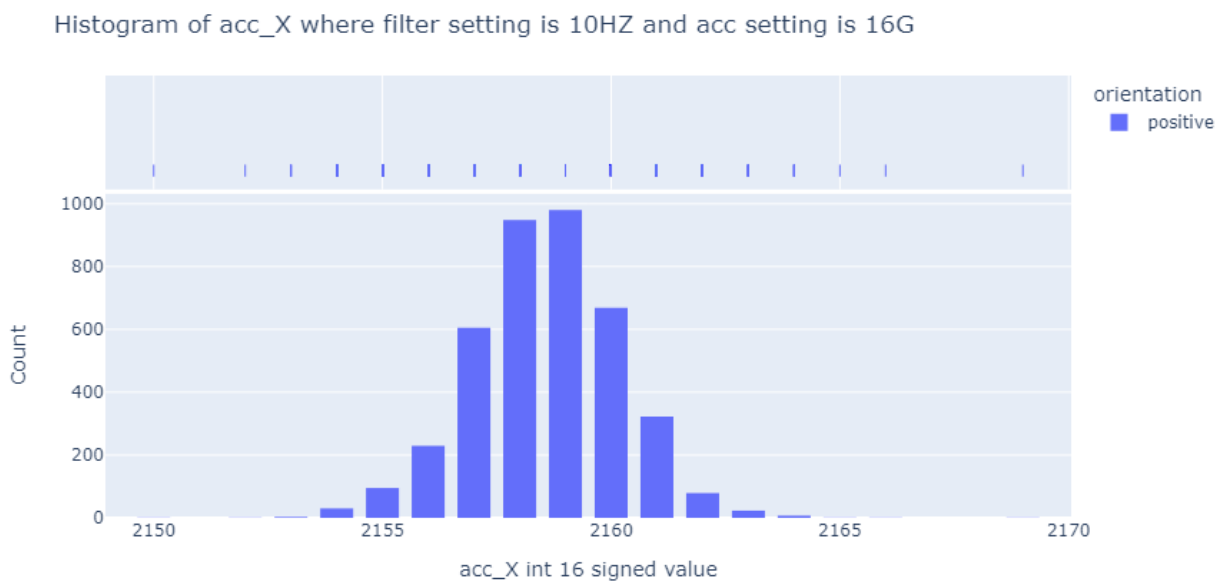
Additionally we attached the raw data set used in the submission.

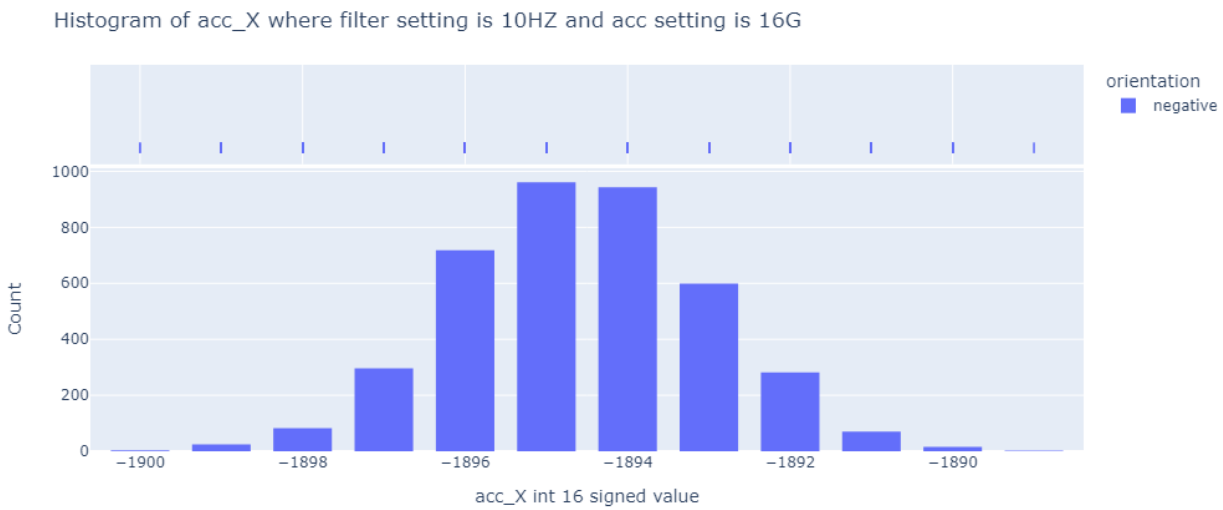


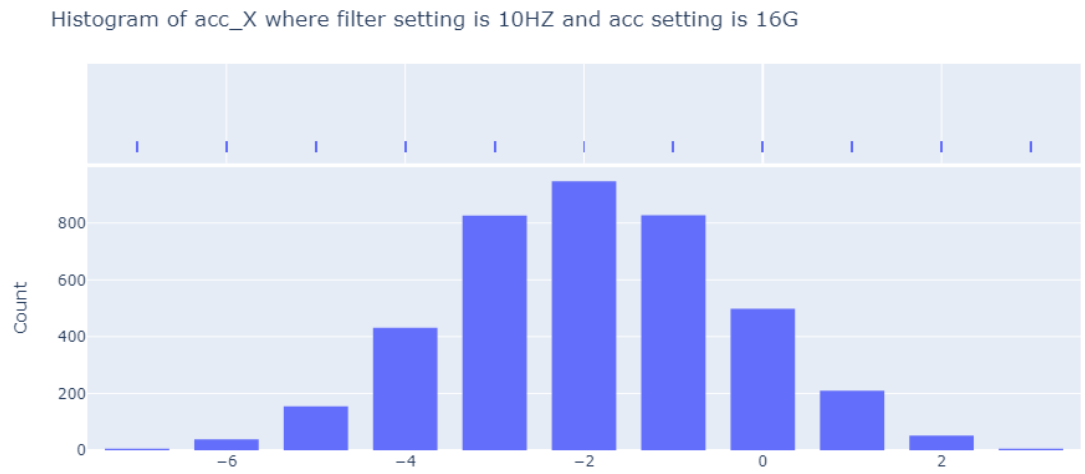




Histogram for 0g position

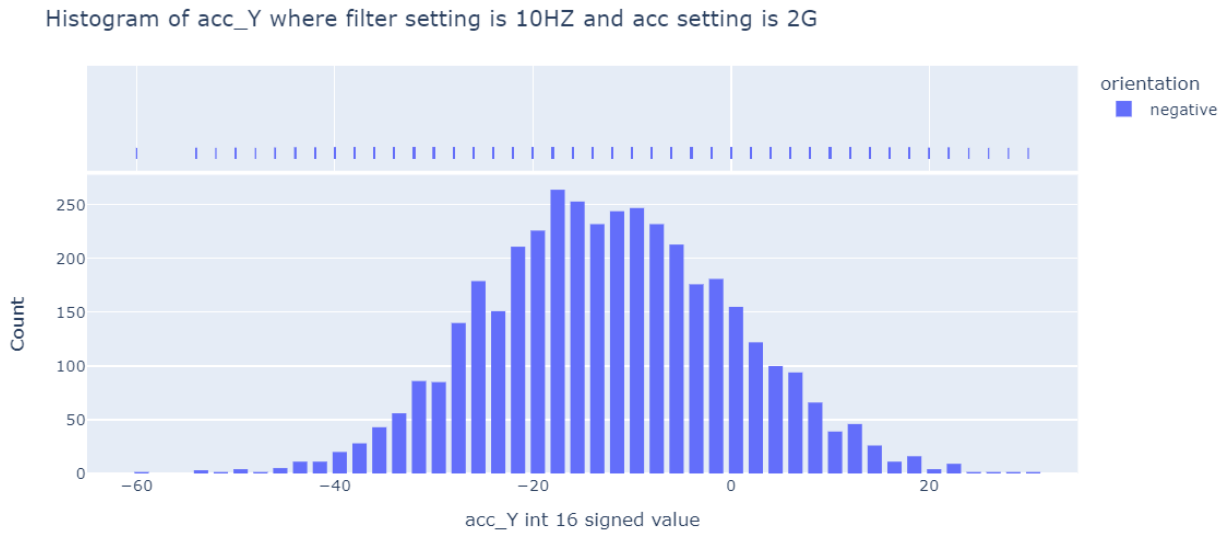

Histogram of acc_X where filter setting is 10HZ and acc setting is 16G
orientation
positive
Count
1000
800
600
400
200
0
2150
2155
2160
2165
2170
acc_X int 16 signed value

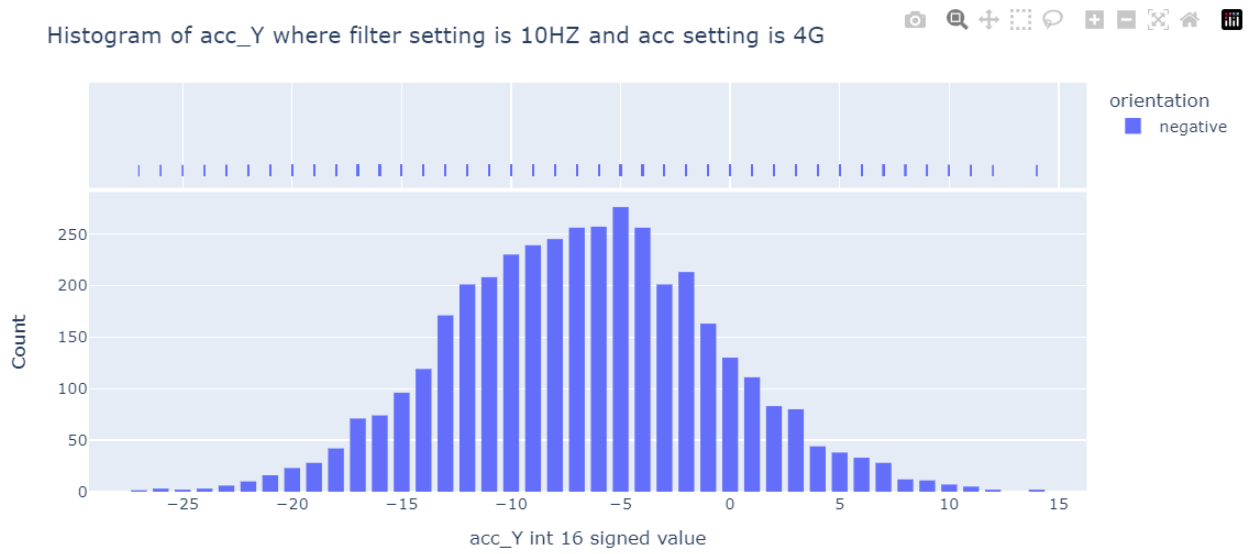

Histogram of acc_X where filter setting is 10HZ and acc setting is 16G
orientation
negative
Count
1000
800
600
400
200
0
−1900
−1898
−1896
−1894
−1892
−1890
acc_X int 16 signed value

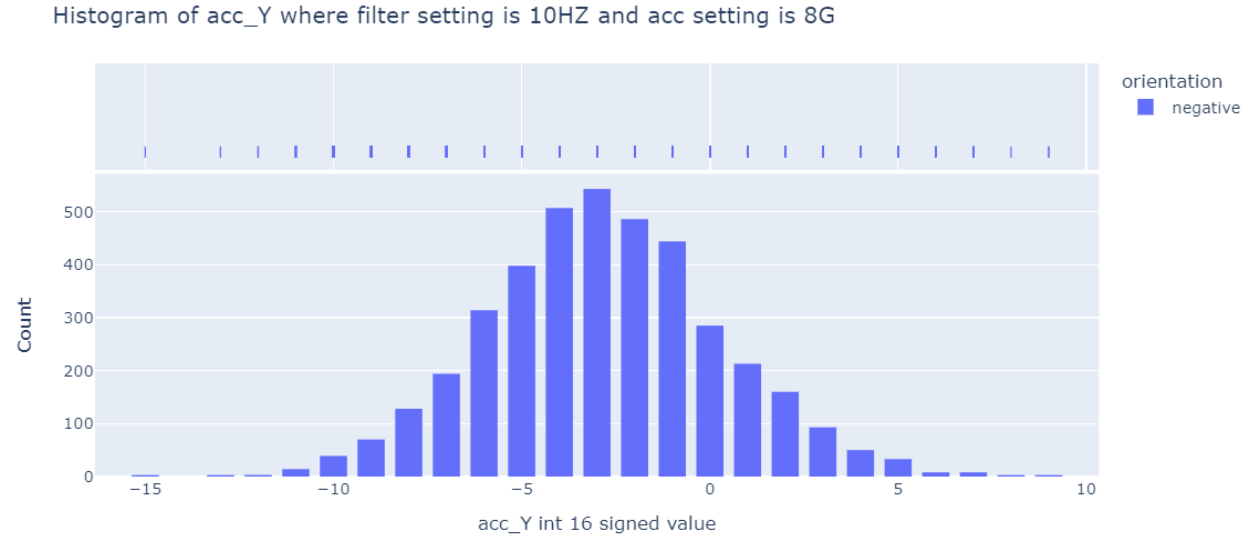


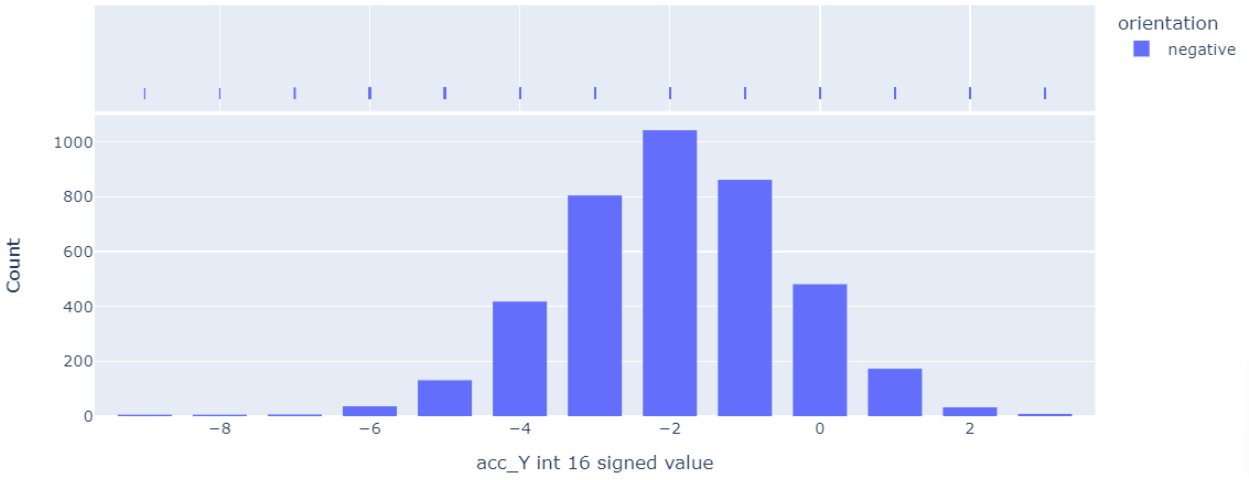
Histogram for 0g position

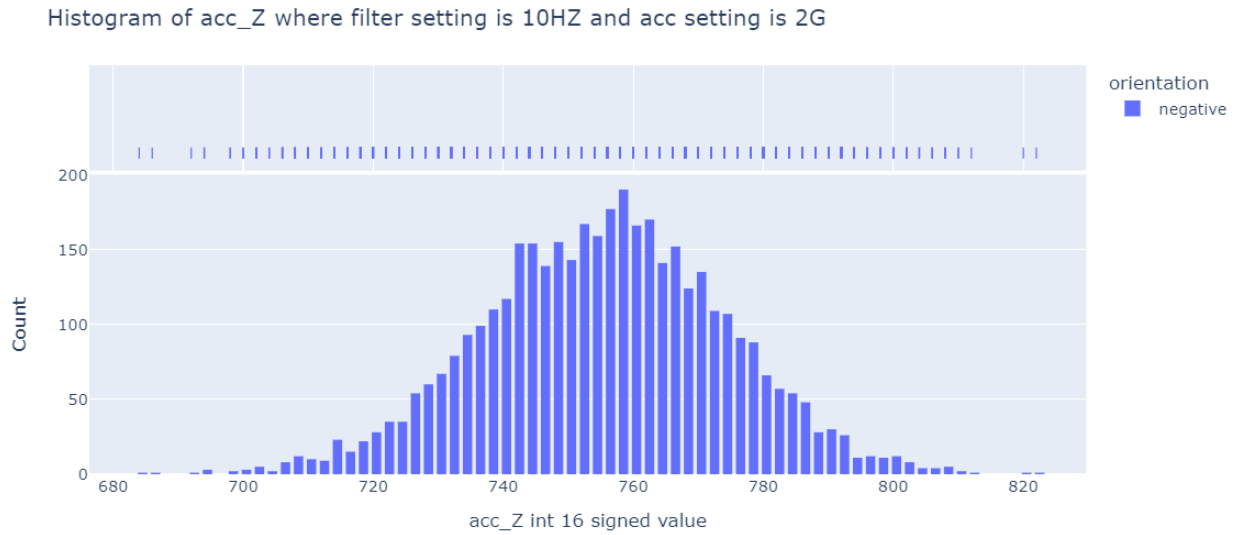
Extra Data from other axis

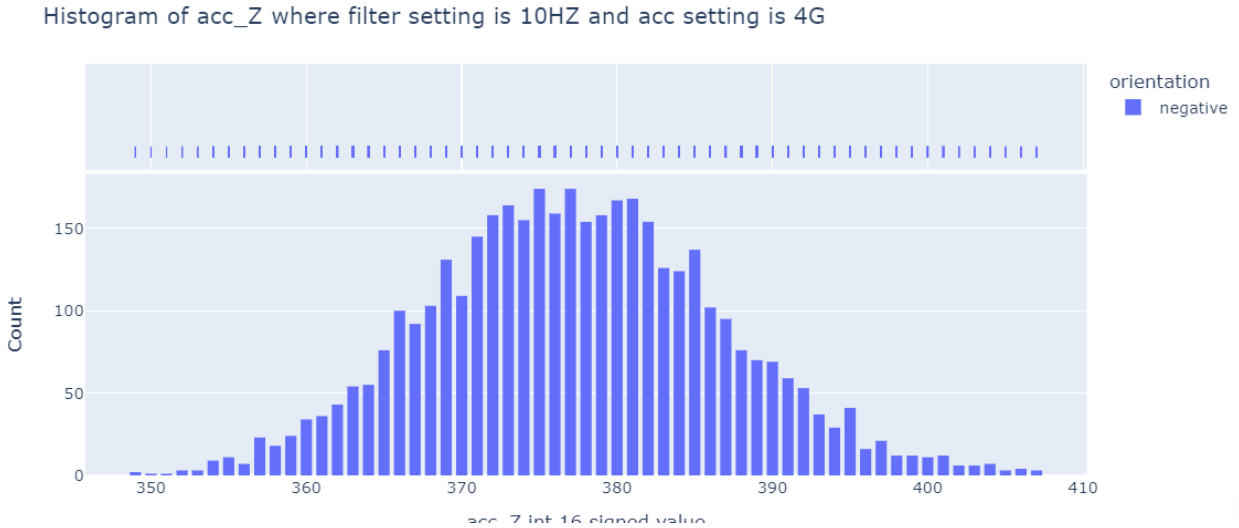


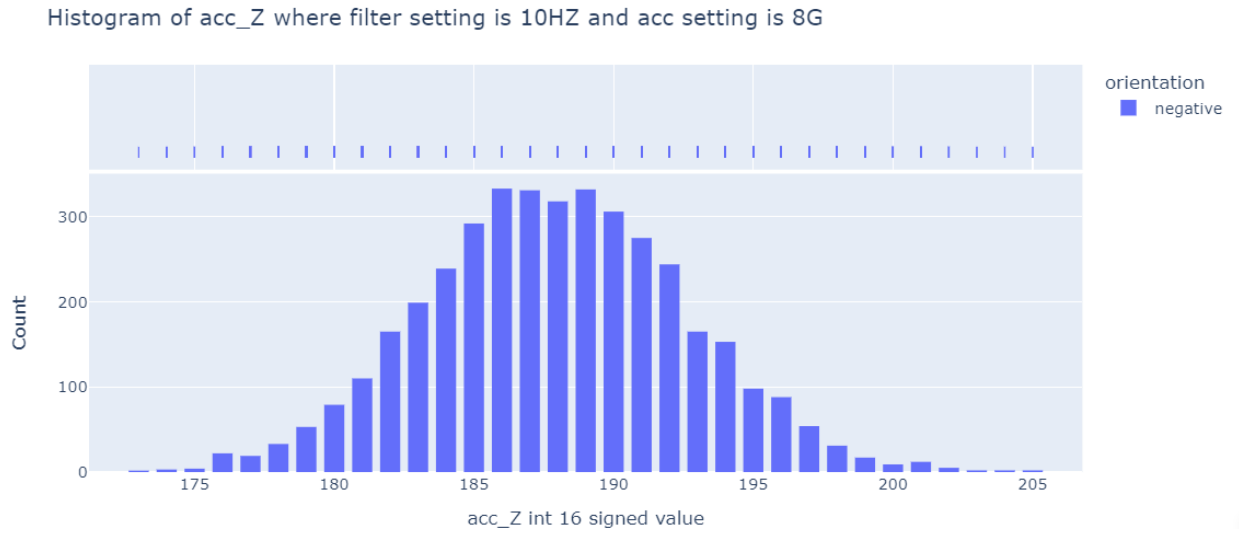

Histogram of acc_Y where filter setting is 10HZ and acc setting is 4G
orientation
negative
Count
0
50
100
150
200
250
−25
−20
−15
−10
−5
0
5
10
15
acc_Y int 16 signed value

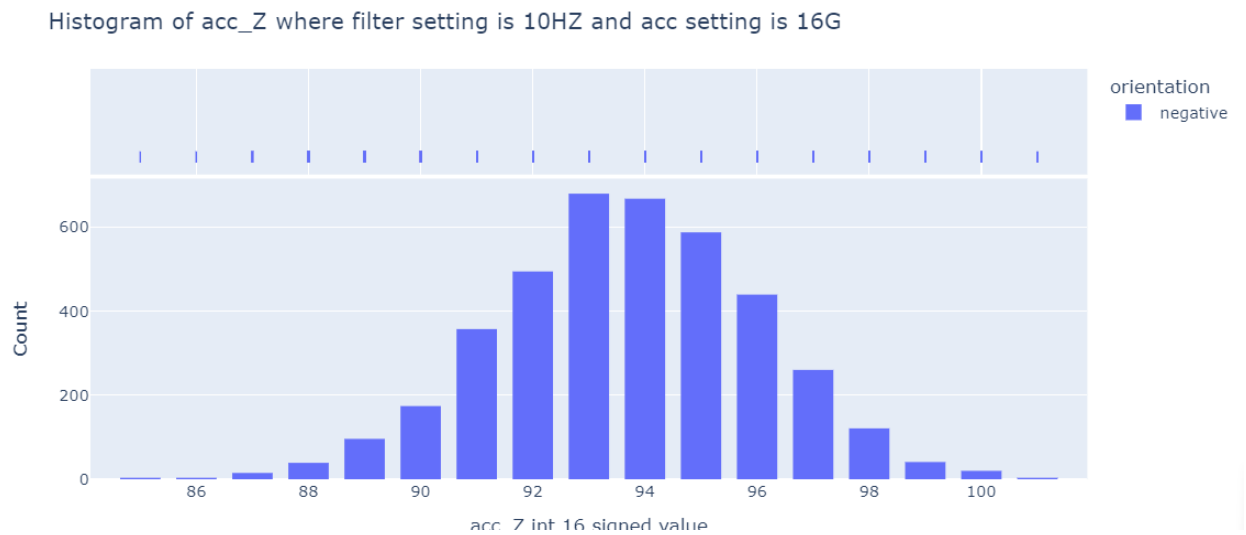

Histogram of acc_Y where filter setting is 10HZ and acc setting is 8G
orientation
negative
Count
0
100
200
300
400
500
−15
−10
−5
0
5
10
acc_Y int 16 signed value


Histogram of acc_Y where filter setting is 10HZ and acc setting is 16G
orientation
negative
Count
0
200
400
600
800
1000
−8
−6
−4
−2
0
2
acc_Y int 16 signed value


Histogram of acc_Z where filter setting is 10HZ and acc setting is 2G
orientation
negative
Count
0
50
100
150
200
680
700
720
740
760
780
800
820
acc_Z int 16 signed value


Histogram of acc_Z where filter setting is 10HZ and acc setting is 4G
orientation
negative
Count
0
50
100
150
350
360
370
380
390
400
410


Histogram of acc_Z where filter setting is 10HZ and acc setting is 8G
orientation
negative
Count
0
100
200
300
175
180
185
190
195
200
205
acc_Z int 16 signed value



What is the resolution of each of these measurements?
given the range int 16 signed from -32768 to 32767, the resolution is 2^16/2 = 32768/2 = 16384 for 1g. 1g/16384 = 0.000061 g/LSB

changing the range translates on to the table below against the theoretical resolution and the measured resolution.

acc-setting	Theoretical resolution [g/LSB]	Measured resolution [g/LSB]
±2g	0.000061	0.0000579
±4g	0.000122	0.000116
±8g	0.000244	0.000232
±16g	0.000488	0.000463

Part 2: Oscilloscope measurements on the I2C Bus

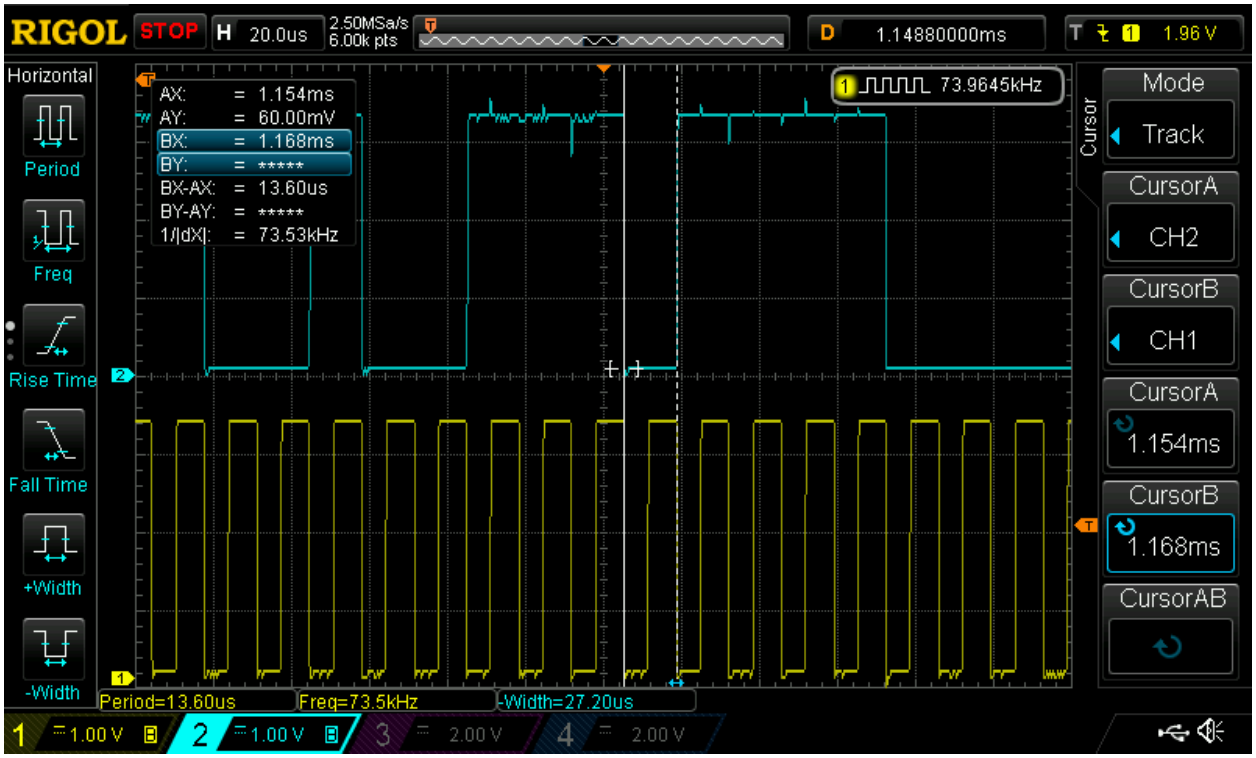
In this part of the experiment the setup was connected to the oscilloscope generating square-wave. One probe was connected to the SCL pin to measure the voltage between SCL and GND and the second probe was connected to SDA to investigate the data line.

The I2C protocol consists of following bits:
- start bit
- 7 bit slave (device) address
- Read/Write bit
- ACK bit
- 8 bit internal register address
- ACK bit
- 8 bit Data
- ACK bit
- Stop bit

Overall 29 bits.
source: https://www.youtube.com/watch?v=6IAkYpmA1DQ&t=68s

A single I2C telegram. SCL signal in yellow, and SDA line in blue measured by an Oscilloscope

To determine the data-rate transferred, the full single I2C telegram can be used. The data-rate between Arduino and sensor is the full time it takes to transfer data through a complete single telegram. This time from the measured data was 1.064 ms so the data rate is 1/1.064ms which is almost 940 measurements per second.

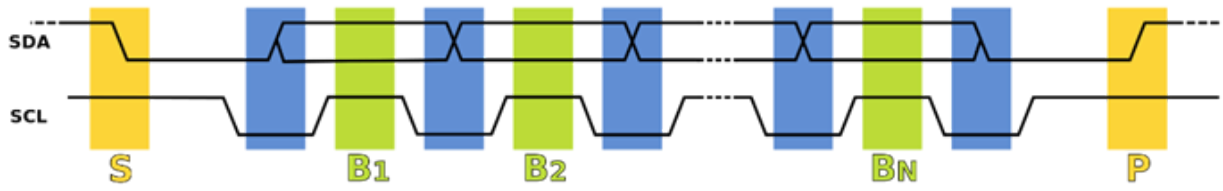


Measurement for the bitrate

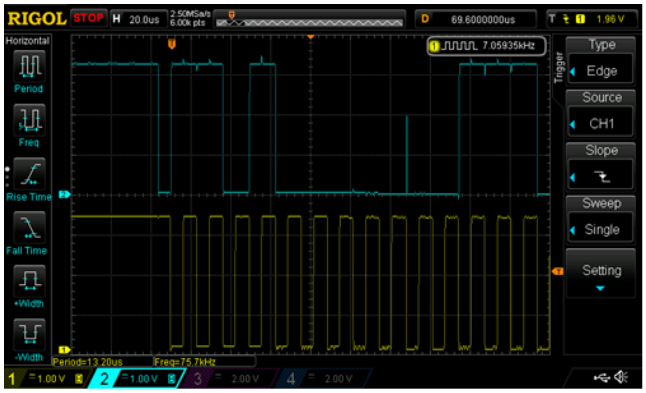
From the data collected by the oscilloscope, it can be seen that the smallest data bit transferred is almost as one period of SCL which was measured as 73.53kHz equal to 13.6 us. This is the frequency of the clock signal and if we assumed 1 bit per clock cycle, then the bitrate would be 73.53kbps, but that would be without taking into account the overhead from the protocol and other factors.

The fact that the smallest data transferred was almost as one SCL cycle indicates that the I2C communication is operating effectively. This means that the I2C transactions are taking advantage of the full bandwidth of the bus and there is no unnecessary delay or idle time between transactions. This improves the system's responsiveness which helps to maximize the data rate of the I2C communication.

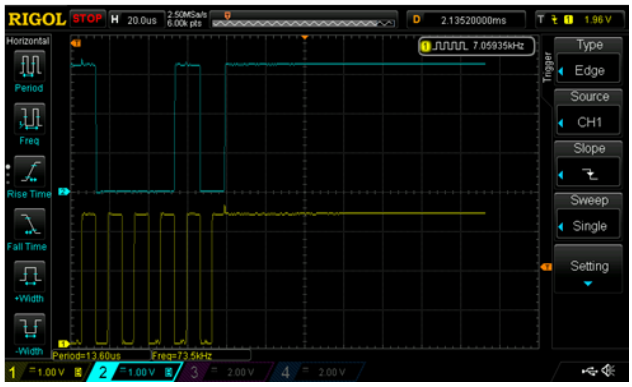
For further investigation of the behavior of the I2C protocol the results from this experiment were compared with the I2C protocol from https://de.wikipedia.org/wiki/I²C.



Time behavior on the I²C bus: The data bits B 1 to B N are transmitted between the start signal (S) and the stop signal (P). source wiki page.

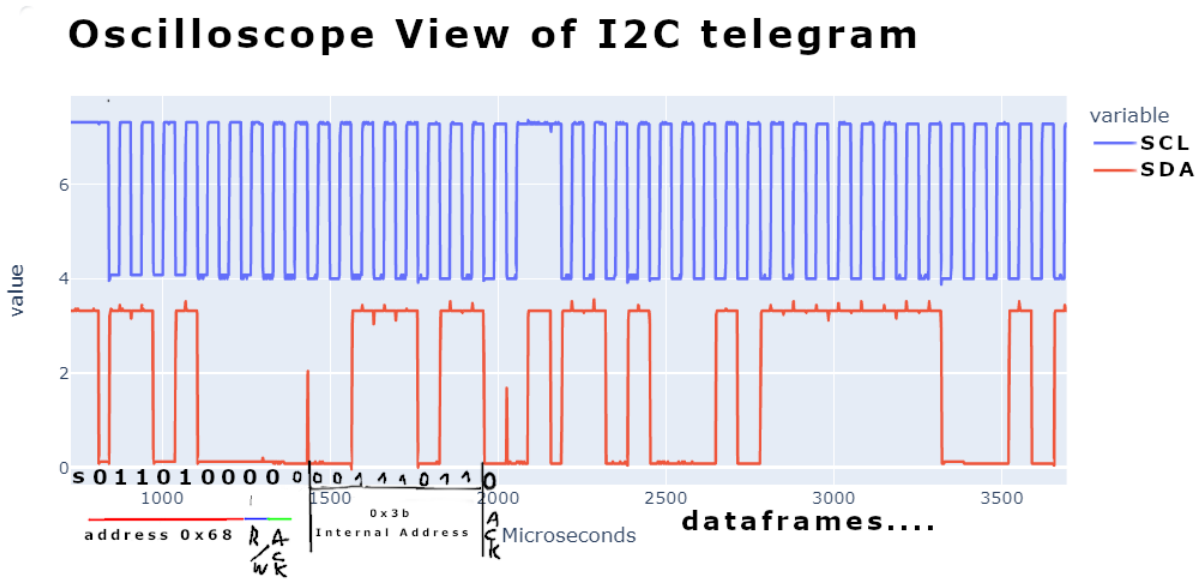


Start signal on falling edge of SDA



Stop signal on rising edge of the SDA

First observation was that the clock signal was present only during the data transmission. See the screenshot of one single I2C telegram.
The start signal(S) which is a falling edge of the data line while the clock is high can be seen in the screenshot above on the left. And the stop signal (P) that is the rising edge of the SDA while SCL is high was also captured and depicted in the screenshot above on the right hand side.



By reviewing the data obtained directly from the oscilloscope as csv values we were able to identify the initial parts of a I2C telegram, as it can be observed the first 8 bits correspond to the slave address in our case 0x68 [hex] (01101000) with the last bit being r/w flag after that we find a low bit ACK, as expected but we observed that the ack bit stays longer that in comparisons with ideal case, this may be due to several factor such as:

- Bus capacitance: it may be the case that there is some parasitic capacitance in our setup(using breadboard can be a factor) or if the pull up resistors are not ideal for the bus specifications.

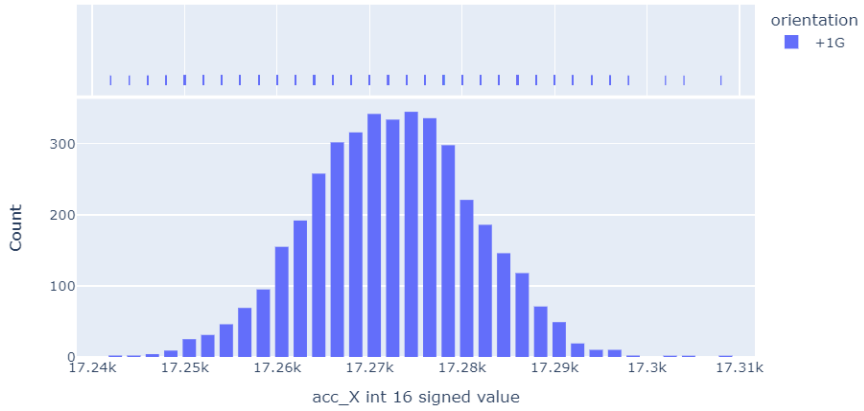
- Software delay: It can be the case that the coded solution was not ideal and further optimizations could be done making the microcontroller take longer than expected.
- Clock stretching: Is a feature mentioned in the I2C protocol and could be used in this case by the slave device if not yet ready by forcing the master to wait until ready[1].

After the ACK we observed the internal register 0x3B [hex] (00111011) corresponds to the start address of the accelerometer x-axis followed by another ACK low bit as expected. After we see the continuation of the message with several data frames as continuation in our case 14 bytes.

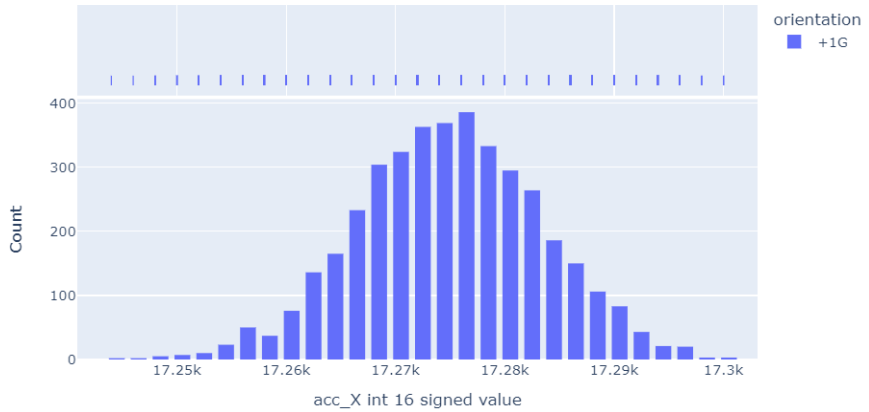
Part 3: Measuring noise on an acceleration sensor

In this section the nose performance of the x-axis for different bandwidth was investigated. The histograms for the full-scale range of ±2g of the accelerometer over all available filters are depicted in the following histogram:
No filter was used!
The part of the code for this test:

```
filter_settings = ['5HZ', '260HZ']
acc_settings = ['2G', '4G', '8G', '16G']
for filter_setting in filter_settings:
    for acc_setting in acc_settings:
        df_filtered = df_all[(df_all['filter_setting'] == filter_setting)
& (df_all['acc_setting'] == acc_setting)]
        display(df_filtered)
        #only where acc_X is negative
        df_filtered = df_filtered[df_filtered['acc_X'] > 0]
        #make a histogram of the acc_X
        fig = px.histogram(df_filtered, x="acc_X", color="orientation",
marginal="rug", hover_data=df_filtered.columns)
        fig.update_layout(
            title_text='Histogram of acc_X where filter setting is ' +
filter_setting + ' and acc setting is ' + acc_setting,
            xaxis_title_text='acc_X int 16 signed value',
            yaxis_title_text='Count',
            bargap=0.2, # gap between bars of adjacent location
coordinates.
            bargroupgap=0.1 # gap between bars of the same location
coordinate.
        )
        fig.show()
```



Histogram Axis X Accelerometer, Filter Setting = 260[Hz], accelerometer setting 2g



Histogram Axis X Accelerometer, Filter Setting = 5[Hz], accelerometer setting 2g

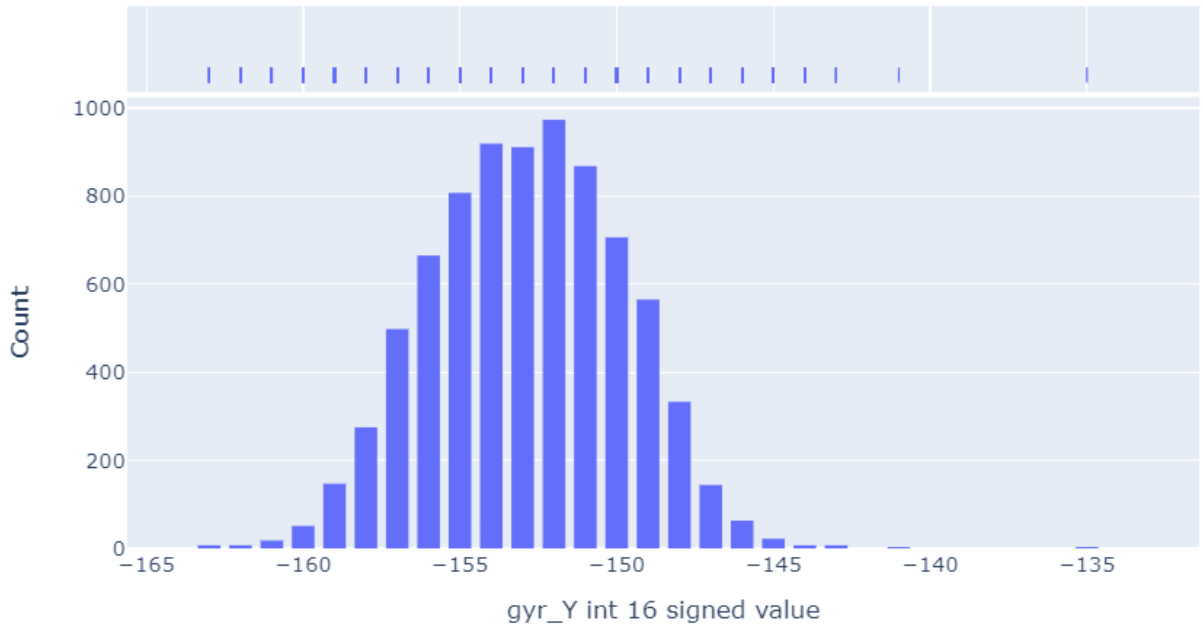
By looking at the distribution shapes of each bandwidth the nature of the noise can be investigated. It can be seen that the distribution is symmetrical and bell-shaped which indicates a normal distribution.
We can observe a larger spread on the 260 Hz corresponds to the low pass filter setting. This means the sensor is more sensible to the noise at higher cutoff frequencies

Part 4: Determination of the noise behavior of a channel of the angular rate sensor

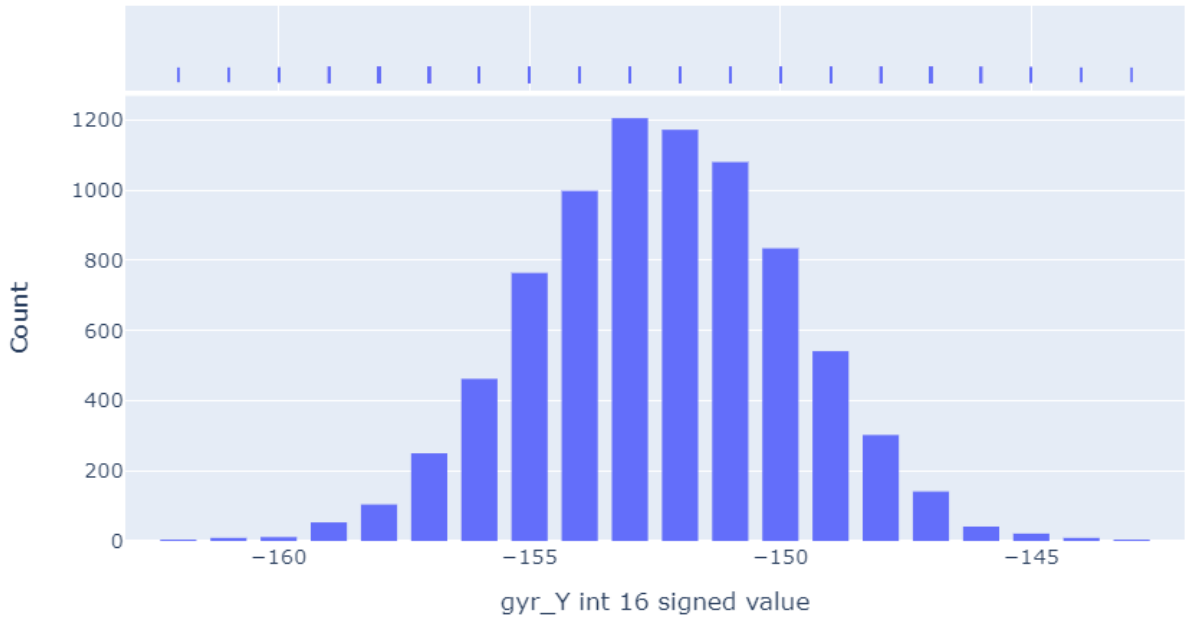
In this part the nose performance of the Y-axis of the gyroscope for 5 Hz and 256 Hz Bandwidth was investigated. The histograms for the full-scale range of ±250 °/s of the gyroscope over all available filters are depicted in the following histogram:
No filter was used!
Code:

```
filter_settings = ['5HZ',  '256HZ']
gyro_settings = ['250']
for filter_setting in filter_settings:
```

```
for gyro_setting in gyro_settings:
    df_filtered = df_all[(df_all['filter_setting'] == filter_setting) & (df_all['gyro_setting'] == gyro_setting)]
    display(df_filtered)
    #make a histogram of the gyr_Y
    fig = px.histogram(df_filtered, x="gyr_Y", marginal="rug", hover_data=df_filtered.columns)
    fig.update_layout(
        title_text='Histogram of gyro_Y where filter setting is ' + filter_setting + ' and gyro setting is ' + gyro_setting,
        xaxis_title_text='gyro_Y int 16 signed value',
        yaxis_title_text='Count',
        bargap=0.2, # gap between bars of adjacent location coordinates.
        bargroupgap=0.1 # gap between bars of the same location coordinate.
    )
    fig.show()
```



Histogram Axis Y Gyroscope, Filter Setting = 256[Hz], Gyroscope setting 250 deg/sec



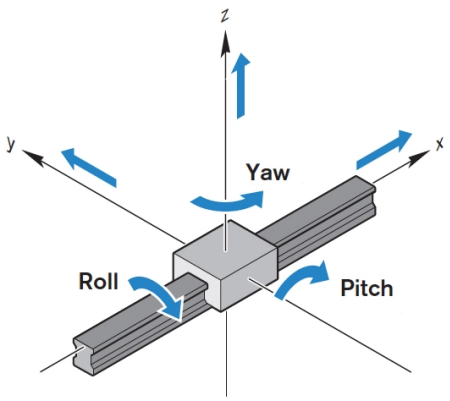
Histogram Axis Y Gyroscope, Filter Setting = 5[Hz], Gyroscope setting 250 deg/sec

As it can be seen also here the shape of the histogram represents a bell-shaped curve indicating that the noise is distributed normally around a mean value, it can be seen that the spread in the case of 256 Hz for the digital low pass filter is wider and in addition with more outliers than the case for a filter setting of 5 Hz meaning that the sensor is more sensible to noise at 256 Hz.

It should be noted that, in order to calculate the yaw rate, the gyroscope measurements must be integrated over time. Since the gyroscope provides angular velocity measurements, it can be integrated to obtain the angle of rotation over time.

Based on these histograms the offset is -153 degrees/sec as opposed to the ideal case where a stationary placement should result in 0 degrees/sec.

Since the sensor only gives angular speed as an output in order to get the euler angles this speed needs to be multiplied by the time between measurements and integrated over it. Given that the sensor was stationary for every test. The angular speed should be zero however that was not the case, if we were doing it simply over time the sensor will drift substantially therefore an offset is used to compensate for this behavior. The offset is obtained from the mean values as follows: roll-offset = -349, pitch_offset = -153, yaw_offset = -140 in this example from the data in the gyroscope set at 250 degrees/sec.



"How to define roll pitch and yaw for linear systems". *[2]*

Part 5: Visualization with "Processing"

For this part the data were evaluated with Processing. The setup had been switched using the given programs: „ArduinoProcessingMPU6050" on the microcontroller board and „ProcessingMPU6050" within Processing.

The function calculates IMU_error capturres 200 samples and calculates the drift on x and y by averaging the samples and obtaining the x and y components of the drift vector.

The value obtained is later used as an offset to correct the drift on the x and y axis.

```
GyroX = GyroX - GyroErrorX ; // GyroErrorX ~(-0.56)
GyroY = GyroY - GyroErrorY; // GyroErrorY ~(2)
GyroZ = GyroZ - GyroErrorZ; // GyroErrorZ ~ (-0.8)
```

Analyzing the the program for processing:

This program reads the data from the serial port, translates the 'cube' and displays the yaw, pitch and roll values.

1. Initiates the serial port:

```
void setup() {
   size (1024, 768, P3D);
  // starts the serial    communication
   myPort = new Serial(this, "COM4", 19200);
   myPort.bufferUntil('\n');
```

```
  // logo = loadImage("UrbanMobilityLab.png");
  // logo2 = loadImage("HAW_Marke.png");
}
```

2. reads the data on serial event

```
void serialEvent (Serial myPort) {
  // reads the data from the Serial Port up to the character '.' and puts
it into the String variable "data".
  data = myPort.readStringUntil('\n');

  // if you got any bytes other than the linefeed:
  if (data != null) {
    data = trim(data);
    // split the string at "/"
    String items[] = split(data, '/');
    if (items.length > 1) {

      //--- Roll,Pitch in degrees
      roll = float(items[0]);
      pitch = float(items[1]);
      yaw = float(items[2]);
    }
  }
}
```

3. draws the cube and the text, call on every frame

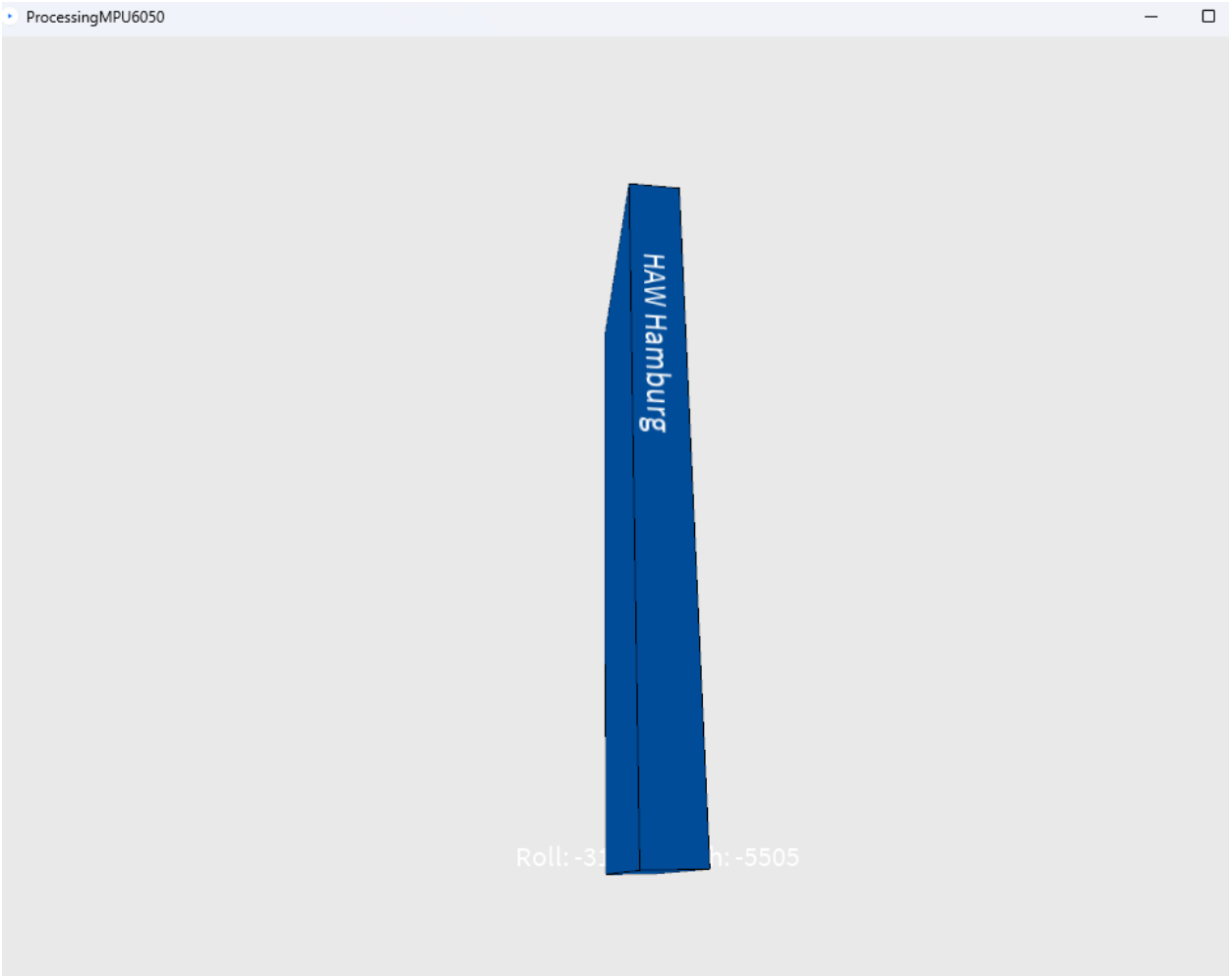
```
void draw() {
  translate(width/2, height/2, 0);
  background(233);
  textSize(22);
  text("Roll: " + int(roll) + "     Pitch: " + int(pitch), -100, 265);

  // Rotate the object
  rotateX(radians(-pitch));
  rotateZ(radians(roll));
  rotateY(radians(yaw));

  // 3D 0bject
  textSize(20);
  fill(0, 76, 153);
  box (500, 40, 200); // Draw box
  textSize(25);
  fill(255, 255, 255);
  text("HAW Hamburg", -183, 10, 101);
  //image(logo,0,-300,400,400);
  //image(logo2,-200,0);
  delay(10);
```

```
  //println("ypr:\t" + angleX + "\t" + angleY); // Print the values to
check whether we are getting proper values
}
```

To try out the program the sensor was moved to observe the behavior on the screen. To know that the sensor is not yet perfectly calibrated the drift of the sensor can be analyzed. So, if the sensor drifts it means that the sensor is not yet perfectly calibrated as the continuous integration of the gyroscope values and its inherent error will add up over time generating a drift.



Animated figure using Processing

Appendix

The following code is the modified of the basis code given by the professor which was used for the part 1 configuration:

```
#include <Arduino.h>
#include <Wire.h>
#define sensor_address 0x68

// keywords = ['@filter_setting', '@accelerometer_setting',
'@gyroscope_setting']
#define FILTER_CONFIG_REG 0x1A

#define SET_FILTER_260HZ 0x06
#define SET_FILTER_184HZ 0x01
#define SET_FILTER_94HZ 0x02
#define SET_FILTER_44HZ 0x03
#define SET_FILTER_21HZ 0x04
#define SET_FILTER_10HZ 0x05
#define SET_FILTER_5HZ 0x06

#define GYRO_CONFIG_REG 0x1B

#define SET_GYRO_250 0x00
#define SET_GYRO_500 0x08
#define SET_GYRO_1000 0x10
#define SET_GYRO_2000 0x18

#define ACC_CONFIG_REG 0x1C

#define SET_ACC_2G 0x00
#define SET_ACC_4G 0x08
#define SET_ACC_8G 0x10
#define SET_ACC_16G 0x18

void SetConfiguration(byte reg, byte setting)
{
  // Aufruf des MPU6050 Sensor
  Wire.beginTransmission(sensor_address);
  // Register Aufruf
  Wire.write(reg);
  // Einstellungsbyte für das Register senden
  Wire.write(setting);
  Wire.endTransmission();
}

void setup()
{
```

```
  Serial.begin(115200);
  pinMode(D7, OUTPUT);
  digitalWrite(D7, LOW);
  delay(1000);
  digitalWrite(D7, HIGH);
  delay(1000);
  Wire.begin();
  delay(1000);

  // Powermanagement aufrufen
  // Sensor schlafen und Reset, Clock wird zunächst von Gyro-Achse Z
verwendet
  // Serial.println("Powermanagement aufrufen - Reset");
  SetConfiguration(0x6B, 0x80);

  // Kurz warten
  delay(500);

  // Powermanagement aufrufen
  // Sleep beenden und Clock von Gyroskopeachse X verwenden
  // Serial.println("Powermanagement aufrufen - Clock festlegen");
  SetConfiguration(0x6B, 0x03);

  delay(500);
  // filter configuration
  SetConfiguration(FILTER_CONFIG_REG, @filter_setting);
  // gyro config
  SetConfiguration(GYRO_CONFIG_REG, @gyroscope_setting);
  // acc config
  SetConfiguration(ACC_CONFIG_REG, @accelerometer_setting);
  delay(500);
}

void loop()
{
  byte result[14];
  result[0] = 0x3B;
  Wire.beginTransmission(sensor_address);
  Wire.write(result[0]);
  Wire.endTransmission();
  Wire.requestFrom(sensor_address, 14);
  for (int i = 0; i < 14; i++)
  {
    result[i] = Wire.read();
  }

  // Accelerometer
```

```
int16_t acc_X = (((int16_t)result[0]) << 8) | result[1];
int16_t acc_Y = (((int16_t)result[2]) << 8) | result[3];
int16_t acc_Z = (((int16_t)result[4]) << 8) | result[5];

// Temperature sensor
int16_t temp = (((int16_t)result[6]) << 8) | result[7];
int16_t tempC = temp / 340 + 36.53;

// Gyroscope
int16_t gyr_X = (((int16_t)result[8]) << 8) | result[9];
int16_t gyr_Y = (((int16_t)result[10]) << 8) | result[11];
int16_t gyr_Z = (((int16_t)result[12]) << 8) | result[13];
// Print data
// json like format
Serial.print("{\"acc_X\":");
Serial.print(acc_X);
//Serial.print(",\"acc_Y\":");
//Serial.print(acc_Y);
//Serial.print(",\"acc_Z\":");
//Serial.print(acc_Z);
//Serial.print(",\"temp\":");
//Serial.print(temp);
//Serial.print(",\"tempC\":");
//Serial.print(tempC);
//Serial.print(",\"gyr_X\":");
//Serial.print(gyr_X);
//Serial.print(",\"gyr_Y\":");
//Serial.print(gyr_Y);
//Serial.print(",\"gyr_Z\":");
//Serial.print(gyr_Z);
//Serial.println("}");
```

References

[1] Clock Stretching

https://www.ti.com/video/6288224255001#:~:text=Clock%20stretching%20is%20a%20method.holding%20the%20SCL%20line%20low.

Texas Instruments.

[2] yaw pitch roll

https://www.linearmotiontips.com/motion-basics-how-to-define-roll-pitch-and-yaw-for-linear-systems/

linear motion tips.