Development of a dual sided and affordable bike power meter

Suitable for increasing training efficiency, injury prevention and injury/illness recovery

Oliver Pargfrieder

A thesis presented for the degree of
Bachelor of Science

Medical Engineering
Univ.-Prof. DI Dr. Werner Baumgartner
Institut für Medizin- und Biomechatronik
Johannes Kepler University Linz
Austria

Thursday 26th February, 2026

Abstract

This projects goal was to build a bike power meter with widely available and cheap parts, but is still accurate enough for practical applications. The device developed, can measure the power put into the bicycle via several sensors which were mounted on the crank set. Four of those sensors, called strain gauges, were glued to each crank and measure how much the crank bends when force is applied to it. Through calibration with known forces, the amount of bending can be linked to the corresponding force in a linear manner. Another type of sensor in form of a microchip which measures angular velocity was also attached. All this data is then fed into a micro controller which is also mounted on the crank set. It can calculate the power out of force, crank length and angular velocity, which it sends via BLE (Bluetooth Low Energy) to a device e.g. a smartphone. Here commercial cycling tracking apps were used to give useful information while cycling and a report when finished. Also a customizable app was made to capture raw data from the device without the preprocessing from commercial apps. The power data is very helpful to make training or medical recovery visible and more efficient.

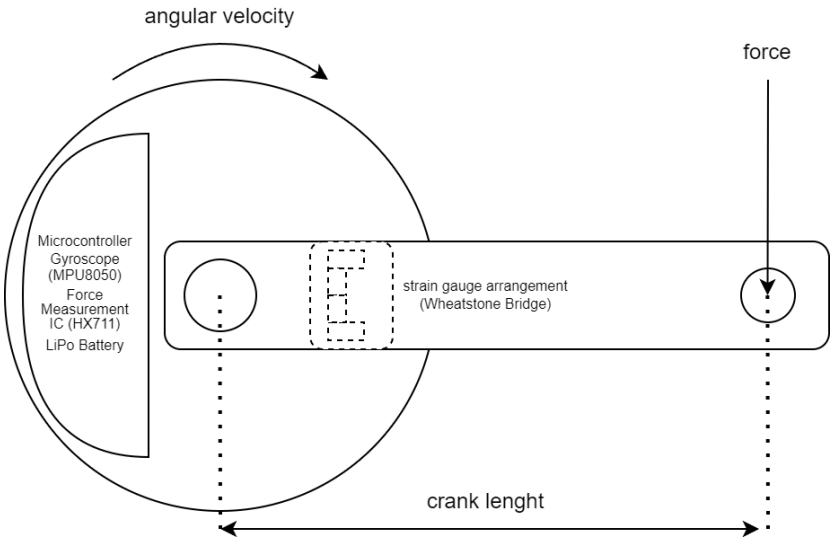


Figure 1: Basic buildup and working principle of the device.

Contents

1 Introduction

The use of bike power meters has increased in recent years as the technology is getting cheaper. But for hobbyists a minimum of 300-500€ can still be too big of a price to pay for such a device. Since the components needed to build a power meter are quite cheap (20-40€ depending on where they are sourced), there is great potential in developing a DIY solution and document this so others can recreate it.

On the current market there are several different solutions to measure power in the field. Virtually all of them use strain gauges to measure the forces applied to a component of the bike. There are pedal based systems where the strain is measured on the pedal axle and all the electronics are mounted on the pedal itself [1]. To measure both the right and left power, two of those pedal power meters are needed which comes with increased cost. Another drawback is that they use clipless pedal systems which need matching cycling shoes. Due to the small spaces available and high build complexity it was decided to not build such a system.

There also exist spider based power meters where the strain is measured near the mounting points of the chain rings to the crank system [2]. They are known to be more accurate because the forces are less complex. Another advantage is that only one system is needed to measure both sides. Even pedal power balance can be estimated by using an algorithm that knows the exact crank position and can therefore conclude which foot applies the force at the moment. Even though this system has many advantages it was decided against it because machined metal parts are needed to build one, which is not good for an easily approachable DYI solution.

In the past there was also a system which was integrated in the shoe and used a flexible force sensor in the sole [3]. To my knowledge, there is no such system on the market right now due to the inaccuracy of it, because foot position influenced the measurements too much. For this reason and the shoe integration it was decided to not go this route.

Then there are crank based power meters, like the one developed here. These measure the strain on the crank arm [2]. The ones available on the market usually are separate left and right crank arm power meters, which communicate via BLE to act as one device. The one built here uses the hollow crank axle to connect the left and right crank arm via cables.

As there are huge benefits in using the data generated by these devices to structure training plans and to prevent under or even over-training as this can lead to injuries and illness [4]. So it makes sense to set training intensity and volume using the Training Stress Score (TSS) to help prevent that. The TSS is a measure applied to a training session to give the user an idea how much stress the users body experienced. Using time (t) in hours, Normalized Power (NP) and Functional Threshold Power (FTP) the TSS can be calculate like:

$$TSS = (NP/FTP)^2 \times t \tag{1}$$

Normalized Power, compared to Average Power, weighs greater power values more than lower ones as they cause more physiological stress [5].

$$NP = \sqrt[4]{\frac{1}{N}\sum_{i=30}^{N}\left(\left(\frac{1}{30}\sum_{j=i-29}^{i}P_j\right)^4\right)} \tag{2}$$

The FTP is the maximum average power a person can produce over one hour. There exist several different methods to derive an approximate value without actually cycling for one hour. For example: $FTP = (averagepowerover20minutes) \times 0.95$. Using the TSS, other scores can be calculated like Acute Training Load (ATL), Chronic Training Load (CTL) and Training Stress Balance (TSB). ATL and CTL are exponentially weighted moving averages. ATL over the last 7 days, thats why it represents acute fatigue. CTL over the last 42 days, so it represents the current fitness level.

$$TSB = CTL - ATL \tag{3}$$

Depending on the goals of the user there are different ranges where the TSB should be. For example while being in a phase of training -10 to -30 is optimal. This means that the user is in a phase of increased stress which causes the body to adapt. For optimal performance in a race or similar, the user should be resting before the event to reach a TSB of +15 to +25. Staying below -30 over several days means overloading the body, which increases the risk of illness or injury. All those calculations are done by the tracking/training software when it has access to power data [6]. A very handy tool to do all this is the free to use website https://intervals.icu/.

2 Methods

2.1 Overview

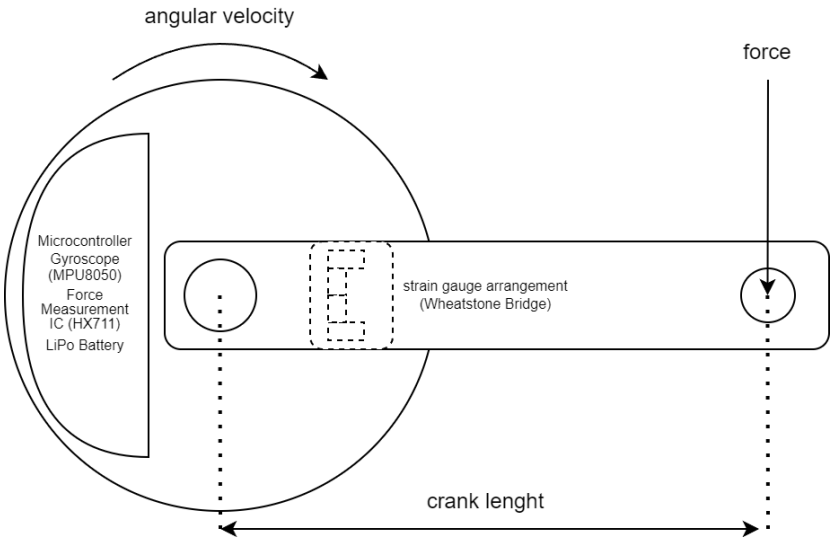


Figure 2: Basic buildup of the power meter.

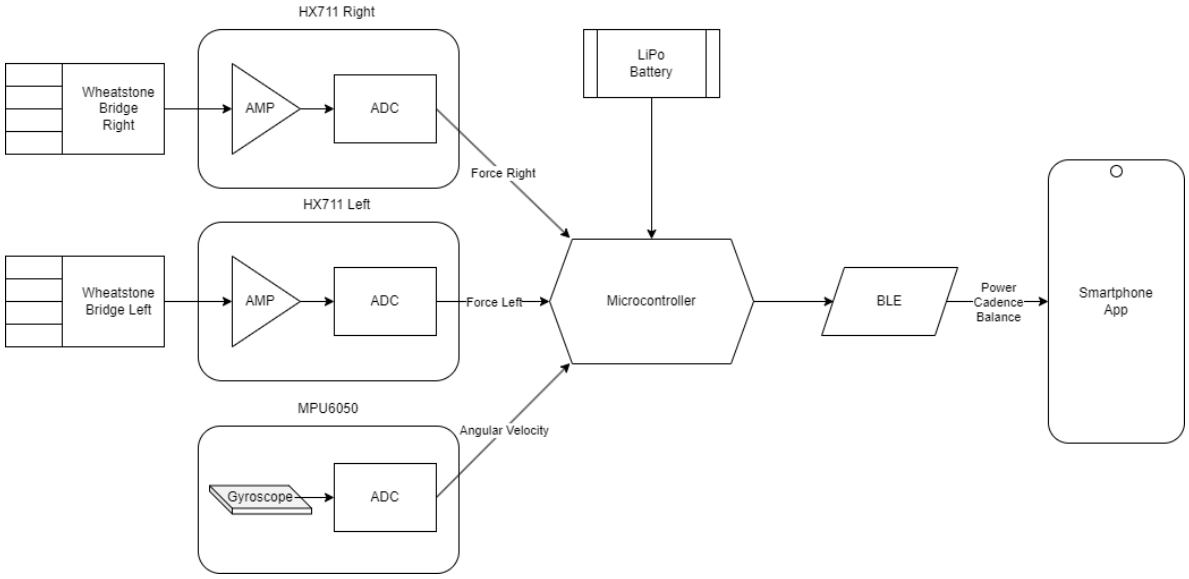
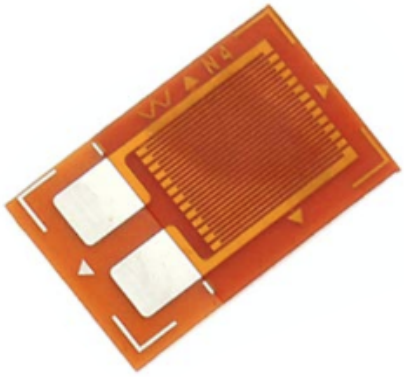


Figure 3: Signal path of the device.

The two wheatstone bridges are each connected to their respective HX711 board, which are further connected to the microcontroller. Data from the HX711s is transmitted via SPI. Also the MPU6050 board is connected via I^2C to the microcontroller board. The latter is the energy distributor for all the boards, and gets its power from a LiPo cell. Processed data is sent to a smartphone using BLE.

2.2 Wheatstone Bridge Configuration and Strain Gauges



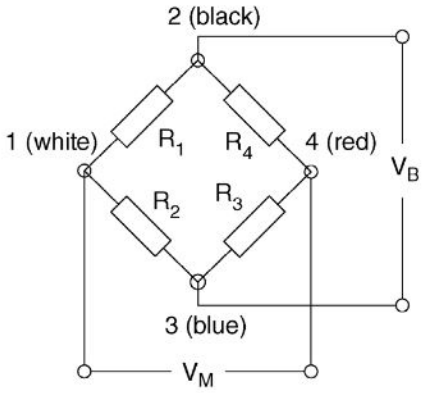


Figure 4: BF350-3AA strain gauge. (left)

Figure 5: The wheatstone-bridge circuit. (right)

The strain gauge is composed of two polymer layers with a conducting layer in between them (see Fig. 4). It is used to measure strain on an object by gluing it on the surface of the object with an appropriate adhesive like 2K-Epoxy. The fundamental principle of the strain gauge is that the changes in the strain on the surface of the object are passed to the electrical conductor. Because its resistance R is dependent on its resistivity ρ, its length l and its cross sectional area A are all altered when strain is applied (Equ. 4). The strain on the object can be determined highly precisely from the change in resistance [7].

$$R = \rho \frac{l}{A} \tag{4}$$

The Wheatstone bridge circuit is used to make the resistance changes measurable. It is made up of four bridge arms with the resistances R_1 to R_4 (see Fig. 5). The effect of the circuit is that if it is excited with a voltage V_B, the voltage V_M, that is the output voltage, is dependent on the difference of the ratios of the resistances (see Equ. 5). This effect can be utilized by exchanging one or more of the resistors in the circuit with a strain gauge. As a result, the output voltage is then a measure of the strain. If the voltage V_B is applied to points 2 and 3 of the bridge circuit (see Fig. 5), then the bridge output voltage V_M between points 1 and 4 is zero when identical resistances are present in all four arms, that is, if R_1=R_2=R_3=R_4. The bridge is therefore said to be balanced. Applying basic principles of electrical engineering, specifically Kirchhoff's Laws, to the Wheatstone bridge circuit yields the correlation [7]:

$$V_M = V_B \left(\frac{R_1}{R_1 + R_2} - \frac{R_3}{R_3 + R_4} \right) \tag{5}$$

To negate strains caused by thermal expansion to mess with the measurements, two strain gauges with a 90 degree angle between them were used as R_1 (longitudinal to the crank) and R_2 (perpendicular to the crank). This works because if R_1=R_2 and

R_3=R_4 the bridge stays balanced, so V_M is zero. And to further increase the V_M and therefore increase the accuracy, the same was done on the opposite side of the crank with two more strain gauges as R_3 (longitudinal to the crank) and R_4 (perpendicular to the crank) (see Fig. 7). This has the effect that while the resistance of R_1 increases, the resistance of R_3 decreases, which yields higher V_M. And even more importantly cancels out strains resulting from stretching the crank longitudinally. This way, a more isolated measurement can be taken. All this was done to both bike cranks on the inside of them as shown in Figure 6. A custom PCB was used to make the cableing of the strain gauges easier, which was also glued to the crank. All of the strain gauges used are BF350-3AA (see Fig. 4) with a nominal resistance of 350 ±0.1 Ohm because they are cheap and widely available. They were glued on, by roughening up the surface of the crank with sandpaper, cleaning the area with acetone and then using conventional 2K-Epoxy (UHU plus endfest 300). To hold them temporarily in place until the glue dried adhesive tape was used. After the glue dried the tape was removed and everything was covered with a small transparent 3D printed shield secured with T-7000 glue to protect it from water and dust. T-7000 from the company Bulaien is a water resistant, solvent based glue that stays rubber like after drying and is often used for smartphone repairs. The rubber-like property should prevent the glue from interfering with the measurements.

Figure 6: Wheatstone bridge position.

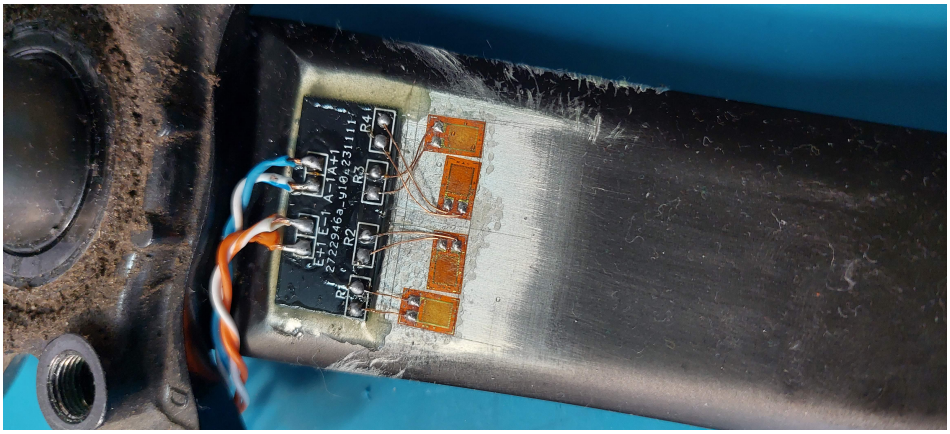


Figure 7: Strain gauge arrangement.

2.3 HX711: a load cell amplifier

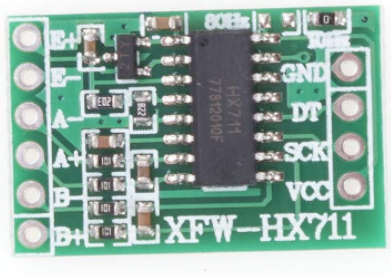
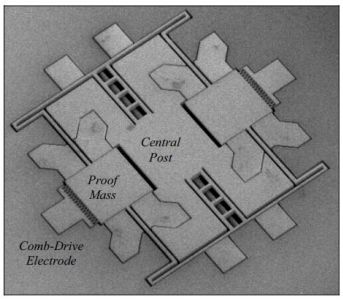


Figure 8: The XFW-HX711 breakout board I used.

The XFW-HX711 breakout board houses the HX711 chip which is an Analog-Digital-Converter specifically made for the use with strain gauges in weight scales [8]. It puts out the exciter voltage V_B between its E+ and E- pins and measures V_M with its A+ and A- pins. The Vcc and GND pins are for supplying power to the board and DT and SCK are for Serial Interface data transmission to a micro controller. The chip was chosen because of its wide availability, its low cost and its easy implementation. Specifically this breakout board was chosen because of the smaller form factor, compared to its competitors and the options to change from 10Hz to 80Hz measurement by just changing the position of a zero Ohm resistor on the board. As a Full Wheatstone Bridge is on each crank two HX711s were needed to measure and transmit the data.

2.4 MPU6050: a MEMS gyroscope



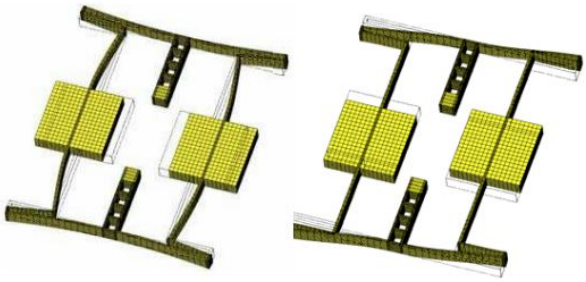


Figure 9: Scanning electron microscope image of a MEMS gyroscope [9]. (left)

Figure 10: Movement simulation of a MEMS gyroscope [9]. (right)

A MEMS (Micro-Electro-Mechanical Systems) gyroscope is a sensor designed for measuring angular velocity or rotational movement. It is a compact apparatus that utilizes micro fabrication techniques to craft exceptionally tiny mechanical components on a silicon base. MEMS gyroscopes find widespread use in a range of applications, including consumer electronics like smartphones, drones, and numerous other devices.

The device's functionality relies on the way a typical tuning fork reacts to rotation. To elaborate, the proof-masses are vibrated at their resonant frequency along the x-axis through the use of comb-drive electrodes. Simultaneously, the Coriolis acceleration,

generated by rotation around the z-axis, is detected capacitively along the y-axis. You can see these resonant modes for both driving (not rotating) and sensing (rotating) in Figure 10 [9].

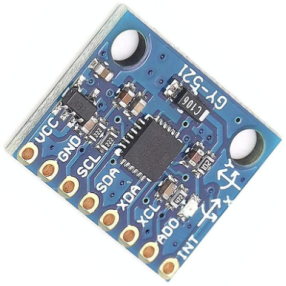


Figure 11: MPU6050 breakout board that was used.

The MPU6050 Chip on the GY-521 breakout board (see Fig. 11) was used, as it is widely available, affordable, compact, and supports I²C communication on its SCL and SDA pins with micro controllers. It houses a 3-axis gyroscope a 3-Axis accelerometer and a temperature sensor. In this project only the Z-axis gyroscope was used.

2.5 Microcontroller: ESP32 and the lolin32 board

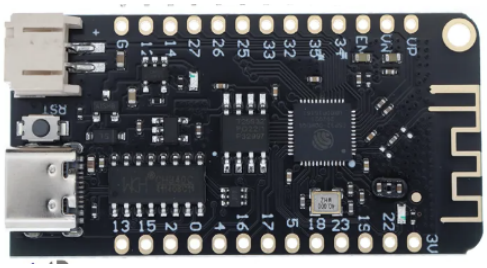


Figure 12: The lolin32 board that was used. (left)

Figure 13: The LiPo battery that was used. (right)

The ESP32 is a micro controller, manufactured by Espressif. It is widely used in the maker community and therefore is very well documented. It supports being programmed via the Arduino IDE. Its power consumption is low, therefore its is very well suited for mobile use with batteries. It has WiFi and Bluetooth capabilities which are essential to the project. The ESP32 also supports I²C and Serial communication, which are needed for the MPU6050 and the HX711 [10].

The lolin32 board (see Fig. 12) was chosen, which houses a ESP32 chip, because of its integrated Lithium-Polymer Battery charging circuit, a USB Type-C connector for charging and connecting to the computer and the integrated antenna. As for the power

supply, a 1100mAh 3.7V LiPo battery from the supplier MakerFocus (see Fig. 13) was used, which according to the tests that were done should yield 8-9h of run time. The batteries came with a JST1.25 plug, but the lolin32 has a JST2.0 soldered to it, so a tiny custom pcb was designed which acts as an adapter to fit a JST1.25 plug on the lolin32 board (see Fig. 14). Because of the smaller footprint of the JST1.25 plug it was possible to also fit a small switch to power off the device without unplugging the battery.

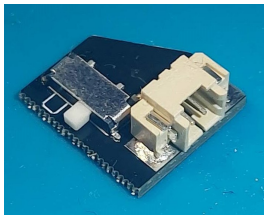


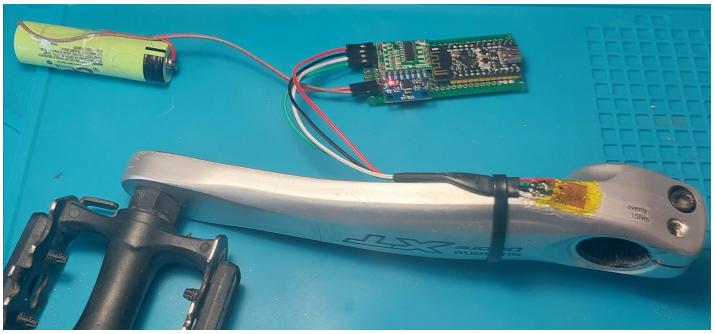
Figure 14: Adapter PCB for the JST1.25 plug.

2.6 Bluetooth Low Energy

To connect the ESP32 to a smartphone or bike computer BLE (Bluetooth Low Energy) was utilized. Its, as the name implies, a version of Bluetooth that uses less energy and is commonly used in commercial bike power meters. For the ESP32 to behave like a commercial power meter and make it therefore able to connect to commercial apps and bike computers the Bluetooth GATT Profiles had to be implemented. Its a way of telling your device what it is and advertising that to other devices, in this case a power meter and not for example a heart rate monitor. For this the Cycling Power Service profile was used. In this project it transmits total power, Cadence and Pedal Power Balance but it can also transmit Accumulated Torque and more [11].

3 The DIY Powermeter

3.1 Hardware



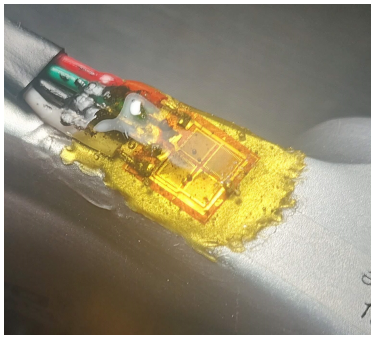


Figure 15: The first semi functional prototype. (left)

Figure 16: Strain gauge for torsion glued on. (right)

Figure 17: The second semi functional prototype.

Before arriving at the final design, several prototypes, that only measure the power from one crank, were built. It was assumed that both legs are of equal strength, so the power measured on the left crank was doubled. Of course in reality the leg strengths can vary, so in the final version both cranks are measured separately to increase accuracy and be able to monitor recovery of a single leg after injury. All the prototypes had a major flaw, that is ruled out in the final version, due to the use of a single special strain gauge, that actually contains four strain gauges, configured in such a way that it only senses torsion on the crank. Depending where the foot is positioned on the pedal, the torsional strains vary, which makes the measurements highly inaccurate. This special strain gauge was used before discovering its properties. Although those prototypes were flawed and only semi functional, they helped resolve software errors and proofed the feasibility of realising this project.

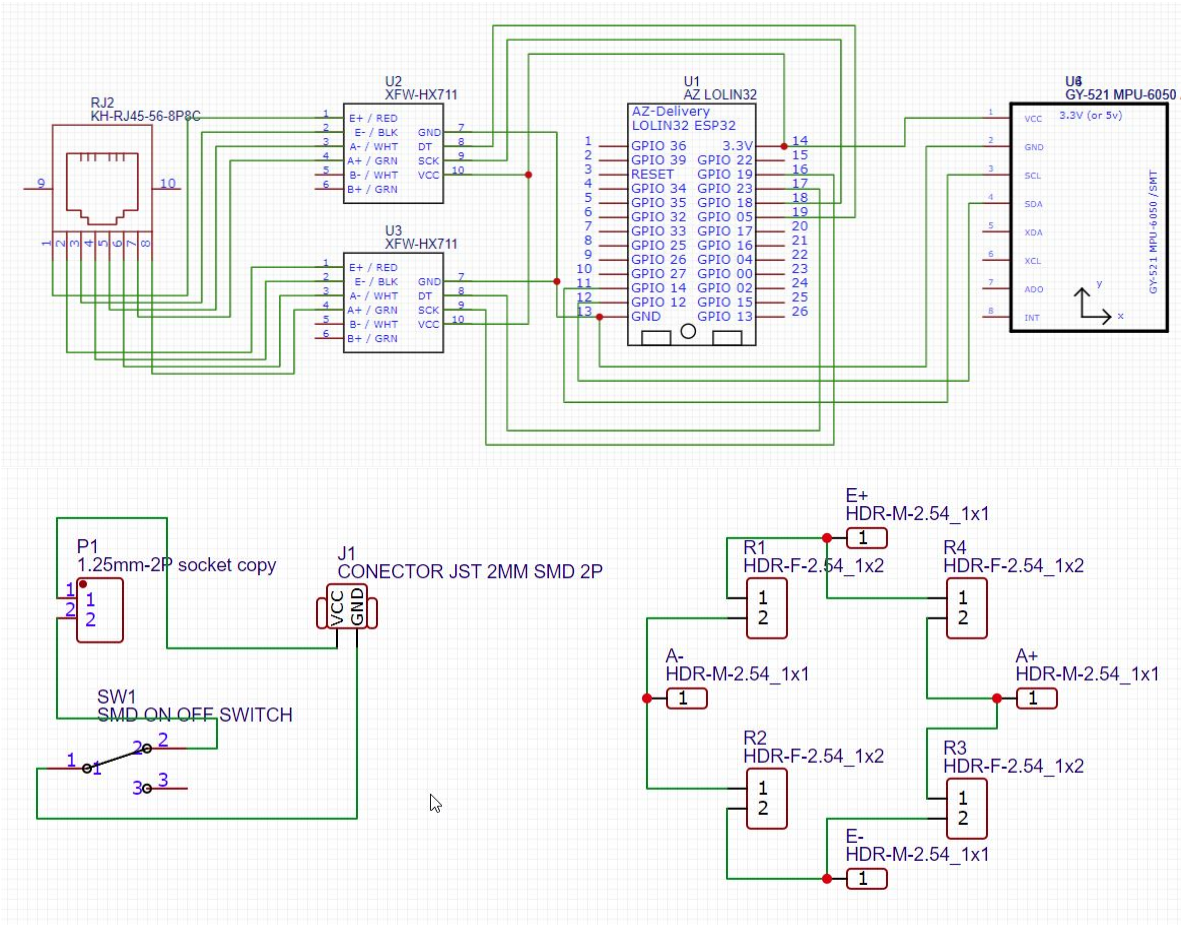


Figure 18: The circuits of the custom PCBs.

To fit all the above mentioned components on the bike crank and make connecting them easier, several custom PCBs (Printed Circuit Board) had to be designed. For that, a free browser based program was used, called EasyEDA. All the parts and their footprints could be found in their library. As shown in Figure 18, all the parts were connected with traces, and also a KH-RJ45-56-8P8C connector (Fig.19) was added to be able to connect and disconnect the components to the crank for recharging and software updating purposes. All the breakout boards are actually meant to be soldered with through hole pins. To save space, they were just soldered to the surface of the PCB similar to a surface-mounted device (SMD) (see Fig.19). Because a sleep function was added after manufacturing, which needed a connection to the interrupt pin of the MPU6050, a thin white cable was used to do this. Also for connecting the strain gauges into a full wheatstone bridge a PCB was designed which was glued right next to them (see Fig. **??**). All those circuits where put on a single PCB layout separate with perforated brake lines to lower the cost of manufacturing.

Here is a Link to the EasyEDA project where all the circuits and PCBs are available: https://u.easyeda.com/join?type=project&key=02087e1d5ad61624d9c4c704eb395e2c&inviter=9d218eaaf44449929435ec07b760872c

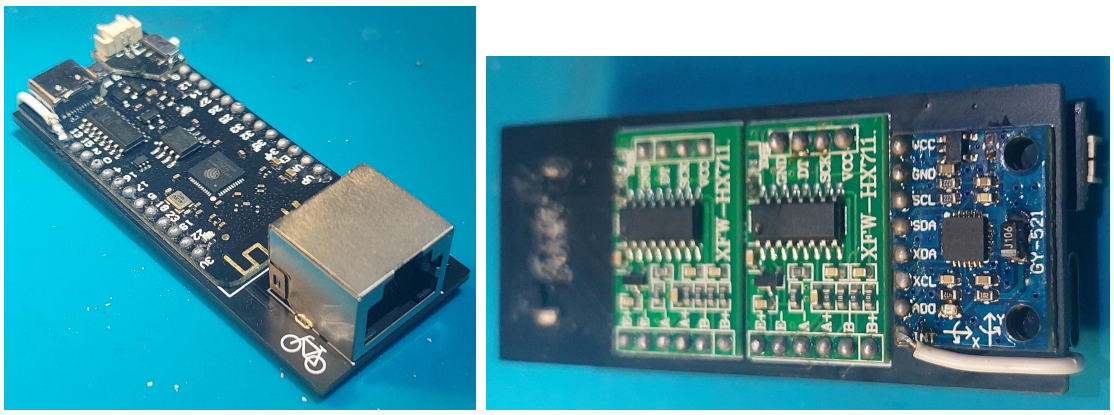


Figure 19: Both sides of the populated main PCB.

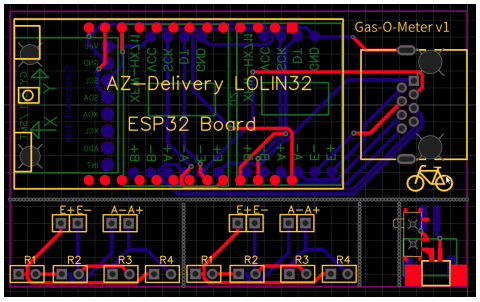


Figure 20: Layout of all the circuits on a single PCB

To house the PCB with the components and the battery a case was designed in CAD that has the CAT-5 cable integrated which provides a stable connection between the PCB and the Wheatstone Bridges due to its spring loaded pins. The case has a built in sliding mechanism, held by the locking latch mechanism of the CAT-5 cable, that allows easy removal of the components. An Fused Deposition Modeling (FDM) 3D-Printer was used to print the case out of polylactic acid (PLA) and it was coated with 2K-epoxy to make it waterproof. A waterproof button mechanism was 3D printed out of thermoplastic polyurethane (TPU) and translucent PLA to access the latch of the CAT-5 cable and to see the status of the LEDs on the PCB. The whole case is mounted to the chainring-crank assembly with cable ties. To connect the main PCB, which is on the right side, to the strain gauges of the left crank, advantage was taken of the Shimano Hollowtech II crank design, which features, as the name implies, a hollow crank axle.

Here are the links to the cases of second prototype(Shimano Ultegra FC-6603/6600):
https://cad.onshape.com/documents/ecfcbcda4d22d249b0cde56d/w/63fb9b1391ca16029c94dcfc/e/4f803ae52c2a9bda11a028a5

and also the final version of the power meter(Shimano Deore XT FC-M8000-B2):
https://cad.onshape.com/documents/693be32f2a318e7374611608/w/2872cf71e40c9873600a5613/e/dcaf888a125a19a23b3fab52

As the final version is tailored to the crank set model I owned it is unlikely to fit on other bicycles. If there is enough space (min. 20mm) between the crank and the chain stays the model from the second prototype should work for most cranks. The shift from

one crank system to another was done because the bicycle frame used for the first two prototype broke during the development.

3.2 Software

The Code that controls the ESP32 micro controller was written in the Arduino IDE. All the libraries used can easily be downloaded in the IDE itself. After setting up connection to the sensors and to a device via bluetooth it starts reading data from the sensors and calculates power P out of angular velocity ω, force F and crank length l with the following formula:

$$P = Fl\omega \tag{6}$$

Power is first calculated separately for the right and left crank to be able to calculate Pedal Power Balance. Only later they added up to get the total power. To be able to set the crank arm force scales to zero while on the bike, a feature was added that does this after back pedaling five revolutions. This is also a common feature in commercial power meters where it is often falsely called calibration. Also a deep sleep function was implemented to reduce the battery consumption while not in use. After one minute of no crank movement it enters deep sleep mode. Once the MPU6050 detects movement it sends an interrupt that wakes the ESP32 up which then automatically reestablishes the BLE connection and continues measuring. This decreases power consumption from 0.45W while in use to 0.13W while in deep sleep according to the measurements. To be able to get raw data from the power meter for custom analysis an easily customizable app was made with the browser based MIT app inventor 2. To access this functionality another code has to be uploaded to the ESP32 as the app does not use Cycling Power Service. The main code is in the appendix.

All the codes are freely available on my GitHub: https://github.com/Leckogrande/diy-power-meter/tree/main

3.2.1 Pseudocode

```
import libraries

define BLE characteristics
define BLE features

create objects for the scales and the mpu6050
create variables and constants
define pins

void setup() {
  start Serial

  set up scales

  start I2C connection
  test I2C connection
  set up MPU6050
```

```
  set up the interupt for sleep mode

  set up BLE and Cycling Power Service
}

void loop() {
  zero the scales

  while (BLE connected) {
    get angular velocity from MPU6050
    get force applied to left and right pedal from the scales

    if (no angular velocity detected for defined time){
      go sleep mode
    }

    count revolution using angular velocity and time

    if(reverse pedal revolution >=5){
      re-zero scales
    }

    ignore negative angular velocity data

    calculate power for left and right pedal
    add up left and right power to get total power

    ignore negative left and right power values
    calculate pedal power balance

    ignore negative total power values
    ignore total power values bigger than 1500

    correct power data using correction function
    derived from linear regression model

    ignore power data when rpm is very low or power is very low

    Serial.print data for troubleshooting

    put flags, power, revolution and pedal balance data
    in the data packet string
    send this string via BLE

  }
}
```

3.3 Calibration

Once the assembly was finished each crank had to be calibrated separately with known weights. A luggage scale was used to do this. It was tied with nylon rope to the saddle and the crank was put into 3 o'clock position on the left side and 9 o'clock position on the right to mimic the load that would be applied while pedaling (see Fig.22). If the back wheel is turned backwards by hand, the tension on the luggage scale increases. The calibration was done with a calibration example code in the HX711 Arduino Library. To check the accuracy of the calibration, different weights were applied using the method explained above. As the maximum weight of the luggage scale was 40kg no higher weight could have been applied. Ideally the accuracy should be checked up to around 80-100kg (rider weight), because this is approximately the highest weight a rider could put on the pedal. The behavior was sufficiently linear on both cranks with an average deviation of 1.1% on the right pedal and 1.8% on the left (see Tab. 1). These deviations might be due to holding the wheel by hand and keeping the tension constant this way while reading the scale.

Force applied[N]	Measurement R[N]	Deviation R[%]	Measurement L[N]	Deviation L[%]
9,8	9,6	-2,1	9,5	-3,2
24,5	26,2	6,8	25,1	2,3
49,1	53,1	8,3	49,6	1,1
73,6	72,3	-1,7	75,2	2,2
98,1	104,3	6,3	100,2	2,1
122,6	128,2	4,5	126,3	3,0
147,2	153,6	4,4	151,7	3,1
171,7	169,9	-1,0	177,5	3,4
196,2	202,1	3,0	199,2	1,5
220,7	224,3	1,6	225,2	2,0
245,3	248,3	1,2	250,1	2,0
269,8	264,7	-1,9	272,2	0,9
294,3	287,4	-2,3	300,3	2,0
318,8	315,5	-1,0	325,1	2,0
343,4	338,1	-1,5	349,5	1,8
367,9	356,8	-3,0	375,8	2,2
392,4	383,7	-2,2	397,9	1,4

Table 1: Force calibration data.

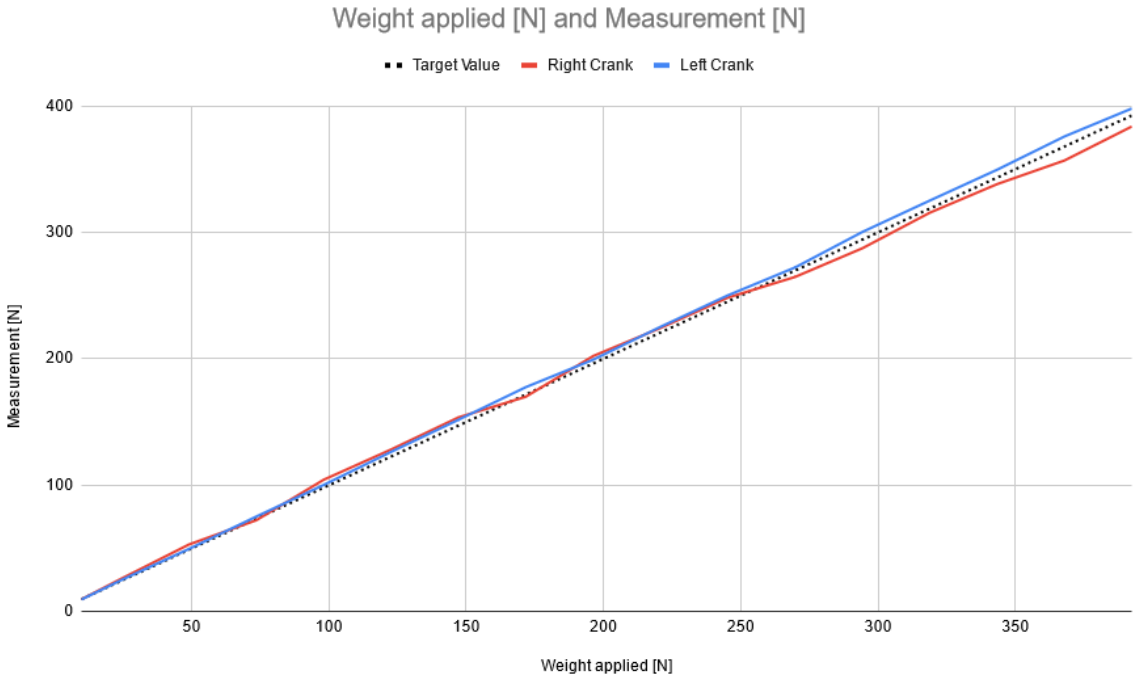


Figure 21: Calibration graph.

Figure 22: Calibration method with luggage scale.

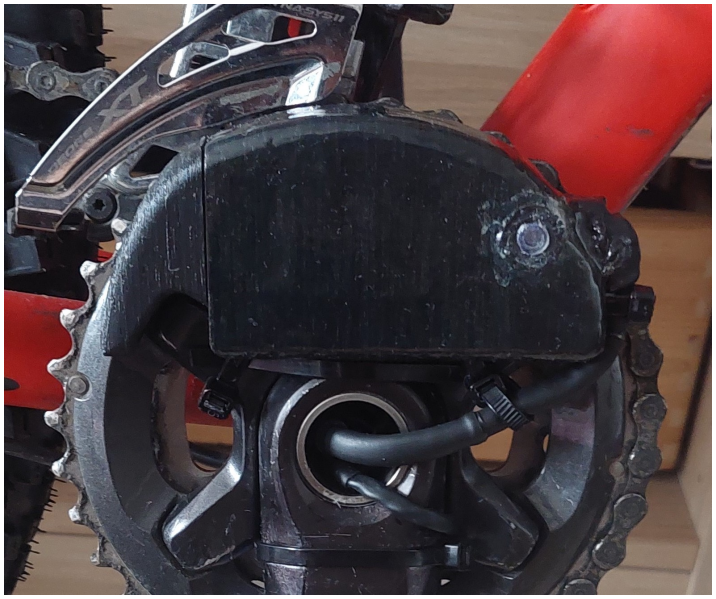


Figure 23: Finished case mounted to the crank assembly.

The power meter was then compared to a commercial one to further validate and increase accuracy. The Zwift Hub One was used for this. It has a claimed accuracy of ±2.5% [12]. The software Zwift, a virtual cycling game, was used to create several test scenarios to compare measurements at different power and crank rotation speed. Three tests were done at 60, 80 and 100 RPM. Each test started with 50W and went as high as 500W (see Fig. 24).

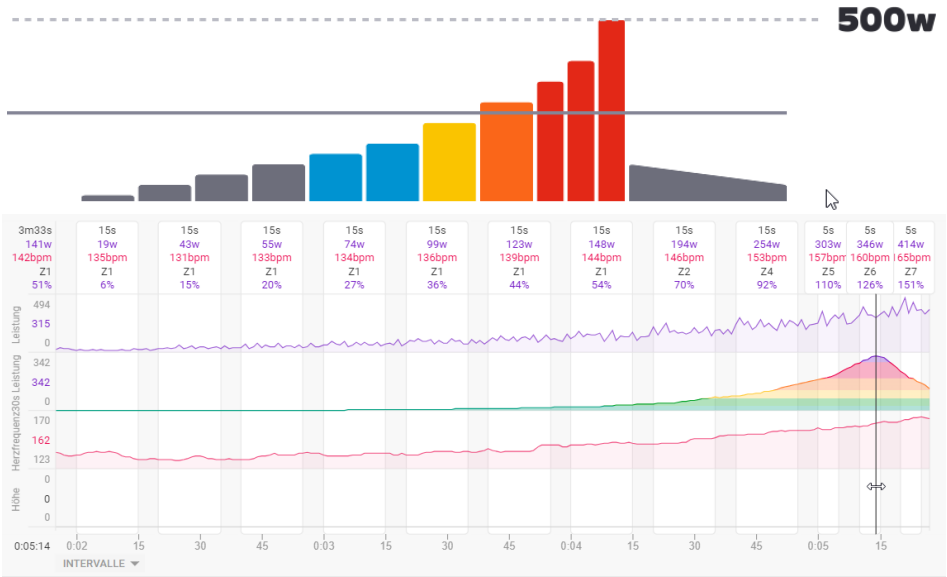


Figure 24: Zwift calibration preset and intervals.icu analysis.

The Zwift Hub One was set in ERG mode which causes it to automatically change resistance according to the preset power and cadence so the final power is constant. So if the training preset is set to 100W and the cyclist pedals at a constant 80rpm it changes the tension on the chain so ultimately 100W are demanded by the cyclist.

This behavior is not perfectly instantaneous, so the first 5 seconds, after an increase in power by the preset, were discarded. The analysis of the recorded data was done on the website intervals.icu with its intervals feature.

These measurements are shown in table 2 and in the diagram 25. As seen in this data the power meter is underestimating the actual power by a huge margin which makes it too inaccurate for the desired task. This is probably due to the frequency of the measurements being too low. That would also explain why the power meter is more accurate at lower cadences. To correct for this non-linear errors, two polynomial regression models were made. Out of those, two correction equation were derived to correct the inaccuracies. Model 2 was made because due to its polynomial of the power of 2, it would be better suited to correct for non-linearities. But as it was not performing significantly better than Model 1 and was due to its polynomial more prone to over fitting, it was discarded.

P should $[W]$	P measured @ 60rpm	P measured @ 80rpm	P measured @ 100rpm
50	27	19	12
75	51	43	27
100	71	55	43
125	98	74	68
150	116	99	88
175	145	123	106
200	164	148	131
250	205	194	177
300	253	254	230
350	309	303	259
400	356	346	307
500	446	414	380

Table 2: Accuracy before correction.

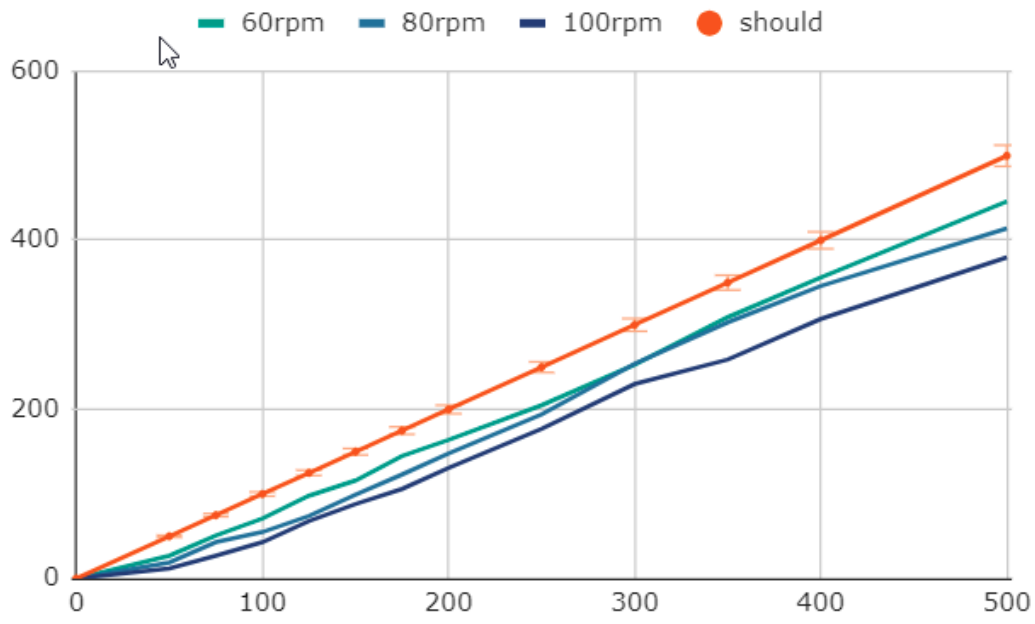


Figure 25: Accuracy before correction.

Model 1 (R^2 score of 0.993)	
P_actual=c1*P_measured+c2*RPM+c3*P_measured*RPM	
c1:	0,8253
c2:	0,41861
c3:	0,00372
Model 2 (R^2 score of 0.997)	
P_actual=b1*P_measured+b2*RPM+b3*$P\_measured^2$+b4*RPM^2+b5*P_measured*RPM	
b1:	0,91327
b2:	0,28955
b3:	0,00001113
b4:	0,0017276
b5:	0,0023779

Table 3: Correction equations derived by polynomial regression.

P measured @ 60rpm	P measured @ 80rpm	P measured @ 100rpm	Deviation [%] @ 60rpm	Deviation [%] @ 80rpm	Deviation [%] @ 100rpm
53,4261	54,8239	56,2286	6,8522	9,6478	12,4572
78,5901	81,7735	74,1881	4,7868	9,031333333	-1,082533333
99,5601	95,2483	93,3449	-0,4399	-4,7517	-6,6551
127,8696	116,5834	123,2774	2,29568	-6,73328	-1,37808
146,7426	144,6559	147,2234	-2,1716	-3,562733333	-1,851066667
177,1491	171,6055	168,7748	1,228057143	-1,939714286	-3,557257143
197,0706	199,678	198,7073	-1,4647	-0,161	-0,64635
240,0591	251,3314	253,7831	-3,97636	0,53256	1,51324
290,3871	318,7054	317,24	-3,2043	6,235133333	5,746666667
349,1031	373,7275	351,9617	-0,2562571429	6,779285714	0,5604857143
398,3826	422,0122	409,4321	-0,40435	5,50305	2,358025
492,7476	498,3694	496,835	-1,45048	-0,32612	-0,633

Table 4: Corrected data with Model 1.

P measured @ 60rpm	P measured @ 80rpm	P measured @ 100rpm	Deviation [%] @ 60rpm	Deviation [%] @ 80rpm	Deviation [%] @ 100rpm
52,11096177	55,19119593	60,04532272	4,22192354	10,38239186	20,09064544
77,47445313	81,69180537	77,31773377	3,29927084	8,92240716	3,090311693
98,62049033	94,94691825	95,74715937	-1,37950967	-5,05308175	-4,25284063
127,1817645	115,9407359	124,5545451	1,745411616	-7,247411296	-0,356363904
146,2316293	143,5764231	147,6104707	-2,512247147	-4,28238458	-1,59301952
176,9382483	170,1197718	168,3684167	1,107570429	-2,788701846	-3,789476183
197,0665285	197,7827275	197,2108619	-1,46673576	-1,10863624	-1,394569035
240,5286183	248,7189167	250,3173118	-3,7885527	-0,512433328	0,126924708
291,4586122	315,2282111	311,563577	-2,847129277	5,07607036	3,854525667
350,9417595	369,6035802	345,1021515	0,2690741514	5,601022906	-1,399385277
400,9189957	417,3647711	400,6554114	0,22974892	4,34119277	0,1638528425
496,7573191	492,9781055	485,240972	-0,648536184	-1,404378904	-2,9518056

Table 5: Corrected data with Model 2.

4 Results

The same tests as in the calibration were repeated, and the following data was recorded.

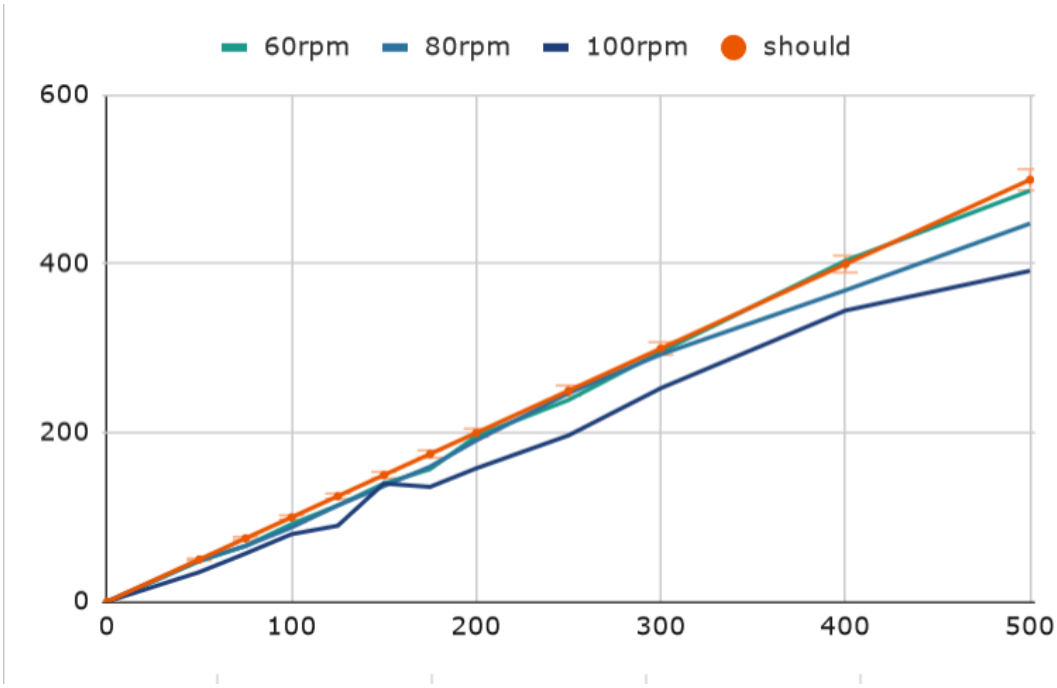


Figure 26: Accuracy after correction.

P should [W]	P measured @ 60rpm	P measured @ 80rpm	P measured @ 100rpm
0	0	0	0
50	48	49	35
75	66	66	57
100	92	88	80
125	114	114	90
150	140	137	140
175	157	160	136
200	196	191	158
250	239	247	197
300	296	293	253
400	404	369	345
500	487	448	392

At 100 rpm the measurements are still far off from the commercial power meter, as visible in Figure 26. At 60 and 80 rpm the model generalizes way better. The inaccuracys at high cadences can be attributed to the measuring frequency of the HX711 which is probably too low.

Another test was done where power was not increased incrementally and also the cadence was not fixed to a certain value.

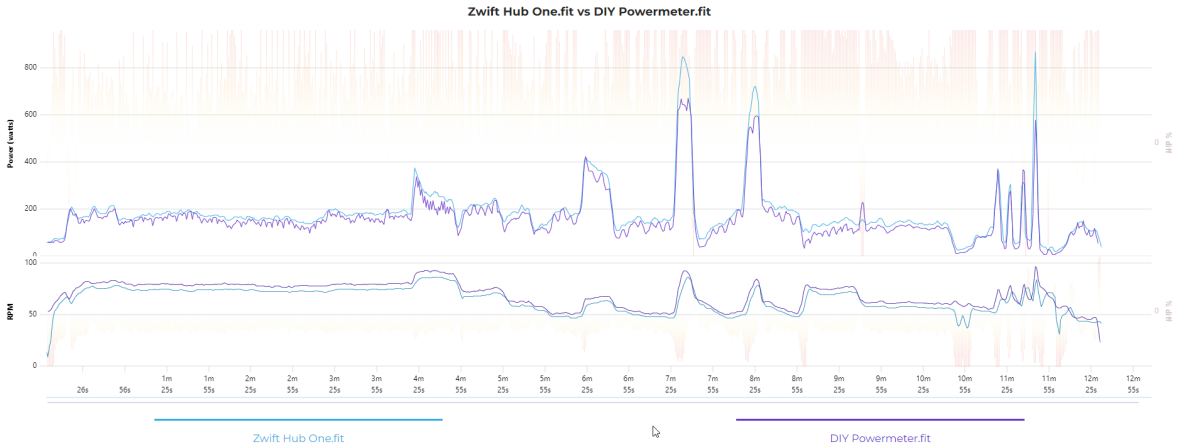


Figure 27: DIY Powermeter compared to the Zwift Hub One.

As also measured in the tests before it underestimates power, on average 15.41% over the mixed test shown in the first graph in Figure 27. The second graph shows that it also measures about 3-4 rpm higher cadence than the commercial product. This could be due to the Zwift Hub One using an algorithm that looks at the fluctuations in power caused by the inconsistent way our feet apply force to the pedal and derives cadence from that but also due to the MPU6050 chip and the way cadence was calculated using it.

5 Conclusion

Due to the HX711 chips measuring frequency of 80Hz the power meter is only accurate at medium to low cadences and even there lacks the accuracy of good high quality devices. This concludes that the power meter is not accurate enough for elite cyclists or lab/science applications, but it is suitable to give a hobbyist more insight about training progress and injury recovery. Nevertheless, the validity of the project stands since making an accurate power meter is not a trivial task, and even big companies like Shimano fail at it. This can be seen in recent in-depth reviews done on their R9200P power meter, which still gets used by professional cycling teams sponsored by Shimano [13]. The DIY solution has inaccuracies of similar magnitude at low cadences as the R9200P, but performs worse at higher cadences due to the crucial mistake of choosing the HX711 chip. It seems that even with linear regression it was not possible to correct that in the software appropriately. The ADS1115 chip would have been a better choice, as it has a sampling rate of 860 SPS [14]. Also the battery life is still too short, as it is only enough for 1-4 sessions. This could be further optimized in the software but ultimately hardware changes need to be done to get it to the duration of commercial devices. Replicating this project can only be recommended to makers which are looking for an interesting build to learn from and spend at least 2-3 whole days on, but not to cyclists inexperienced with electronics, which are just looking for a cheap alternative to the commercial products.

References

[1] B. SASAKI, "Bicycle pedal," Patent US 2016/0 052 583 A1, 2016.

[2] J. Meyer, "Crankset based bicycle power measurement," Patent US 8,006,574 B2, 2011.

[3] G. M. Hauschildt, "Portable power meter for calculating power applied to a pedal and crank arm based drive mechanism and a method of calculating the power," Patent US 2007/0 245 835 A1, 2007.

[4] S. L. Halson, G. I. Lancaster, A. E. Jeukendrup, and M. Gleeson, "Immunological responses to overreaching in cyclists," 2003.

[5] S. A. Jobson, L. Passfield, G. Atkinson, G. Barton, and P. Scarf, "The analysis and utilization of cycling training data," 2009.

[6] Science2Sport. (2018) Monitoring your training load. [Online]. Available: https://bikehub.co.za/news/monitoring-your-training-load-r7477/

[7] P. D.-I. S. Keil, "High performance matched-mode tuning fork gyroscope," 2017.

[8] *24-Bit Analog-to-Digital Converter (ADC) for Weigh Scales*, Avia Semiconducter, 2012.

[9] M. Zaman, A. Sharma, and F. Ayazi, "High performance matched-mode tuning fork gyroscope," 2006.

[10] *2.4 GHz Wi-Fi + Bluetooth® + Bluetooth LE SoC*, Espressif Systems, 2023, version 4.3.

[11] *Cycling Power Service Bluetooth® Service Specification*, Bluetooth SIG Proprietary, 2016, cPS v1.1.

[12] E. Schlange. (2022) Zwift hub smart trainer: An insider review. [Online]. Available: https://zwiftinsider.com/zwift-hub-review/

[13] D. Rainmaker. (2023) Shimano r9200p power meter in-depth review: Astonishingly inaccurate. [Online]. Available: https://www.dcrainmaker.com/2023/02/shimano-r9200p-power-meter-review-astonishingly-bad.html

[14] *ADS111x Ultra-Small, Low-Power, I2C-Compatible, 860-SPS, 16-Bit ADCs With Internal Reference, Oscillator, and Programmable Comparator*, Texas Instruments, 2018, datasheet.

6 Appendix

6.1 Code

```
//import all librarys necessary
#include "BLEDevice.h"
#include "BLEUtils.h"
#include "BLEServer.h"
#include <BLE2902.h>
#include "Arduino.h"
#include <Adafruit_MPU6050.h>
#include <Adafruit_Sensor.h>
#include <Wire.h>
#include "HX711.h"
#include <esp_sleep.h>


//part of the code is commented to be able to set up WebSerial if needed to
    debug without wired connection
/*
#include <WiFi.h>
#include <AsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include <WebSerial.h>

AsyncWebServer server(80); // WebSerial is accessible at "<IP
    Address>/webserial" in browser

const char* ssid = "xxxxx"; // Your WiFi SSID
const char* password = "xxxxx"; // Your WiFi Password
*/


// Define the UUIDs for the services and characteristics
#define CYCLING_POWER_SERVICE_UUID    "1818"
#define CYCLING_POWER_MEASUREMENT_UUID "2A63"
#define CYCLING_POWER_FEATURE_UUID    "2A65"
#define SENSOR_LOCATION_UUID          "2A5D"

// Flags (including the Cadence and Pedal Power Balance)
uint16_t flags = (1 << 0) | (1 << 1) | (1 << 5);
// Define the values for the Cycling Power Feature and Sensor Location as
    constants
const uint32_t CYCLING_POWER_FEATURES = (1 << 0) | (1 << 3); // Supports
    Pedal Power Balance and Cadence
const uint8_t SENSOR_LOCATION = 0x01; // Example: Left crank

// Declare BLE Server and Characteristics globally
BLEServer *pServer;
```

```
BLECharacteristic *pCyclingPowerMeasurementCharacteristic;
BLECharacteristic *pCyclingPowerFeatureCharacteristic;
BLECharacteristic *pSensorLocationCharacteristic;


Adafruit_MPU6050 mpu; //creates the object for the MPU6050
HX711 scaleR;       //creates the object for the right scale
HX711 scaleL;       //creates the object for the left scale

//create all variables needed
float crank_lenght = 0.175; //TODO set the crank lenght
float scaleR_calibration_value = 5.099433; //TODO you need to calibrate this
    yourself after mounting the crank to the bike. use the calibration
    example code of the HX711 library
float scaleR_calibration_offset = 4294667228;
float scaleL_calibration_value = -14.402885;
float scaleL_calibration_offset = 4294216973;
float c1 = 0.8253; //TODO you need to compare the powermeter against a
    commercial one using different rpm and power to make a linear regression
    model to get a correction function which uses these variables
float c2 = 0.41861;
float c3 = 0.00372;
#define MOTION_THRESHOLD 2 // Adjust threshold if needed
#define MOTION_DURATION 1 // Adjust duration if needed
int sleep_timer = 120000; //Adjust sleep timer if needed in ms

unsigned char bleBuffer[8];
int16_t power = 0; // in Watts, signed 16-bit
uint8_t balance = 0; // Pedal power balance, 0-100, representing 0%-100%
uint16_t rpm = 0; // Revolutions per minute, 16-bit unsigned
int16_t power_corrected = 0;
float FR = 0;
float Mr = 0;
float PR = 0;
float FL = 0;
float ML = 0;
float PL = 0;

bool deviceConnected = false;
volatile bool movementDetected = true;
volatile uint32_t lastMovementTime = 0;
float nowTime = 0;
float prevTime = 0;
float nowAlpha = 0;
float alpha = 0;
int cal_counter = 0;

uint16_t revolutions = 0;
uint16_t timestamp = 0;
int t_diff = 0;
```

```
int t_prev = 0;
uint32_t start, stop;

//set all pins
#define sdaPin 12
#define sclPin 14
uint8_t dataPinR = 23;
uint8_t clockPinR = 19;
uint8_t dataPinL = 5;
uint8_t clockPinL = 18;

class MyServerCallbacks: public BLEServerCallbacks {
    void onConnect(BLEServer* pServer) {
      deviceConnected = true;
      Serial.println("Device connected");
    };

    void onDisconnect(BLEServer* pServer) {
      deviceConnected = false;
      Serial.println("Device disconnected");
    }
};

void setup()
{
  Serial.begin(115200);

  /*
  //sets up Wifi for WebSerial
  WiFi.mode(WIFI_STA);
  WiFi.begin(ssid, password);
  if (WiFi.waitForConnectResult() != WL_CONNECTED) {
    Serial.printf("WiFi Failed!\n");
    return;
  }
  Serial.print("IP Address: ");
  Serial.println(WiFi.localIP());
  WebSerial.begin(&server);
  server.begin();
  */

  //sets up scales
  scaleR.begin(dataPinR, clockPinR);
  scaleR.set_offset(scaleR_calibration_offset);
  scaleR.set_scale(scaleR_calibration_value);
  scaleR.tare(); //reset the scale to zero = 0
  scaleL.begin(dataPinL, clockPinL);
  scaleL.set_offset(scaleL_calibration_offset);
  scaleL.set_scale(scaleL_calibration_value);
  scaleL.tare(); //reset the scale to zero = 0
```

```
Wire.begin(sdaPin, sclPin); //starts the I2C connection

while (!Serial)
  delay(10); //pause until serial console opens

Serial.println("Adafruit MPU6050 test!"); // Try to initialize connection
    to MPU6050
if (!mpu.begin()) {
  Serial.println("Failed to find MPU6050 chip");
  while (1) {
    delay(10);
  }
}
Serial.println("MPU6050 Found!");
mpu.setGyroRange(MPU6050_RANGE_1000_DEG);
mpu.setFilterBandwidth(MPU6050_BAND_21_HZ);

mpu.setInterruptPinPolarity(false); // sets polaity of int pin
mpu.setInterruptPinLatch(true); //sets latching behavior of int pin
mpu.setMotionInterrupt(true); // enables motion interrupt
mpu.setMotionDetectionThreshold(MOTION_THRESHOLD);
mpu.setMotionDetectionDuration(MOTION_DURATION);

Serial.println("");
delay(100);

esp_sleep_enable_ext0_wakeup(GPIO_NUM_13,1); //defines which GPIO pin will
    receive the Motion Interrupt to wake up



BLEDevice::init("DIY Powermeter");

// Create BLE Server
pServer = BLEDevice::createServer();

// Create BLE Service for Cycling Power
BLEService *pCyclingPowerService =
    pServer->createService(CYCLING_POWER_SERVICE_UUID);

// Create a BLE Characteristic for Cycling Power Measurement
pCyclingPowerMeasurementCharacteristic =
    pCyclingPowerService->createCharacteristic(
                                      CYCLING_POWER_MEASUREMENT_UUID,
                                      BLECharacteristic::PROPERTY_NOTIFY
                                    );
BLE2902 *pCpmDesc = new BLE2902();
pCpmDesc->setNotifications(true);
pCyclingPowerMeasurementCharacteristic->addDescriptor(pCpmDesc);
```

```
  // Create a BLE Characteristic for Cycling Power Feature
  pCyclingPowerFeatureCharacteristic =
      pCyclingPowerService->createCharacteristic(
                                        CYCLING_POWER_FEATURE_UUID,
                                        BLECharacteristic::PROPERTY_READ
                                      );
  pCyclingPowerFeatureCharacteristic->setValue((uint8_t*)&CYCLING_POWER_FEATURES,
      sizeof(CYCLING_POWER_FEATURES));

  // Create a BLE Characteristic for Sensor Location
  pSensorLocationCharacteristic = pCyclingPowerService->createCharacteristic(
                                    SENSOR_LOCATION_UUID,
                                    BLECharacteristic::PROPERTY_READ
                                  );
  pSensorLocationCharacteristic->setValue((uint8_t*)&SENSOR_LOCATION,
      sizeof(SENSOR_LOCATION));

  // Start the service
  pCyclingPowerService->start();

  // Start advertising
  BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
  pAdvertising->addServiceUUID(CYCLING_POWER_SERVICE_UUID);
  pAdvertising->start();

  Serial.println("Cycling Power Meter service started");

}

void loop()
{
  scaleR.tare(); //reset the scale to zero = 0
  scaleL.tare(); //reset the scale to zero = 0
  Serial.println("scales tared and ready to connect...");
  delay(1000);

  while (pServer->getConnectedCount() > 0) {

    sensors_event_t a, g, temp; //creates objects for the MPU6050 data
    mpu.getEvent(&a, &g, &temp); //gets data from MPU6050

    prevTime = nowTime;
    nowTime = millis();
    float dt = nowTime - prevTime;     //time between this measurement and
        the last

    float t = temp.temperature -10;
    float w = g.gyro.z + 0.02;    //reads out angular velocity of the
        z-axis, sets an offset because of sensor drift
```

```
FR = scaleR.get_units(1) /1000 *9.81; //reads out the right scale and
    converts it into newtons
FL = scaleL.get_units(1) /1000 *9.81; //reads out the left scale and
    converts it into newtons

//records time since last movement
if (abs(w) <= 0.1){
  lastMovementTime = lastMovementTime + dt;
} else {
  lastMovementTime = 0;
}

//checks if the set sleeptime has been reached
if ( lastMovementTime >= sleep_timer){
  movementDetected = false;
} else {
  movementDetected = true;
}

if (movementDetected == false){
Serial.println("No movement detected. Going to sleep...");
delay(100); // Delay for a short time to allow any Serial.print()
    messages to be sent
esp_deep_sleep_start(); // Enter deep sleep mode
}

float nowAlpha = w *180/PI * dt /1000; //calulate by how much the angle
    of the crank has changed since last measurement
alpha = alpha + nowAlpha;          //calculates total angle of the crank

if (alpha >= 360.0) {             //if angle of crank reaches 360 degree
    set it back to 0, adds up total revolutions, sets timestamps for
    CyclePowerService
  alpha -= 360.0;
  t_diff = millis() - t_prev;
  revolutions = revolutions + 1;
  timestamp = timestamp + t_diff *1000 /1024;
  t_prev = millis();
  cal_counter = 0;

}

 // Ensure that the timestamp rolls over correctly
timestamp = timestamp % 65536; // 65536 is 2^16, max uint16_t value

//protocol to re-tare the scales
if (alpha <= -360.0) {             //if angle of crank reaches -360
    degree set it back to 0, increases counter until 5 revolutions to
    start re-tareing
  alpha += 360.0;
```

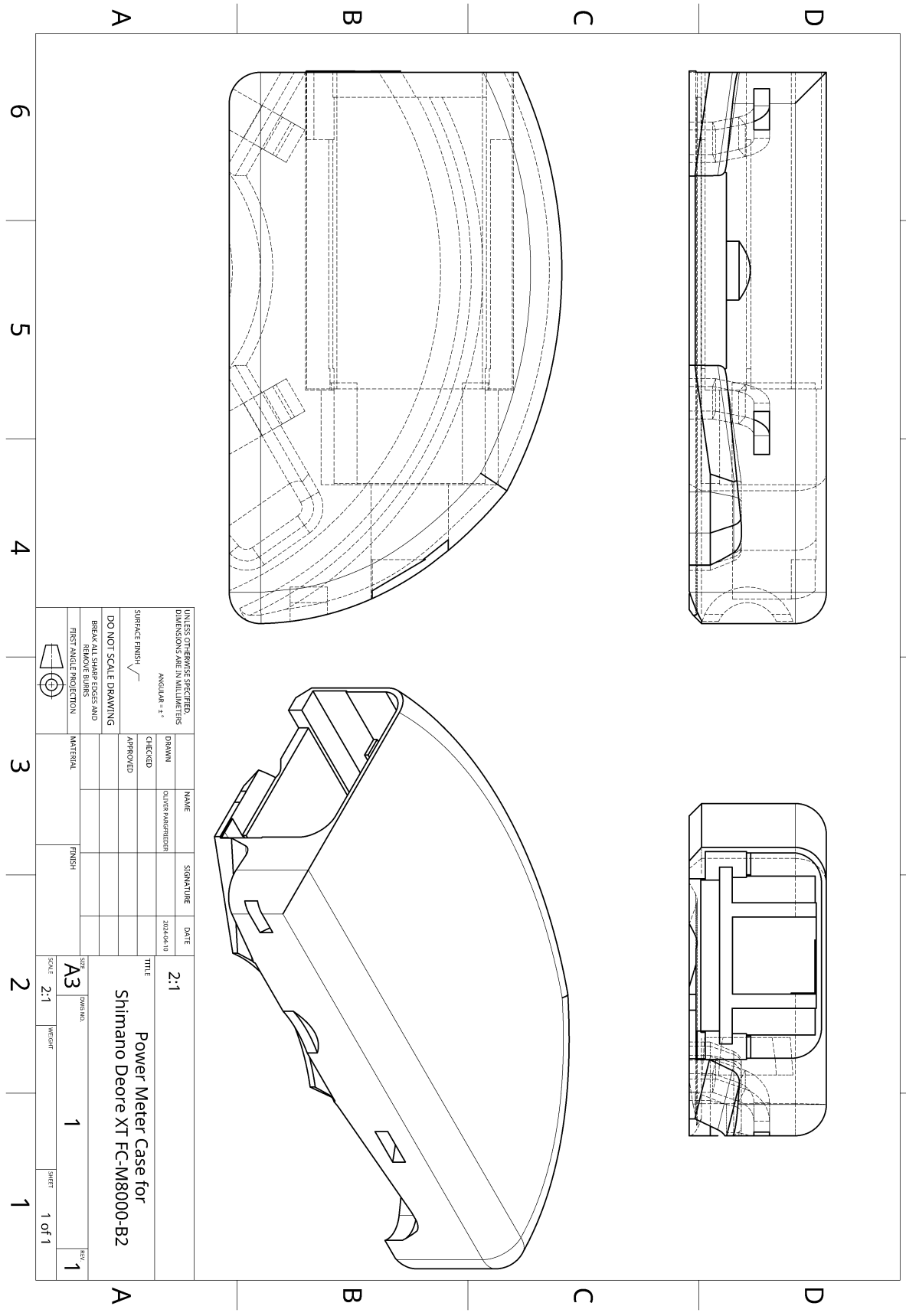
```
  cal_counter = cal_counter + 1;
  if (cal_counter == 5){
    delay(2000);
    scaleR.tare(); //reset the scale to zero = 0
    scaleL.tare(); //reset the scale to zero = 0
    cal_counter = 0;
  }
}


//angular velocity must be positive
if (w<0){
 w = 0;
}

Mr = FR * crank_lenght; //calculates the torque on the right crank
PR =  Mr * w;     //calculates the power of the right crank
ML = FL * crank_lenght; //calculates the torque on the left crank
PL =  ML * w;     //calculates the power of the left crank

power = PL + PR; //add left and right power
rpm = w * 60/(2*PI);

if (PL<0){  //removes negative power values which would cause errorsin
    the power balance
 PL = 0;
}
if (PR<0){  //removes negative power values which would cause errors in
    the power balance
 PR = 0;
}

balance = (PL/(PL + PR + 0.001))*100;

if (power<0){ //removes negative power values which would cause errors
 power = 0;
}
if (power>1500){ //filters measurements caused by bad contact to the
    wheatstone bridges
 power = 1500;
}

power_corrected = (c1*power+c2*rpm+c3*power*rpm); //correction function
    derived from linear regression model

//to correct for possible wrong power numbers due to the correction
    function
if (rpm<5){
 power_corrected = 0;
}
```

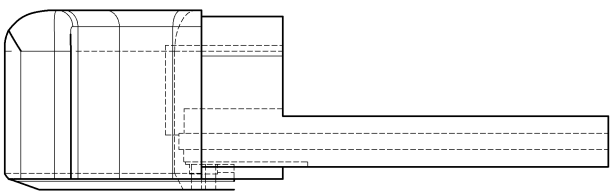
```
if (power<10){
 power_corrected = 0;
}

//prints data for calibration and troubleshooting
Serial.print("Sent Power: ");
Serial.print(power_corrected);
Serial.print("W ");
Serial.print("Balance: ");
Serial.print(balance);
Serial.print("% ");
Serial.print("rpm: ");
Serial.print(rpm);
Serial.print("rpm ");
Serial.print("uncorrected Power: ");
Serial.print(power);
Serial.print("W ");
Serial.print("PR PL: ");
Serial.print(PR);
Serial.print("W ");
Serial.print(PL);
Serial.print("W ");
Serial.print("FR FL: ");
Serial.print(FR);
Serial.print("N ");
Serial.print(FL);
Serial.print("N ");
Serial.print("Temp: ");
Serial.print(t);
Serial.print("degree C ");
Serial.print("Omega: ");
Serial.print(w);
Serial.print("rad/s ");
Serial.print("Alpha: ");
Serial.print(alpha);
Serial.print("degree ");
Serial.print("revolutions: ");
Serial.print(revolutions);
Serial.print(" ");
Serial.print("timestamp: ");
Serial.print(timestamp);
Serial.print(" ");
Serial.print("lastMovementTime: ");
Serial.print(lastMovementTime);
Serial.print("ms ");
Serial.print("dt: ");
Serial.print(dt);
Serial.println("ms");

/*
```

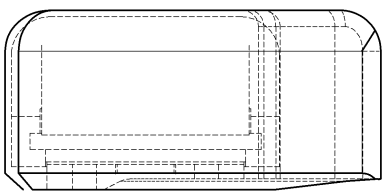
```
//WebSerial.print("Sent Power: ");
//WebSerial.println(power_corrected);
//WebSerial.print("W ");
//WebSerial.print("PR PL: ");
//WebSerial.print(PR);
//WebSerial.print("W ");
//WebSerial.print(PL);
//WebSerial.print("W ");
//WebSerial.print("FR FL: ");
WebSerial.print(FR);
WebSerial.print("N ");
WebSerial.print(FL);
WebSerial.print("N ");
//WebSerial.print("Omega: ");
WebSerial.print(w);
WebSerial.println("rad/s ");
//WebSerial.print("Alpha: ");
//WebSerial.print(alpha);
//WebSerial.print("degree ");
//WebSerial.print("dt: ");
//WebSerial.print(dt);
//WebSerial.println("ms");

delay(500);
*/

// Prepare the data packet as a std::string
std::string value;

// Add the flags
value += static_cast<char>(flags & 0xFF); // Flags LSB
value += static_cast<char>((flags >> 8) & 0xFF); // Flags MSB

// Add the power
value += static_cast<char>(power & 0xFF); // Power LSB
value += static_cast<char>((power >> 8) & 0xFF); // Power MSB

// Add pedal balance if present
if (flags & (1 << 0)) {
    value += static_cast<char>(balance);
}

// Add crank revolution data if present
if (flags & (1 << 5)) {
    value += static_cast<char>(revolutions & 0xFF); // Revolutions LSB
    value += static_cast<char>((revolutions >> 8) & 0xFF); //
        Revolutions MSB
}

// Add last crank event time if present
```

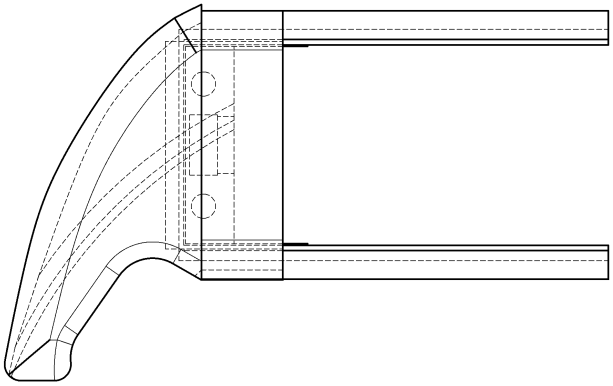
```
    if (flags & (1 << 5)) {
        value += static_cast<char>(timestamp & 0xFF); // Last Crank Event
            Time LSB
        value += static_cast<char>((timestamp >> 8) & 0xFF); // Last Crank
            Event Time MSB
    }

    // Set the characteristic value and notify
    pCyclingPowerMeasurementCharacteristic->setValue((uint8_t*)value.data(),
        value.length());
    pCyclingPowerMeasurementCharacteristic->notify();
  }
}
```

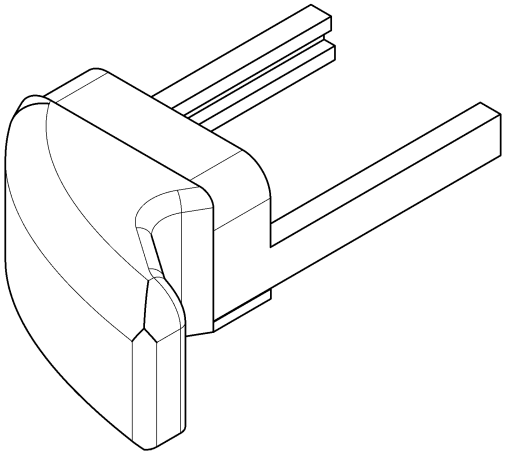
6.2 Technical Drawings











UNLESS OTHERWISE SPECIFIED, DIMENSIONS ARE IN MILLIMETERS		NAME	SIGNATURE	DATE
ANGULAR = ± °	DRAWN	OLIVER PARGFRIEDER		2024-04-10
SURFACE FINISH	CHECKED			
	APPROVED			
DO NOT SCALE DRAWING				
BREAK ALL SHARP EDGES AND REMOVE BURRS				
FIRST ANGLE PROJECTION	MATERIAL		FINISH	

TITLE: Power Meter Slider/Lid for Shimano Deore XT FC-M8000-B2

SIZE	DWG NO.	REV
A3	2	1

SCALE	WEIGHT	SHEET
2:1		1 of 1