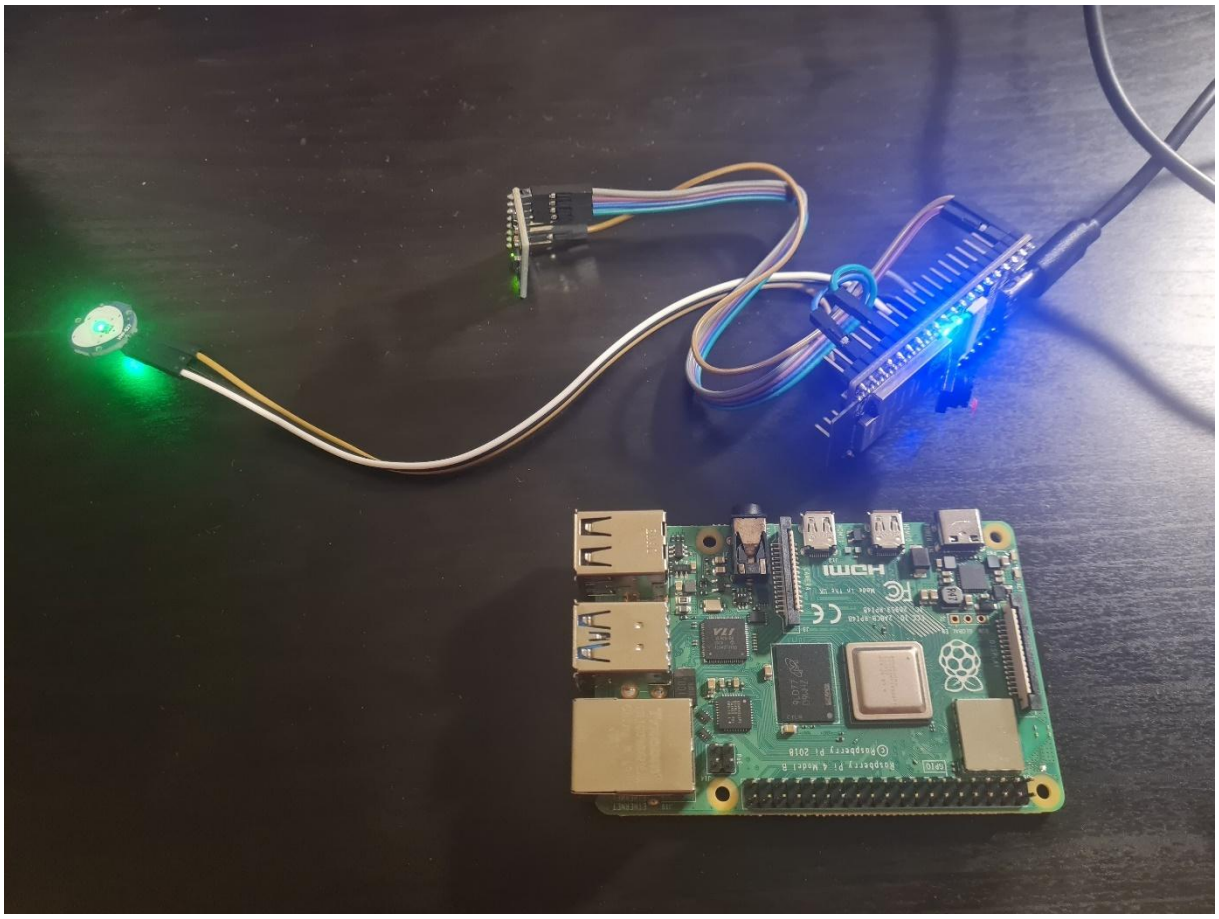


Emerging Technologies 2025

Wearable-Projekt – Ein Handschuh zur Überwachung des Schlafs

Badi Al-Bahra

Index

Abstract **3**

Idee **3**

Aufbau **3**

Kommunikation zwischen ESP32 und Raspberry Pi: MQTT **5**

Was ist MQTT? *5*

Empfangen und Senden von Daten am ESP32 ..... **5**

Empfang von Daten am Raspberry Pi **7**

Datenverarbeitung **8**

Pulssensor *8*

Peakerkennung 8

Fourier-Transformation 9

Gyroskop *10*

Ergebnisse **11**

Fazit und Ideen für die Zukunft **13**

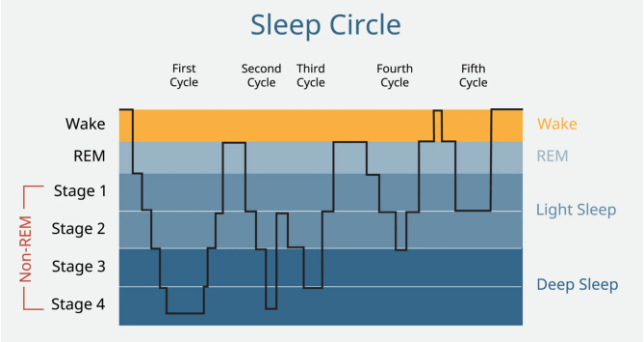
Quellen und Hilfreiche Links **14**

Abstract

Diese Dokumentation handelt von meinem Projekt über ein tragbares System, das den Schlaf überwacht: Ich habe den Prototypen eines Handschuhs entwickelt, der mithilfe eines Pulssensors und eines Bewegungssensors bzw. Gyroskops Werte während des Schlafens sammelt. Diese Werte sollen dazu genutzt werden, den Schlaf zu analysieren: So kann z.B. durch einen geringeren Puls tendenziell auf eine Non-REM-Schlafphase geschlossen werden bzw. umgekehrt auf eine REM-Schlafphase. Dennoch gilt zu beachten, dass die Hardware, die in diesem Projekt verwendet wurde, natürlich eher für den Privatgebrauch und zum Experimentieren gedacht ist und nicht für aussagekräftige wissenschaftliche Ergebnisse.

Idee

Ich persönlich habe oft Probleme mit Weckerstellen und dem Aufstehen zur richtigen Zeit. Manchmal schlafe ich 6 Stunden und habe erholsameren Schlaf als andere Male, wo ich 8 Stunden geschlafen habe. Prinzipiell kann man sagen, dass es einfacher ist, in einer leichten Schlafphase aufzuwachen als in einer tiefen. Leider hat der Wecker kein Gespür für sowas. Deswegen habe ich mir gedacht: Wäre es nicht nützlich, ein Gerät zu haben, was dabei hilft, im richtigen Moment aufzuwachen? So kam die Idee für den Prototypen. Dieser soll meinen Schlaf überwachen, Werte sammeln, wodurch ich dann vielleicht besser meinen Schlaf verstehen kann und mir dann auch hoffentlich besser meine Wecker stellen kann.



Schlafphasen: https://www.simplypsychology.org/sleep-stages.html

Aufbau

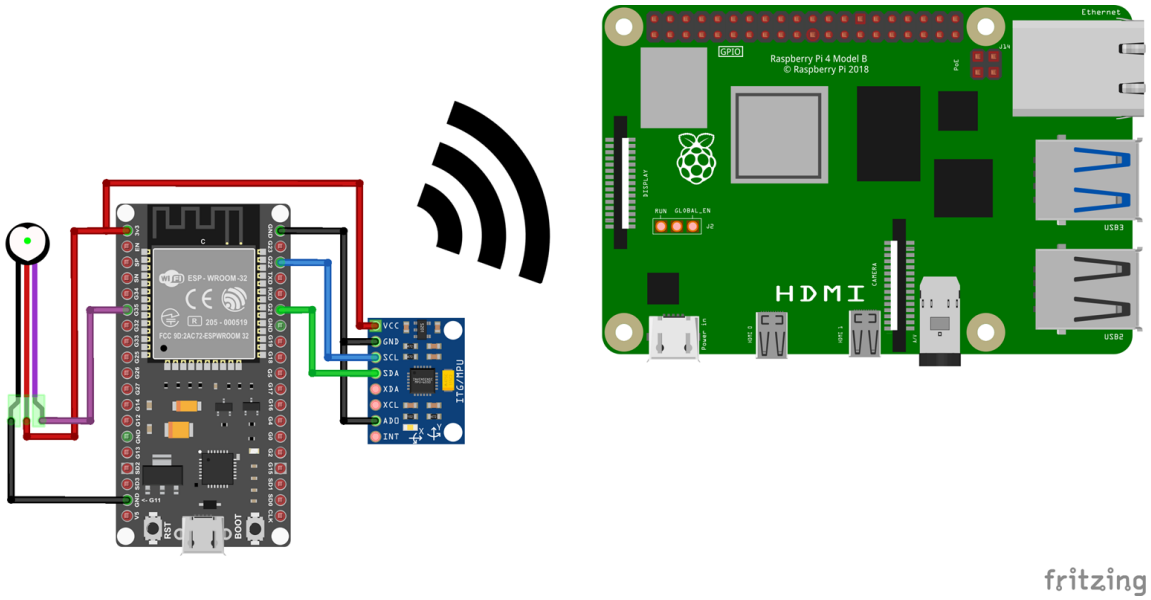
Den Aufbau des Projekts wollte ich erstmal möglichst einfach halten, um mir die Programmierung nicht unnötig zu verkomplizieren. Die Auswahl der möglichen geeigneten Komponenten war auch nicht besonders groß, daher kam ich auf die einfache Kombination eines Pulssensors mit einem Accelerometer bzw. eines Gyroskops.

Der HW-827-Pulssensor soll dabei die Herzfrequenz überwachen. Des Weiteren verwende ich das Gyroskop des MPU6050, was die Bewegung beobachten soll. Ich habe mich gegen die Verwendung des Accelerometers entschieden, da ich die Werte des Gyroskops als einfacher auszuwerten empfunden habe. Die Qualität der Werte beider Sensoren unterscheidet sich dabei allerdings nicht.

Die beiden Sensoren werden an den ESP32 angeschlossen: Der Pulssensor an die Pins 3.3V, GND und einen analogen Pin, über den die Werte des Sensors übertragen werden, in meinem Fall ist das Pin 34. Der MPU6050 kommuniziert über das I2C Protokoll, hat also dementsprechend einen SCL-Pin, der über den GPIO-Pin 22 angeschlossen ist, und einen SDA-Pin, der mit dem Pin 21 verbunden ist. Diese beiden Pins sind die Standard-I2C-Pins am ESP32. Natürlich ist auch dieser an 3.3V und GND angeschlossen. Außerdem habe ich den AD0-Pin ebenfalls an GND angeschlossen. Zuvor hatte ich das nicht getan, was des Öfteren zu fehlerhaften Werten geführt hatte. Ich habe nachgelesen, dass es helfen soll, AD0 an GND anzuschließen. Und tatsächlich wurde das Problem dadurch behoben.

Nach einiger Zeit rumexperimentieren mit dem Pulssensor habe ich gemerkt, dass die Werte nicht besonders zuverlässig sind. Sie fluktuieren andauernd und es gibt ein sehr starkes Rauschen. Wie ich genau an dieses Problem herangegangen bin, erkläre ich später genauer. Allerdings wollte ich es mir einfacher machen, und die Daten daher nicht direkt auf dem ESP32 auswerten. Da ich noch einen Raspberry Pi 4B rumliegen hatte, habe ich mich entschieden, diesen auch zu verwenden.

Der Raspberry Pi ist nicht nur deutlich leistungsfähiger, sondern hat auch die Möglichkeit alle Daten auf einer Oberfläche anzuzeigen. Deswegen habe ich anschließend an folgende Aufgabenverteilung gedacht: Der ESP32 inklusive der zwei Sensoren sammelt alle nötigen Daten und sendet diese über das lokale Netzwerk an den Raspberry Pi, der diese dann verarbeitet und anschließend grafisch darstellt.



Aufbau des Projekts (gemacht mit *Fritzing*)

Kommunikation zwischen ESP32 und Raspberry Pi: MQTT

Was ist MQTT?

MQTT ist ein offenes Netzwerkprotokoll, was vor allem bei IoT Anwendung findet. Es ist einfach zu implementieren, sowohl auf Raspi-Seite als auch auf ESP32-Seite. Ganz grob funktioniert es konkret bei meinem Projekt folgendermaßen: Auf dem Raspberry Pi läuft ein MQTT-Server. Der ESP32 kann als Client an diesen Server Nachrichten senden. In diesen Nachrichten befinden sich die Sensorwerte. Auf dem Raspberry Pi ist außerdem ein weiterer Client, der diese Daten vom Server anfragt und dann weiterverarbeiten kann.

Empfangen und Senden von Daten am ESP32

Folgendermaßen sehen meine `setup()` auf dem ESP32 aus:

```
void setup() {
  Serial.begin(115200);
  pinMode(ledPin, OUTPUT);

  while (!Serial)
    delay(10); // will pause Zero, Leonardo, etc until serial console opens

  Serial.println("Adafruit MPU6050 test!");

  // setup mpu6050 sensor
  if (!mpu.begin()) {
    Serial.println("Failed to find MPU6050 chip");
    while (1) {
      delay(10);
    }
  }
  Serial.println("MPU6050 Found!");

  mpu.setAccelerometerRange(MPU6050_RANGE_8_G);
  mpu.setGyroRange(MPU6050_RANGE_500_DEG);
  mpu.setFilterBandwidth(MPU6050_BAND_21_HZ);

  // setup WiFi Connection to MQTT Broker
  setup_wifi();
  // 1883 default MQTT-port, mqtt_server contains server IP
  client.setServer(mqtt_server,1883);
  client.setCallback(callback);

  delay(100);
}
```

```
void loop() {
  //Check Connection to MQTT-Server
  if (!client.connected()) {
    connect_mqttServer();
  }
  client.loop();

  //Pulse Sensor
  handlePulseSensor();

  //Motion Sensor
  handleMotionSensor();
}
```

In der `loop()` werden die Funktionen `handlePulseSensor()` und `handleMotionSensor()` in einer Endlosschleife ausgeführt. In diesen Funktionen werden die Daten vom Sensor aufgenommen und an den Server gesendet. Die `handlePulseSensor()` sieht folgendermaßen aus:

```
void handlePulseSensor() {
  unsigned long currentTime = millis();
  // small delay between data points
  if (currentTime - lastTimeMeasured_pulse > 0) {
    lastTimeMeasured_pulse = currentTime;
    count_p--;
    sig = analogRead(psPIN); // Read the PulseSensor's value.
    average_pulse += sig;

    if (count_p == 0) {
      // calculate the average of the last 20 values
      int pulseData = (int)average_pulse / iterations_pulse;

      // Format: timestamp,value
      char message[50];
      sprintf(message, "%lu,%d", currentTime, pulseData);
      // send the data to the server on the topic "esp32/pulse"
      client.publish("esp32/pulse", message);

      Serial.println(message); // Debugging output

      // reset the values
      average_pulse = 0;
      count_p = iterations_pulse;
    }
  }
}
```

Die `handleMotionSensor()` sieht funktioniert auf dieselbe Art und Weise. Durch das Sammeln von 20 Datenpunkten und anschließendes Bilden des Mittelwerts kann schon einiges an Rauschen entfernt werden.

Die Daten werden dann vom Raspberry-Pi empfangen. Die primäre Aufgabe des RPi-Clients wird es sein, die Daten zu analysieren. Dementsprechend habe ich mich dafür entschieden, Python zu verwenden, da es viele nützliche Libraries bietet, die einem die Arbeit erheblich erleichtern.

Empfang von Daten am Raspberry Pi

Um die Daten zu empfangen, gibt es auf dem Raspberry-Pi, wie vorher schon erwähnt, einen weiteren Client, der der Verarbeitung dient.

```
client = mqtt.Client("rpi_client1", protocol=mqtt.MQTTv311)
client.on_connect = on_connect
client.on_disconnect = on_disconnect
client.message_callback_add('esp32/pulse', callback_esp32_sensor1)
client.message_callback_add('esp32/movement', callback_esp32_sensor2)
client.connect('127.0.0.1', 1883)
client.loop_start()

try:
    while True:
        time.sleep(4)
        if flag_connected != 1:
            print("Trying to reconnect to MQTT server...")
except KeyboardInterrupt:
    print("Shutting down...")
    client.loop_stop()
    client.disconnect()
    print("Stopped.")
```

In diesem Teil des Codes wird der Client konfiguriert. Die callback-Funktionen werden immer dann aufgerufen, wenn der Client auf dem jeweiligen Topic (`'esp32/pulse'` und `'esp32/movement'`) eine Nachricht erhält. `client.loop_start()` sorgt dafür, dass der Client startet und in einem loop verharrt. In den callback-Funktionen werden die empfangenen Daten geparsed und in eine Data-Queue gelegt:

```
def callback_esp32_sensor1(client, userdata, msg):
    try:
        timestamp, value = map(int, msg.payload.decode('utf-8').split(','))
        pulse_data_queue.put({
            "timestamp": np.array([timestamp]),
            "value": np.array([value], dtype=np.float64)
            })
    except ValueError as e:
        print(f"Error parsing pulse data: {e}")
```

Sobald sich die Daten in der Queue befinden, können sie bearbeitet werden.

Datenverarbeitung

Pulssensor

Mein erster Ansatz, den Puls zu erkennen war es, die Peaks der Impulse zu erkennen und dann die Differenz zum nächsten Peak zu berechnen und durch die Differenz der Zeit zu teilen.

Peakerkennung

Dies ist in Python besonders einfach, wenn man die *scipy*-Funktion verwendet, da diese eine vorgefertigte Funktion für die Erkennung von Peaks mitbringt. Dafür habe ich die `analyze_pulse_data()` geschrieben:

Wenn genügend Werte zur Berechnung eines Impulses vorhanden sind, berechne die minimale Höhe des Peaks anhand des Mittelwerts und der Standardabweichung.

```
if pulse_values.size >= window_size:
    min_peak_height = np.mean(pulse_values) + 0.5 * np.std(pulse_values)
    detected_peaks, _ = find_peaks(
        pulse_values,
        height=min_peak_height,
        distance=0.25 * time_window
    )
```

Wenn mindestens zwei Peaks erkannt wurden, berechne die durchschnittlichen Zeitintervalle und daraus die Herzfrequenz. Im Allgemeinen kann davon ausgegangen werden, dass die menschliche Herzfrequenz immer zwischen etwa 40 und 200 BPM liegt.

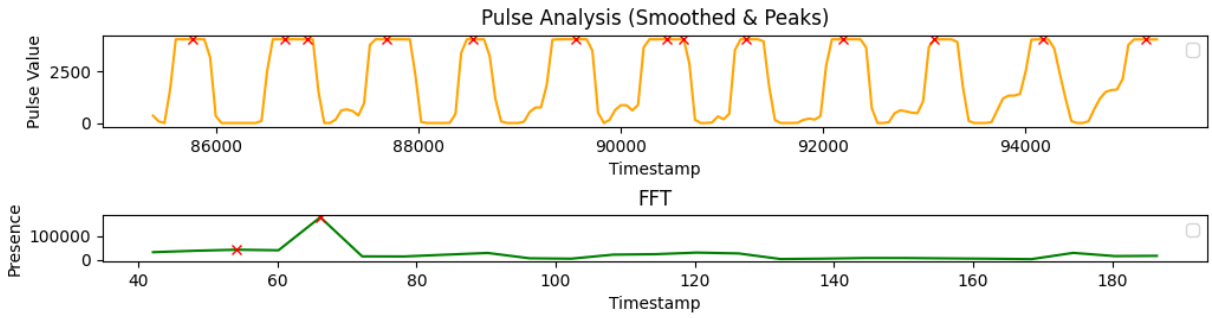
```
    if detected_peaks.size > 1:
        peak_time_stamps = rec_p_time[detected_peaks]
        average_peak_interval = np.mean(np.diff(peak_time_stamps))
        heartbeat = 60000 / average_peak_interval

        if 40 < heartbeat < 200:
               print(f"hearbeat (peak detection): {heartbeat:.1f} bpm")
        else:
               print("hearbeat (peak detection): ---- bpm")
    else:
           print("hearbeat (peak detection): no peaks")
```

Diese Variante funktioniert gut, solange die Werte stabil bleiben, wenig fluktuieren und wenig Störsignale vorkommen. Wie ich aber zuvor schon erwähnt hatte, ist das bei diesem Sensor nicht der Fall. Manchmal hat man Glück, aber meistens eben nicht, daher muss eine andere Methode her.

Fourier-Transformation

Eine andere hilfreiche Methode, auf die mich ein Freund hingewiesen hat, ist die Fourier-Transformation. Die Fourier-Transformation dient dazu, mathematische Funktionen oder Signale in ihre Frequenzbereiche zu unterteilen. In meinem Fall kann ich sie dafür nutzen, das Pulssignal von den Störsignalen zu unterscheiden. *Numpy* bringt eine Funktion für die schnelle Fourier-Transformation (FFT) mit, ein Algorithmus für die schnelle Berechnung. Daher ist die Implementation hier auch recht einfach.

```
fft_pulse_values = abs(np.fft.fft(pulse_values))
```



Peakerkennung- und FFT-Diagramme

Das obere Diagramm zeigt die Peakerkennung. Wie man sieht, funktioniert diese mehr oder weniger in Ordnung. Hier hätte man noch den Mindestabstand der Punkte verbessern können. Das Ergebnis der FFT kann man in dem unteren Diagramm sehen. Man erkennt, dass in etwa 65 BPM durch die FFT erkannt wird.

Um genauere Werte für die Herzfrequenz zu errechnen, kann man nun einen Curvefit durchführen. Hier habe ich mir von Chat GPT helfen lassen, den Code zu schreiben. Dafür habe ich eine neue Funktion `gaussian_fit_frequency()`. Zunächst suchen wir wieder alle Peaks, die auf dem Frequenzspektrum vorkommen, also die am stärksten vertretenen Frequenzen (im Diagramm sind diese in BPM gegeben).

```
min_peak_height = (
    np.mean(fft_pulse_values[fft_start:fft_end])
    + 0.4 * np.std(fft_pulse_values[fft_start:fft_end])
)
detected_fft_peaks, _ = find_peaks(
    fft_pulse_values[fft_start:fft_end],
    height=min_peak_height
)
detected_fft_peaks += fft_start
```

`fft_start` und `fft_end` begrenzen dabei das Spektrum auf sinnvolle Werte, in meinem Fall sind das die Frequenzen zwischen ca. 40 und 200 BPM. Danach prüfen wir, ob Peaks gefunden wurden, wenn ja, dann soll der Peak mit dem höchsten Wert, also die am stärksten vertretene Frequenz, ausgesucht werden:

```
if detected_fft_peaks.size > 0:
    peak_index = detected_fft_peaks[max(
        range(len(fft_pulse_values[detected_fft_peaks])),
        key = fft_pulse_values[detected_fft_peaks].__getitem__
    )]
```

Nun legen wir die Grenzen des Curvefit fest mit `window_size`:

```
    window_size = 3
    start = max(fft_start, peak_index - window_size)
    end = min(len(fft_pulse_values[fft_start:fft_end]),
        peak_index + window_size)
    x_data = fft_x_axis[start:end]
    y_data = fft_pulse_values[start:end]
```

Konkret bedeutet das, dass wir nun die 6 zum Peak umliegenden Werte (3 in jede Richtung) in unsere Rechnung mit einbeziehen, sodass nicht nur die Frequenz des Peaks, sondern auch die umliegenden berücksichtigt werden. Als nächstes können wir die von *scipy* vorgefertigte Funktion `curve_fit()` nutzen, um die finale Rechnung durchzuführen.

```
    try:
        popt, _ = curve_fit(gaussian, x_data, y_data,
            p0=[np.max(y_data), fft_x_axis[peak_index], 5.0])
        _, heartbeat, _ = popt

        print(f"hearbeat:  {heartbeat:.1f} bpm", end="\t", flush=True)
    except:
        print(f"hearbeat:  ---- bpm", end="\t", flush=True)
```

Der try-except-Block fängt Fehler ab. Diese könnten auftreten, wenn die Daten zu schlecht sind oder das Curvefit scheitert.

Das Programm betrachtet immer die Werte der letzten 10 Sekunden, berechnet für diese dann einen Herzschlag mithilfe der FFT und des Curvefits und speichert diesen Wert dann in einem Array `heartbeat_values`, was den Verlauf des gesamten Schlafs darstellen soll.

Das waren die wichtigsten Teile für den Pulssensor. Das Gyroskop ist dahingegen deutlich einfacher zu erklären.

Gyroskop

Das Gyroskop hat mich im Vergleich zum Pulssensor deutlich weniger Zeit gekostet. Die Werte des Gyroskops haben deutlich weniger Rauschen. Eine Bewegung kann eindeutig erkannt werden, genauso wie ihre Intensität.

Das Gyroskop hat normalerweise drei Werte: x, y und z. Der Vorteil von Gyroskop-Werten zu den Accelerometer-Werten ist, dass die Werte bei Stillstand nahe 0 bleiben, was bei dem Accelerometer anders ist.

Da ich nur grob messen möchte, wieviel sich eine Person im Schlaf bewegt, brauche ich nur einen Wert, der die Intensität der Bewegung über die Zeit darstellt. Deshalb habe ich alle Werte zu einem einzigen aufaddiert und in dem Array `gyro_sum` gespeichert. Für das Gyroskop möchte ich alle 30 Sekunden einen Wert speichern, der die Summe aller Bewegungen in diesem Zeitraum darstellt.

```
g_val_count += 1
recent_g_integral = 0
if g_val_count == int(60/time_window * window_size / 2):
    for i in range(window_size):
        recent_g_integral += gyro_sum[i] if gyro_sum[i] > 0 else 0
    g_integrals = np.append(g_integrals, recent_g_integral)
    gyro_time = np.append(gyro_time, new_timestamp/60000)
    g_val_count = 0
```

In `g_integrals` werden diese Summen der 30-Sekunden-Fenster gespeichert und dann über den gesamten Zeitraum dargestellt. Im Unterschied zum Pulssensor, wird hier jede 30 Sekunden aufsummiert, wohingegen beim Pulssensor immer die letzten 10 Sekunden betrachtet werden. Beim Puls habe ich also ein sich kontinuierlich bewegendes Fenster, während beim Gyroskop jede 30 Sekunden separat betrachtet werden.

Ergebnisse

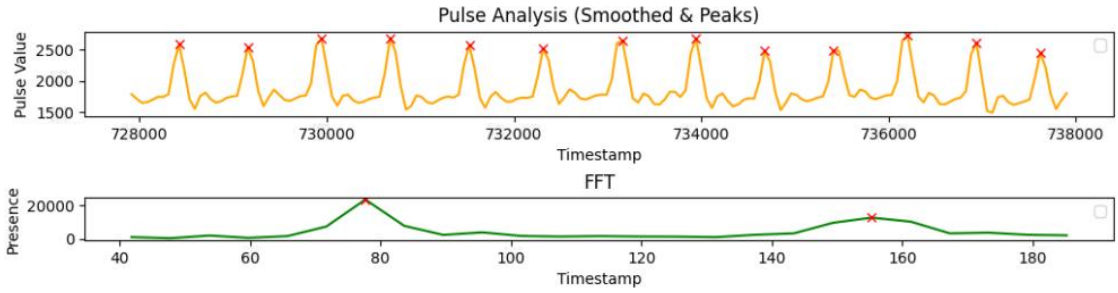
Das Gyroskop liefert eindeutige Werte, mit denen man sehr gut arbeiten kann. Die Verarbeitung dieser ist recht simpel und erfordert nicht viel Aufwand.
Dementsprechend kann die Bewegung ziemlich gut überwacht werden.

Das Problem ist eher der Pulssensor. Dieser liefert zufällige Werte, wenn kein Finger aufliegt. Sobald man seinen Finger auflegt, liefert er manchmal gute, manchmal ziemlich verrauschte Werte, abhängig davon, wie gut der Finger platziert ist, und manchmal auch komplett zufällige Werte. Das stellt ein großes Problem dar, da es im Schlaf natürlich nicht möglich ist, dies zu prüfen.

Insgesamt habe ich 5 Diagramme, die alles darstellen:

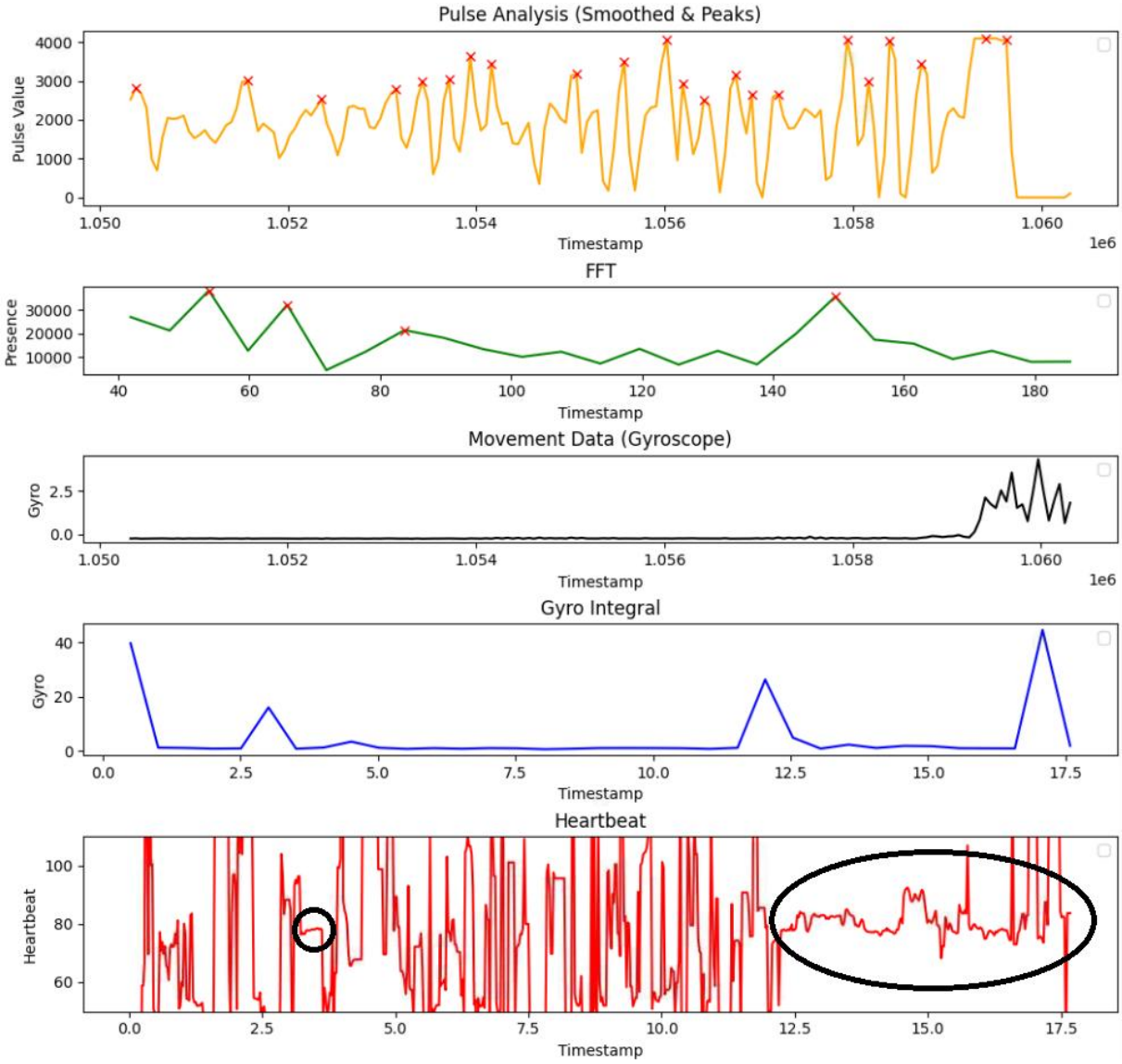
1. Das erste Diagramm zeigt die Rohdaten der letzten 10 Sekunden des Pulssensor und die durch die *scipy*-Peakerkennung erkannten Peaks.
2. Das zweite zeigt die Ergebnisse der FFT.
3. Das dritte Diagramm zeigt die Summe der Rohdaten der letzten 10 Sekunden des Gyroskops.
4. Das vierte Diagramm zeigt den Verlauf der Bewegungen, also das zuvor erwähnte Array `g_integrals`.

5. Das letzte Diagramm zeigt der Verlauf des Herzschlags, also `heartbeat_values`.



Idealwerte für die Peakerkennung und FFT

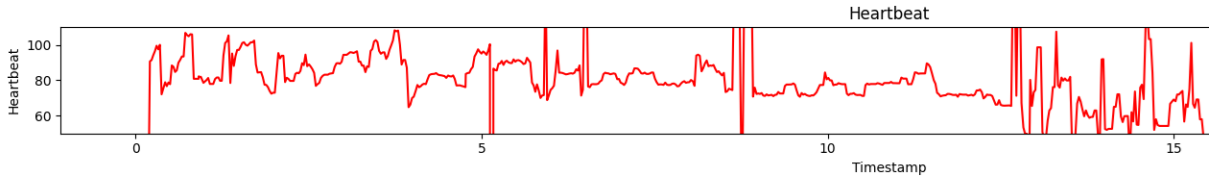
So sehen ideale Werte des Pulssensors aus. Sowohl die Peakerkennung als auch die FFT liefern eine ideale Auswertung. Leider sind die Werte selten so gut.



Alle Diagramme

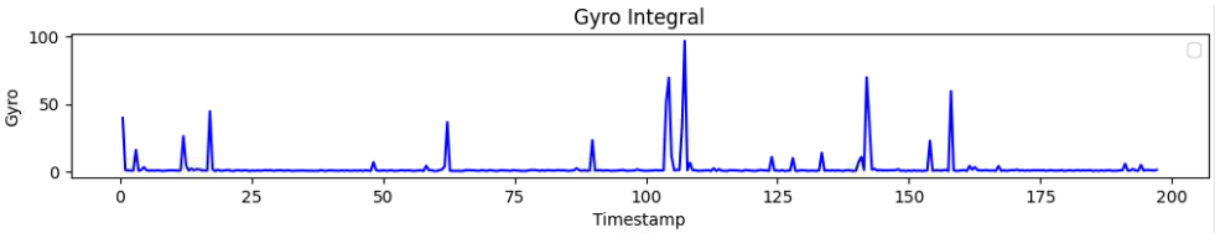
Hier sieht man einmal alle Diagramme. Zu diesem Zeitpunkt hatte ich meinen Finger aufliegen. Dennoch sind die Werte unbrauchbar. In den schwarz umkreisten Bereichen

kann man allerdings erkennen, dass für einige Minuten ein brauchbares Signal vorhanden war. Dennoch sind diese nicht frei von Ausreißern. Ab Minute 17 etwa, werden die Werte dann allerdings wieder unbrauchbar.



Guter Verlauf der Herzfrequenz

Hier ein noch ein halbwegs brauchbarer Verlauf. Von Minute 0 bis ca. Minute 13 hatte ich meinen Finger aufliegen. Es gibt einige Ausreißer, dennoch größtenteils sinnvolle Werte. Ab Minute 13 dann habe ich meinen Finger dann abgenommen.



Guter Verlauf des Gyroskops

Hier nochmal ein Verlauf des Gyroskops über 200 Minuten. Die Bewegungen und deren Intensitäten kann man sehr eindeutig an den Peaks erkennen.

Zum Ende des Projektes hin, habe ich probieren wollen, den Handschuh im Schlaf zu tragen, allerdings, ist der Pulssensor mittlerweile aus mir unerklärlichen Gründen nicht mehr funktionsfähig. Das heißt, dass egal ob ich meinen Finger oder z.B. mein Ohrläppchen auflege, die Werte bleiben größtenteils zufällig. An diesem Punkt bringt die FFT natürlich auch nichts mehr. Ganz selten kommen noch brauchbare Werte zustande, das ist aber eine Rarität.

Fazit und Ideen für die Zukunft

Insgesamt kann ich also sagen, dass der Prototyp auch nur ein Prototyp bleibt. Der Pulssensor ist einfach zu ungenau, um sinnvolle Ergebnisse zu liefern. Die Bewegung kann gut gemessen werden. Wirklich sinnvoll verwenden, kann ich dieses Modell allerdings noch nicht.

Ich bin definitiv motiviert, dieses Projekt in Zukunft weiterzuführen. Mir selbst würden brauchbare Ergebnisse auch möglicherweise dabei helfen, meinen Schlaf besser zu verstehen. Dafür braucht es aber sicher einen verlässlicheren Pulssensor. Außerdem finde ich dieses spezifische ESP32-Modell etwas groß, ein kleineres Modell wäre sicher von Vorteil. Außerdem könnte man weitere Sensoren ergänzen, um bessere Aussagen über die Schlafphasen treffen zu können.

Wenn dann final sinnvolle Werte vorliegen, könnte ich auch z.B. sowas wie einen Smart-Alarm implementieren: Man gibt eine Uhrzeit vor, zu der man spätestens aufstehen möchte. Das Programm analysiert, wann man sich in einer leichten Schlafphase befindet, und weckt einen dann zum richtigen Zeitpunkt auf. Zur Erkennung der Richtigen Schlafphasen, könnte es sinnvoll sein, die Ergebnisse mit Machine Learning oder KI zu analysieren.

Quellen und Hilfreiche Links

- https://es.wikipedia.org/wiki/Transformada_de_Fourier
- https://lastminuteengineers.com/pulse-sensor-arduino-tutorial
- https://es.wikipedia.org/wiki/Ajuste_de_curvas
- https://en.wikipedia.org/wiki/MQTT
- ChatGPT