

Tecnologías Emergentes 2024/25

Un sistema vestible – Un Guante Para Monitorizar El Sueño

Badi Al-Bahra

Índice

Abstract	3
Idea	3
Estructura	3
Comunicación entre el ESP32 y el Raspberry Pi: MQTT	4
<i>¿Qué es MQTT?.....</i>	<i>4</i>
Recogida y envío de datos en el ESP32	5
Recepción de datos en el Raspberry Pi	6
Procesamiento de datos	7
<i>Sensor de pulso</i>	<i>7</i>
Detección de picos	8
Transformada de Fourier	8
<i>Giroscopio</i>	<i>10</i>
Resultados.....	11
Conclusión e ideas para el futuro	13
Bibliografía	14

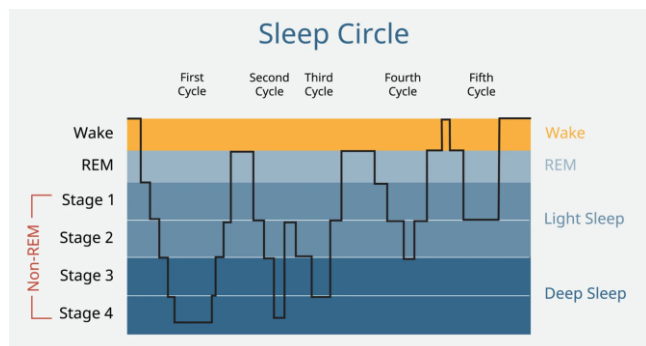
Abstract

Esta documentación trata de mi proyecto sobre un sistema vestido que monitoriza el sueño: He desarrollado un prototipo de un guante que utiliza un sensor de pulso y un sensor de movimiento o giroscopio para recoger valores durante el sueño. Estos valores se utilizarán para analizar el sueño: Por ejemplo, un pulso más bajo suele indicar una fase de sueño no REM o, a la inversa, una fase de sueño REM. No obstante, hay que señalar que el hardware utilizado en este proyecto está destinado, por supuesto, al uso privado y a la experimentación, más que a obtener resultados científicos significativos.

Idea

Personalmente, a menudo tengo problemas para poner el despertador y levantarme a la hora correcta. A veces duermo 6 horas y tengo un sueño más reparador que otras veces, cuando he dormido 8 horas. En principio, se puede decir que es más fácil despertarse en una fase de sueño ligero que en una profunda. Por desgracia, el despertador no tiene sentido para esto.

Así que pensé: ¿no sería útil tener un dispositivo que te ayudara a despertarte en el momento adecuado? Así surgió la idea del prototipo. Se supone que tiene que vigilar mi sueño, recoger datos, para que quizá pueda entender mejor mi sueño y, con un poco de suerte, programar mejor mi despertador.



Fases del sueño: <https://www.simplypsychology.org/sleep-stages.html>

Estructura

Quise mantener la estructura del proyecto lo más simple posible al principio, para no complicarme innecesariamente con la programación. La selección de los componentes adecuados disponibles no era particularmente grande, por lo que opté por la combinación sencilla de un sensor de pulso con un acelerómetro o un giroscopio.

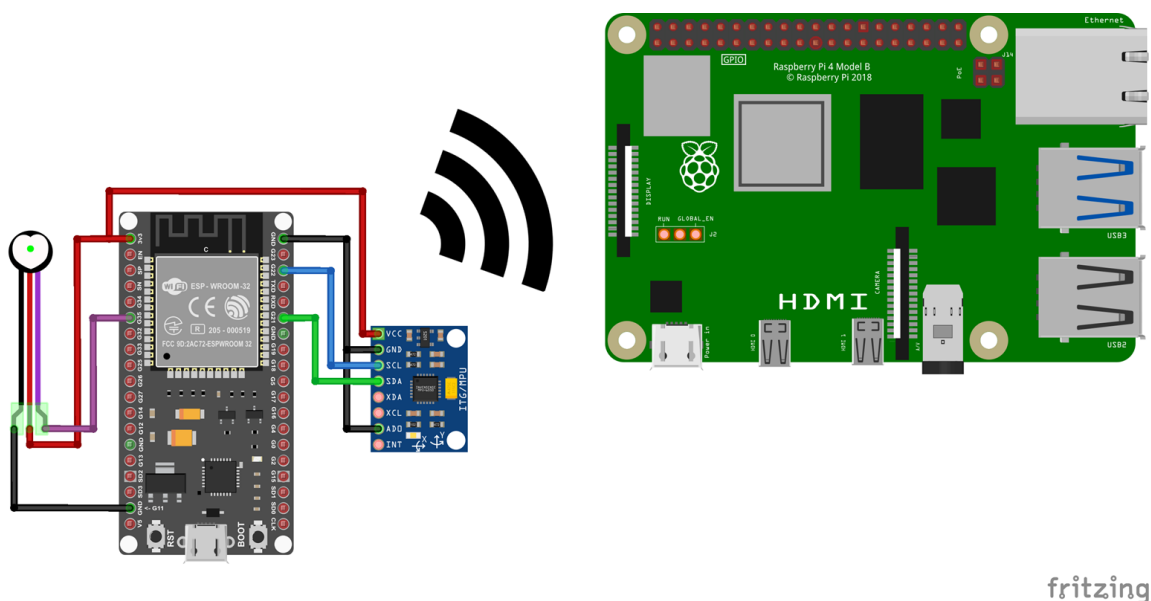
El sensor de pulso HW-827 está destinado a monitorear la frecuencia cardíaca. Además, uso el giroscopio del MPU6050 para observar el movimiento. Decidí no usar el acelerómetro porque encontré más fácil interpretar los valores del giroscopio. Sin embargo, la calidad de los valores de ambos sensores no varía significativamente.

Ambos sensores están conectados al ESP32: el sensor de pulso se conecta a los pines 3.3V, GND y a un pin analógico, por el cual se transmiten los valores del sensor; en mi caso, el pin 34. El MPU6050 se comunica mediante el protocolo I2C, por lo que

tiene un pin SCL, que está conectado al pin GPIO 22, y un pin SDA, que está conectado al pin 21. Estos dos pines son los pines estándar de I2C en el ESP32. Por supuesto, también está conectado a 3.3V y GND. Además, conecté el pin AD0 también a GND. Anteriormente no lo había hecho, lo que a menudo resultaba en valores incorrectos. Leí que conectar AD0 a GND debería ayudar y, de hecho, esto resolvió el problema.

Después de experimentar un tiempo con el sensor de pulso, me di cuenta de que los valores no eran particularmente confiables. Fluctuaban constantemente y había mucho ruido. Explicaré más adelante cómo abordé exactamente este problema. Sin embargo, para simplificar las cosas, decidí no analizar los datos directamente en el ESP32. Como tenía un Raspberry Pi 4B disponible, decidí usarlo también.

El Raspberry Pi no solo es significativamente más potente, sino que también permite mostrar todos los datos en una interfaz visual. Por esta razón, pensé en la siguiente distribución de tareas: el ESP32, junto con los dos sensores, recopila todos los datos necesarios y los envía a través de la red local al Raspberry Pi, que luego los procesa y los presenta gráficamente.



Estructura del proyecto: hecho con *Fritzing*

Comunicación entre el ESP32 y el Raspberry Pi: MQTT

¿Qué es MQTT?

MQTT es un protocolo de red abierto que se utiliza principalmente para IoT. Es fácil de implementar, tanto en el lado del RPi como en el lado ESP32. A grandes rasgos, funciona de la siguiente manera para mi proyecto: Un servidor MQTT se ejecuta en el Raspberry Pi. El ESP32 puede enviar mensajes a este servidor como cliente. Estos mensajes contienen los valores de los sensores. También hay otro cliente en el Raspberry Pi que solicita estos datos desde el servidor y luego los puede procesar.

Recogida y envío de datos en el ESP32

Este es el aspecto de función setup() en el ESP32:

```
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);
}
```

Y este de mi función loop():

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

  //Pulse Sensor
  handlePulseSensor();

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

Las funciones `handlePulseSensor()` y `handleMotionSensor()` se ejecutan en bucle en `loop()`. En estas funciones, los datos del sensor son registrados y enviados al servidor. La función `handlePulseSensor()` tiene el siguiente aspecto:

```
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;
        }
    }
}
```

La función `handleMotionSensor()` funciona de la misma manera. Al recopilar 20 puntos de datos y luego calcular el promedio, se puede eliminar una buena cantidad de ruido. Los datos son recibidos posteriormente por el Raspberry Pi. La tarea principal del cliente en el RPi será analizar los datos. Por lo tanto, decidí usar Python, ya que ofrece muchas bibliotecas útiles que facilitan significativamente el trabajo.

Recepción de datos en el Raspberry Pi

Para recibir los datos, como ya se mencionó anteriormente, hay otro cliente en el Raspberry Pi, que se encarga del procesamiento.

```

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.")

```

En esta parte del código, se configura el cliente. Las funciones de callback se ejecutan siempre que el cliente recibe un mensaje en el respectivo tópico ('esp32/pulse' y 'esp32/movement'). El comando `client.loop_start()` asegura que el cliente se inicie y permanezca en un bucle. En las funciones de callback, los datos recibidos se descodifican y se colocan en una queue:

```

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}")

```

Una vez que los datos están en la cola, pueden ser procesados.

Procesamiento de datos

Sensor de pulso

Mi primera idea para detectar el pulso fue identificar los picos de los impulsos de los valores de los últimos diez segundos, luego calcular la diferencia media de tiempo entre los picos, y finalmente dividir 60 000 por el resultado (en milisegundos).

Detección de picos

Esto es particularmente sencillo en Python si se utiliza la biblioteca *scipy*, ya que incluye una función predefinida para la detección de picos. He escrito la función `analyze_pulse_data()` para lograr eso:

Si hay bastante valores para calcular un pulso, calcula la altura mínima del pico usando el valor medio y la desviación típica.

```
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
    )
```

Si se ha detectado al menos dos picos, calcula los intervalos de tiempo medios y calcula con estos la frecuencia cardíaca. En general, se puede suponer que la frecuencia cardíaca humana se sitúa siempre entre 40 y 200 BPM aproximadamente.

```
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

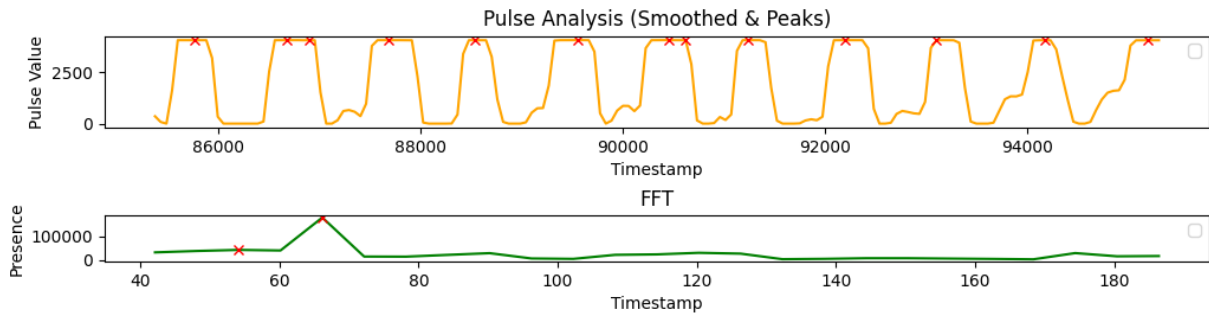
    if 40 < heartbeat < 200:
        print(f"heartbeat (peak detection): {heartbeat:.1f} bpm")
    else:
        print("heartbeat (peak detection): ---- bpm")
else:
    print("heartbeat (peak detection): no peaks")
```

Este método funciona relativamente bien siempre que los valores sean estables, fluctúen poco y haya pocas señales de interferencia. Sin embargo, como mencioné anteriormente, este no es el caso con este sensor. A veces hay suerte, pero la mayoría de las veces no, por lo que se requiere otro método.

Transformada de Fourier

Otro método útil, que un amigo me recomendó cuando le conté de mi proyecto y mi problema actual, es la Transformada de Fourier. Esta técnica se utiliza para descomponer funciones matemáticas o señales en sus componentes de frecuencia. En mi caso, puedo usarla para distinguir la señal del pulso de las señales de interferencia. *Numpy* proporciona una función para realizar la Transformada Rápida de Fourier (FFT), un algoritmo que permite un cálculo rápido. Por lo tanto, la implementación aquí también es bastante sencilla.

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



Diagramas que muestran la detección de picos y la FFT

El diagrama superior muestra la detección de picos. Como se puede ver, funciona más o menos bien. Aquí se podría haber mejorado la distancia mínima entre los puntos. El resultado de la FFT se puede observar en el diagrama inferior. Se puede notar que la FFT detecta aproximadamente 65 BPM.

Para calcular valores más precisos de la frecuencia cardíaca, se puede realizar un ajuste de curva (Curvefit). Para esto, utilicé la ayuda de ChatGPT para escribir el código. Creé una nueva función llamada `gaussian_fit_frequency()`. Primero, buscamos nuevamente todos los picos presentes en el espectro de frecuencias, es decir, las frecuencias más representadas (en el diagrama, estas se expresan en BPM).

```
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
```

Los parámetros `fft_start` y `fft_end` limitan el espectro a valores entre 40 y 200 BPM. Después, verificamos si se han encontrado picos; si es así, se selecciona el pico con el valor más alto, es decir, la frecuencia más representada:

```
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__
    )]
```

A continuación, definimos los límites del ajuste de curva con `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]
```

En concreto, esto significa que tomamos los 6 valores que rodean el pico (3 en cada dirección) para incluirlos en nuestro cálculo, de manera que no solo se considere la frecuencia del pico, sino también las frecuencias circundantes. Luego, utilizamos la función predefinida `curve_fit()` de *scipy* para realizar el cálculo final.

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

El bloque try-except captura errores que podrían ocurrir si los datos son demasiado deficientes o si el ajuste de curva falla.

El programa siempre considera los valores de los últimos 10 segundos, calcula la frecuencia cardíaca utilizando FFT y el ajuste de curva, y almacena este valor en un array `heartbeat_values` que representa el curso completo del sueño.

Estas son las partes más importantes relacionadas con el sensor de pulso. El giroscopio, en cambio, es mucho más sencillo de explicar.

Giroscopio

La programación del giroscopio me tomó significativamente menos tiempo que la del sensor de pulso. Los valores del giroscopio tienen mucho menos ruido. Es posible identificar claramente un movimiento, así como su intensidad.

El giroscopio normalmente proporciona tres valores: x, y y z. La ventaja de los valores del giroscopio frente a los del acelerómetro es que los valores del giroscopio permanecen cerca de 0 cuando no hay movimiento, lo que no ocurre con el acelerómetro.

Dado que solo quiero medir de forma aproximada cuánto se mueve una persona mientras duerme, necesito solo un valor que represente la intensidad del movimiento a lo largo del tiempo. Por ello, sumé todos los valores en un solo resultado, que almacené en el array `gyro_sum`. Para el giroscopio, quiero guardar un valor cada 30 segundos, que represente la suma de todos los movimientos en ese intervalo.

```

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

```

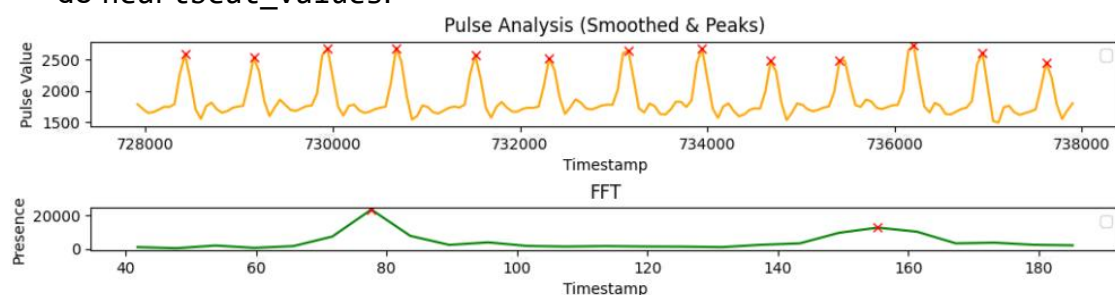
En `g_integrals` se almacenan estas sumas de intervalos de 30 segundos y se presentan para todo el período de tiempo. A diferencia del sensor de pulso, que considera continuamente los últimos 10 segundos, el giroscopio suma los datos cada 30 segundos de forma independiente. Es decir, para el pulso utilizo una ventana móvil continua, mientras que, para el giroscopio, se evalúan intervalos fijos de 30 segundos.

Resultados

El giroscopio proporciona valores claros que son muy útiles y fáciles de procesar, sin requerir mucho esfuerzo. Por lo tanto, el movimiento puede ser monitoreado de manera bastante efectiva. El problema principal es el sensor de pulso. Este genera valores aleatorios cuando no se coloca un dedo sobre él. Incluso cuando se coloca el dedo, a veces proporciona buenos valores, otras veces valores con mucho ruido, dependiendo de qué tan bien esté colocado el dedo. En ocasiones, incluso proporciona valores completamente aleatorios. Esto es un problema importante, ya que no es posible controlar esto mientras se duerme.

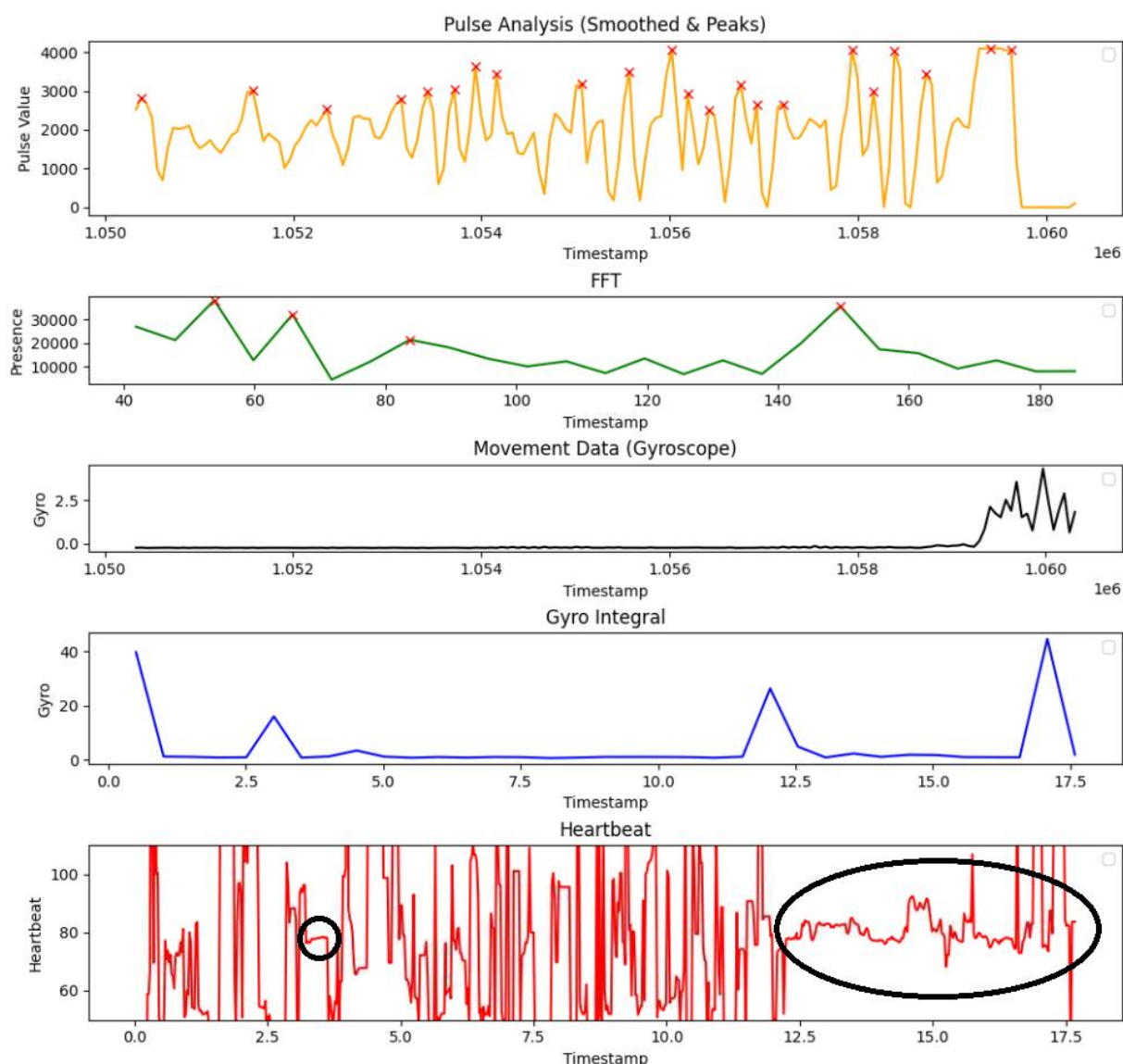
En total, generé 5 diagramas para visualizar los datos:

1. El primer diagrama muestra los datos brutos de los últimos 10 segundos del sensor de pulso y los picos detectados mediante la función de detección de picos de *scipy*.
2. El segundo diagrama muestra los resultados de la FFT.
3. El tercer diagrama presenta la suma de los datos brutos de los últimos 10 segundos del giroscopio.
4. El cuarto diagrama representa el desarrollo de los movimientos, es decir, el array mencionado `g_integrals`.
5. El último diagrama muestra el desarrollo del ritmo cardíaco, es decir, los valores de `heartbeat_values`.



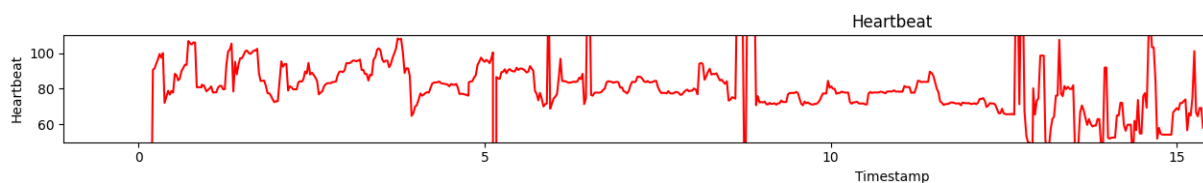
Diagramas que muestran valores ideales de la detección de picos y la FFT

Así se ven los valores ideales del sensor de pulso. Tanto la detección de picos como la FFT proporcionan evaluaciones ideales. Desafortunadamente, los valores rara vez son tan buenos.



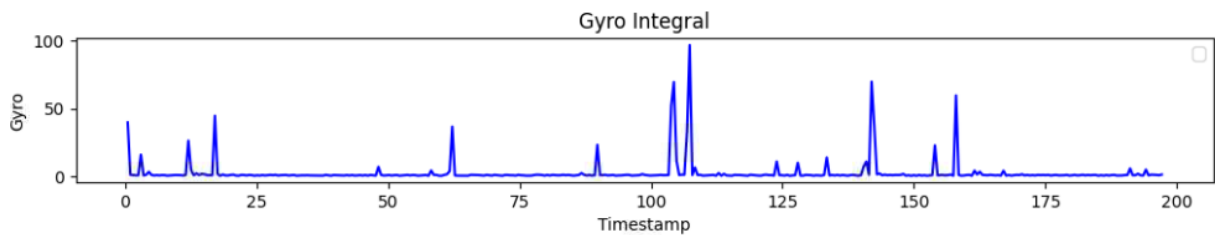
Todos los diagramas

En este ejemplo se pueden ver todos los diagramas. En ese momento, tenía el dedo colocado. Sin embargo, los valores son inutilizables. En las áreas marcadas en negro se puede observar que durante unos minutos había una señal aceptable, aunque no estaba libre de errores. A partir del minuto 17, los valores vuelven a ser inutilizables.



Progresión de la frecuencia cardíaca buena

Aquí hay un ejemplo con un desarrollo más o menos aceptable. Desde el minuto 0 hasta aproximadamente el minuto 13, tenía el dedo colocado. Hay algunos valores atípicos, pero en su mayoría los datos son razonables. Después del minuto 13, quité el dedo.



Progresión del movimiento buena

Aquí se muestra nuevamente el desarrollo del giroscopio durante 200 minutos. Los movimientos y sus intensidades pueden identificarse claramente en los picos.

Hacia el final del proyecto intenté llevar el guante mientras dormía, pero por razones que no entiendo, el sensor de pulso dejó de funcionar correctamente. Esto significa que, independientemente de si coloco mi dedo o, por ejemplo, mi lóbulo de la oreja, los valores permanecen en su mayoría aleatorios. En este punto, la FFT ya no tiene sentido. Rara vez se obtienen valores utilizables, pero esto es una excepción.

Conclusión e ideas para el futuro

En general, puedo decir que este prototipo sigue siendo solo un prototipo. El sensor de pulso es demasiado impreciso para proporcionar resultados útiles. El movimiento puede medirse bien, pero este modelo aún no es realmente utilizable. Definitivamente estoy motivado para continuar con este proyecto en el futuro. Obtener resultados útiles podría ayudarme a entender mejor mi sueño. Para ello, sin embargo, sería necesario un sensor de pulso más confiable. Además, este modelo específico del ESP32 es algo grande; un modelo más pequeño sería una ventaja. También podría agregar más sensores para obtener mejor información sobre las fases del sueño.

Si en el futuro se obtienen valores útiles, podría implementar algo como una alarma inteligente: se establecería una hora límite para despertar y el programa analizaría cuándo estás en una fase de sueño ligero para despertarte en el momento adecuado. Para identificar correctamente las fases del sueño, podría ser útil analizar los resultados con aprendizaje automático o inteligencia artificial.

Bibliografía

- [https://es.wikipedia.org/wiki/Transformada de Fourier](https://es.wikipedia.org/wiki/Transformada_de_Fourier)
- <https://lastminuteengineers.com/pulse-sensor-arduino-tutorial>
- [https://es.wikipedia.org/wiki/Ajuste de curvas](https://es.wikipedia.org/wiki/Ajuste_de_curvas)
- <https://en.wikipedia.org/wiki/MQTT>
- ChatGPT