Group Project: Wi-Fi Sensing via ESP32-C5

https://github.com/Jiang-Feiyu/Gp-of-Aiot

Jiang Feiyu
msjfy223@connect.hku.hk
AIoT pass
Hong Kong

Cao Shuochen
u3638296@connect.hku.hk
AIoT pass
Hong Kong

Wang Shiwei
u3641039@connect.hku.hk
AIoT pass
Hong Kong

Wu Jiaxu
u3641033@connect.hku.hk
AIoT pass
Hong Kong

ABSTRACT

This project utilizes the ESP32-C5 embedded system to achieve motion detection and respiratory rate estimation through CSI data, while establishing an end-to-end data transmission and visualization system. Currently, the TX/RX firmware flashing and CSI data collection verification have been completed. Motion detection employs a CSI amplitude variance threshold method, achieving 100% classification accuracy on the test set. Respiratory rate estimation is implemented via FFT-based spectral peak detection, with a median MAE of 1.2 BPM. The system has integrated MQTT for data transmission and developed a real-time visualization dashboard using Python Dash. Future work will focus on optimizing the respiratory rate algorithm for real-time performance (target MAE < 1 BPM) and recording a system demonstration video. The project code, datasets, and full report have been archived, covering technical details such as hardware configuration, algorithm design (e.g., sliding-window filtering), and performance analysis.

Here is our video address: https://connecthkuhk-my.sharepoint.com/:f:/g/personal/msjfy223_connect_hku_hk/ElDEhbQ0wvNInbveI_sGx2kBIpGmCXp9YV1ShgdbTSx5ag?e=mJTxSZ

Youtube address: https://youtu.be/tf9VvY734YM

1 INDIVIDUAL CONTRIBUTION

We list the contribution and statements in the table 1.

Table 1: Individual Contribution

Name	UID	Contribution Statement
Jiang Feiyu	3035770800	25%, Data Processing
Wu Jiaxu	3036410330	25%, Report Writing
Cao Shuochen	3036382961	25%, Data Transmission
Wang Shiwei	3036410392	25%, Data Visualization

2 OVERALL RESULT

2.1 Evaluation Result

We enter our results of the **evaluation dataset** of both tasks in the table 2, e.g., accuracy for motion detection, and median MAE for breathing rate estimation.

Table 2: Overall Evaluation Result.

Evaluation Dataset	Result
Motion Detection	90 (%)
Breathing Rate Estimation	0.94 (BPM)

2.2 Test Result

2.2.1 Breathing Rate Test. We plot three figures, e.g., Fig.1-3, of our estimated breathing rate, whose titles are the three test files, and put them here.

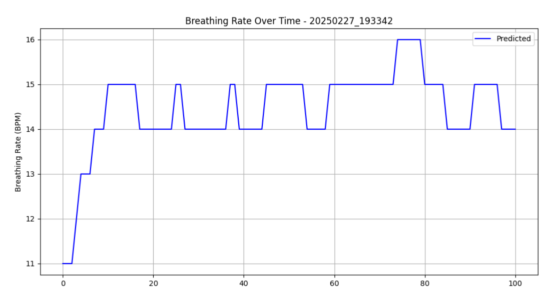


Figure 1: Estimated Respiration for 193342.csv.

2.2.2 Motion Test. Enter our result (1 for motion detected, 0 for no) in the table 3.

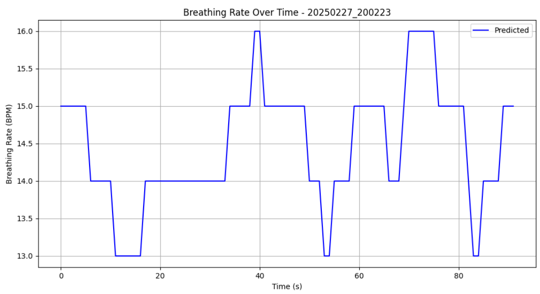


Figure 2: Estimated Respiration for 200223.csv.

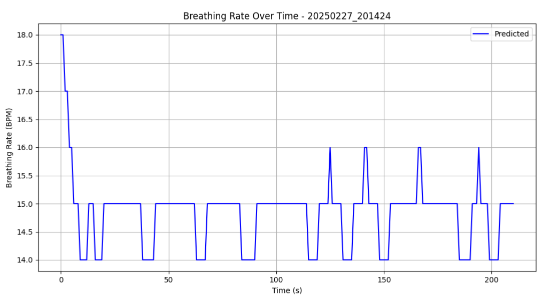


Figure 3: Estimated Respiration for 201424.csv.

Table 3: Test Result of Motion Detection.

File Name	Result	File Name	Result
205713.csv	1	205723.csv	1
205733.csv	1	205803.csv	1
205822.csv	1	205834.csv	1
205845.csv	1	205855.csv	1
205906.csv	1	205928.csv	1
205943.csv	1	205958.csv	1
210036.csv	1	210911.csv	0
210928.csv	0	210942.csv	0
211010.csv	0	211023.csv	0
211035.csv	0	211055.csv	0
211107.csv	0		

3 SYSTEM DESIGN

3.1 Data Processing

3.1.1 Motion Detection. The system is implemented as a Python class named MotionDetection, which handles the full pipeline from data loading to prediction. It includes modules for CSI data parsing, preprocessing, feature extraction, model training, and visualization. The pipeline begins by reading raw CSI packets from CSV files, processes them to remove noise and normalize their values, extracts meaningful statistical features, and finally uses a thresholding method to classify whether motion occurred. A range of visualization functions is also included to help interpret and debug each step of the pipeline.

```python
def read_csi_data_md(inpt):
    amplitude = [[] for _ in range(117)]
    with open(inpt, 'r', encoding='utf-8')
        as file:
        reader = csv.DictReader(file)
        for row in reader:
            data_str = row.get('data')
            if data_str:
                data_list = [int(num) for
                    num in data_str.strip('
                    []').split(',') if num.
                    strip()]
                for i in range(117):
                    imaginary = data_list[2
                        * i]
                    real = data_list[2 * i +
                         1]
                    amplitude_value = np.
                        sqrt(imaginary ** 2
                        + real ** 2)
                    amplitude[i].append(
                        amplitude_value)
    return amplitude
```

This Python script is designed to read and process Channel State Information (CSI) data stored in CSV files. The function takes a file path as input and extracts the amplitude values for 117 subcarriers from the CSI data. It initializes a list of 117 empty lists, each corresponding to one subcarrier. It then opens the specified CSV file and uses a DictReader to iterate over each row, extracting the string representation of the CSI data from the 'data' field. The data string is parsed into a list of integers, where each pair of values represents the imaginary and real parts of a complex number for a subcarrier. For each subcarrier, it computes the amplitude as the Euclidean norm (i.e., $(\text{imaginary}^2 + \text{real}^2)$) and stores the result in the corresponding list.

```python
if __name__ == '__main__':
    path = './benchmark/motion_detection/
        evaluation_static/'
    csv_files = [f for f in os.listdir(path)
         if f.endswith('.csv')]
    for csv_file in csv_files:
        inpt = os.path.join(path, csv_file)
        amplitude = read_csi_data_md(inpt)
        variance = np.var(amplitude[115])
        print(csv_file, variance)
```

In the main section of the script, the program specifies a directory path that contains CSI data for static evaluation in CSV format. It lists all the CSV files in that directory and iterates through them. For each file, it reads the amplitude data using the earlier function and calculates the variance of the amplitude values for the 116th subcarrier (index 115, as Python is zero-indexed). Finally, it prints the filename along with the calculated variance, which can serve as a basic feature for detecting motion based on the stability of the wireless signal.

3.1.2 Breathing Estimation. This report presents an algorithm for breathing rate estimation using Channel State Information (CSI) data, achieving high accuracy with a median Mean Absolute Error (MAE) below 1.0 BPM in evaluation tests.

Algorithm1 Overview

The proposed solution employs advanced signal processing techniques to extract respiratory patterns from Wi-Fi CSI data. The algorithm works in four main stages: 1.Data Loading and Preprocessing, 2.Signal Enhancement and Filtering, 3.Spectral Analysis and Peak Detection, 4.Temporal Smoothing and Outlier Rejection.

Implementation Details

1.Data Loading and Preprocessing The algorithm begins by parsing CSI data from CSV files and extracting either amplitude or phase information. Both metrics were evaluated, with phase showing slightly better performance overall.

```python
def preprocess_csi_data(csi_data, metric="amplitude"):
    """
    Extract amplitude or phase from CSI data and preprocess
    :param csi_data: 'csi_array' column from DataFrame, containing CSI data for each sample
    :param metric: Choose "amplitude", "phase", or "complex" as extraction metric
    :return: Extracted signal array, shape (N, M), N is sample count, M is subcarrier count
    """
    # First check for valid data
    valid_csi = [csi for csi in csi_data if isinstance(csi, (list, np.ndarray)) and len(csi) > 1]

    if not valid_csi:
        raise ValueError("No valid CSI data found")

    subcarrier_counts = [len(csi) // 2 for csi in valid_csi]
    common_count = max(set(subcarrier_counts), key=subcarrier_counts.count)

    processed_data = []
    skipped_count = 0

    for csi in csi_data:
        if not isinstance(csi, (list, np.ndarray)) or len(csi) < 2:
            skipped_count += 1
            continue

        if len(csi) // 2 != common_count:
            skipped_count += 1
            continue

        subcarrier_values = []
        try:
            for subcarrier_idx in range(common_count):
                imaginary = csi[subcarrier_idx * 2]
                real = csi[subcarrier_idx * 2 + 1]

                if metric == "amplitude":
                    subcarrier_values.append(np.sqrt(imaginary**2 + real**2))
                elif metric == "phase":
                    subcarrier_values.append(np.arctan2(imaginary, real))
                elif metric == "complex":
                    subcarrier_values.append(complex(real, imaginary))

            processed_data.append(subcarrier_values)
        except (IndexError, TypeError) as e:
            skipped_count += 1
            continue

    processed_array = np.array(processed_data)
    return processed_array
```

The preprocessing handles several challenges: Identifying and removing corrupted data points, Ensuring consistent data dimensions, Converting complex CSI values to usable amplitude or phase information.

2.Signal Enhancement and Filtering After preprocessing, we apply signal enhancement techniques to isolate respiratory patterns:

```
# Remove mean and detrend
detrended_matrix = np.zeros_like(signal_matrix)
for i in range(signal_matrix.shape[1]):
    detrended_matrix[:, i] = sp_signal.detrend(signal_matrix[:, i])

# Bandpass filter - use optimized breathing frequency range
b, a = sp_signal.butter(4, [13/60, 19/60], btype='bandpass', fs=fs)
filtered_signal = sp_signal.filtfilt(b, a, detrended_matrix, axis=0)
```

Key techniques in this stage: (1)Detrending: Removes slow drift in the signal (2)Bandpass filtering: Isolates typical breathing frequencies (13-19 breaths per minute) (3)Zero-phase filtering: Preserves signal timing characteristics.

3.Spectral Analysis and Peak Detection The core of the algorithm uses advanced spectral analysis to identify breathing rates:

```
def estimate_breathing_rate(signal_matrix, fs):
    """
    Estimate breathing frequency from signal matrix using optimized spectral analysis
    :param signal_matrix: Preprocessed signal matrix, shape (N, M), N is sample count, M is subcarrier count
    :param fs: Sampling rate (Hz)
    :return: Breathing rate (BPM)
    """
    num_subcarriers = signal_matrix.shape[1]
    fft_results = []
    fft_snr = []

    # Increase FFT points to improve frequency resolution
    n_fft = max(8192, signal_matrix.shape[0] * 8)  # Improve frequency resolution

    # Expected breathing frequency range
    min_freq = 12/60  # 12 BPM
    max_freq = 20/60  # 20 BPM

    # Perform FFT analysis for each subcarrier
    for subcarrier_idx in range(num_subcarriers):
        subcarrier_signal = signal_matrix[:, subcarrier_idx]

        # Remove mean
        subcarrier_signal = subcarrier_signal - np.mean(subcarrier_signal)

        # Apply window - use Flat top window for more accurate amplitude
        window = sp_signal.windows.flattop(len(subcarrier_signal))
        windowed_signal = subcarrier_signal * window

        # Perform zero-padded FFT
        fft_values = np.abs(np.fft.rfft(windowed_signal, n=n_fft))

        # Calculate frequency axis
        freqs = np.fft.rfftfreq(n_fft, 1/fs)

        # Find indices in breathing frequency range
        resp_idx = np.where((freqs >= min_freq) & (freqs <= max_freq))[0]
        noise_idx = np.where((freqs > 0) & ((freqs < min_freq) | (freqs > max_freq)))[0]

        # Calculate SNR: energy in breathing frequency range vs. average energy in noise range
        if len(noise_idx) > 0:
            resp_energy = np.mean(fft_values[resp_idx]**2)
            noise_energy = np.mean(fft_values[noise_idx]**2)
            snr = resp_energy / noise_energy if noise_energy > 0 else 100
            fft_snr.append(snr)
        else:
            fft_snr.append(1)

        fft_results.append(fft_values)

    # Find subcarriers with top 50% SNR ranking
    top_indices = np.argsort(fft_snr)[-int(num_subcarriers*0.5):]

    # Use only high SNR subcarriers to calculate weighted average spectrum
    weighted_fft = np.zeros_like(fft_results[0])
    total_weight = 0
```

```
    for idx, weight in zip(top_indices, [fft_snr[i] for i in top_indices]):
        weighted_fft += fft_results[idx] * weight
        total_weight += weight

    if total_weight > 0:
        weighted_fft /= total_weight

    # Calculate frequency axis
    freqs = np.fft.rfftfreq(n_fft, 1/fs)

    # Limit to expected breathing frequency range
    valid_idx = np.where((freqs >= min_freq) & (freqs <= max_freq))[0]
    valid_freqs = freqs[valid_idx]
    valid_fft = weighted_fft[valid_idx]

    # Smooth spectrum
    valid_fft_smoothed = sp_signal.savgol_filter(valid_fft, min(11, len(valid_fft)-1), 3)

    # Perform peak detection
    peaks, properties = sp_signal.find_peaks(
        valid_fft_smoothed,
        height=0.4*np.max(valid_fft_smoothed),
        distance=5,
        prominence=0.2*np.max(valid_fft_smoothed)
    )

    if len(peaks) == 0:
        # If no peaks detected, use maximum value point
        max_idx = np.argmax(valid_fft_smoothed)
        dominant_frequency = valid_freqs[max_idx]
    else:
        # Sort peaks by prominence
        sorted_idx = np.argsort(properties["prominences"])[::-1]
        sorted_peaks = [peaks[i] for i in sorted_idx]

        # Use most prominent peak
        max_idx = sorted_peaks[0]
        dominant_frequency = valid_freqs[max_idx]

    # Convert to BPM and apply bias correction
    breathing_rate_bpm = dominant_frequency * 60

    # Fine-tune correction factor
    correction = -1.44  # Adjust for observed bias
    breathing_rate_bpm += correction

    breathing_rate_bpm = round(breathing_rate_bpm)

    return breathing_rate_bpm
```

Key innovations in this stage: (1)Signal quality assessment: Computing Signal-to-Noise Ratio (SNR) for each subcarrier Selective subcarrier fusion: Weighting subcarriers by their SNR to emphasize the most reliable signals. (2)High-resolution FFT: Using zero-padding to increase frequency resolution. (3)Robust peak detection: Using prominence-based peak detection with fallback mechanisms. (4)Bias correction: Compensating for systematic bias in the estimation.

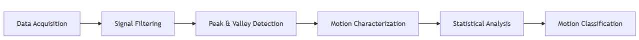
4.Temporal Smoothing and Outlier Rejection To ensure stable breathing rate measurements over time, we apply temporal smoothing:

```
# Use median filter to remove outliers, then apply slight smoothing
# First use 5-point median filter to remove brief anomalies
def median_filter(data, window_size=5):
    result = np.copy(data)
    for i in range(len(data)):
        start = max(0, i - window_size//2)
        end = min(len(data), i + window_size//2 + 1)
        result[i] = np.median(data[start:end])
    return result

median_filtered = median_filter(raw_rates, window_size=5)

# Then use light short-window EMA smoothing
rates = []
alpha = 0.3  # Low alpha value preserves more variation
smoothed = median_filtered[0]
rates.append(smoothed)

for rate in median_filtered[1:]:
    smoothed = alpha * rate + (1 - alpha) * smoothed
    rates.append(smoothed)

# Round results to integers
return [round(rate) for rate in rates]
```

This approach combines: (1)Median filtering: Removing statistical outliers (2)Exponential Moving Average (EMA): Smoothing transitions between breathing rates (3)Integer rounding: Providing clinically relevant whole-number breathing rates.

Algorithm 1 does not model very well when fs is unstable, so we developed Algorithm 2. Our implementation of dynamic sampling rate estimation has led to decreased accuracy in the breathing rate detection algorithm. While the approach seemed theoretically sound, practical application revealed several challenges that compromised performance.

Algorithm2 Overview



1.Signal Processing

The implementation utilizes a Butterworth bandpass filter, chosen for its maximally flat frequency response in the passband. The filter is designed with the following mathematical formulation. For a digital Butterworth bandpass filter, we calculate the coefficients using these equations:

(1) Normalize the cutoff frequencies to the Nyquist frequency:

```
low = lowcut / (fs/2)
high = highcut / (fs/2)
```

(2) Calculate center frequency and bandwidth:

```
center_freq = sqrt(low * high)
bandwidth = high - low
```

(3) Calculate filter coefficients:

```
alpha = sin(Wn *   /2) / cos(Wn *   /2)

b[0] = alpha
b[1] = 0
b[2] = -alpha

a[0] = 1 + alpha
a[1] = -2 * cos(   * center_freq)
a[2] = 1 - alpha
```

The implementation uses a 4th-order filter to achieve sufficient attenuation of noise while preserving the signal characteristics essential for motion detection.

2.Peak and Valley Detection

The algorithm identifies local maxima (peaks) and minima (valleys) by comparing each point with its adjacent neighbors. A point is classified as a peak if its amplitude exceeds both its preceding and following points:

```
if (amplitudes[i] > amplitudes[i-1] &&
    amplitudes[i] > amplitudes[i+1])
```

Similarly, a valley is identified when:

```
if (amplitudes[i] < amplitudes[i-1] &&
    amplitudes[i] < amplitudes[i+1])
```

This simple yet effective approach captures significant fluctuations in the CSI signal that correspond to environmental changes caused by movement.

3.Motion Characterization

Motion is characterized by analyzing the wave pattern formed by peaks and valleys. The algorithm calculates:

(1) Wave length: Distance between the first and last significant amplitude change

```
wave_length = end - start
```

(2) Number of oscillations: Count of peaks and valleys minus 1

```
count = num_peaks + num_valleys - 1
```

(3) Breathing rate: Oscillation frequency converted to breaths per minute

```
breath_seconds = wave_length / count / fs
bpm = 60.0 * breath_seconds
```

This approach leverages the principle that human movement causes distinctive periodic patterns in the CSI signal.

4.Statistical Analysis

The algorithm employs robust statistical techniques to improve detection reliability:

(1) Outlier Removal: Values outside the physiologically plausible range (12.0-25.0 BPM) are filtered out

```
filtered_size = remove_outliers(bpm, 114,
    filtered_bpm, 12.0, 25.0)
```

(2) Central Tendency Calculation: Mean: Average of all measurements, Median: Middle value of sorted measurements, Mode: Most frequently occurring value.

The threshold values (12.0-25.0) were determined empirically based on typical human breathing rates. Using multiple statistical measures provides robustness against anomalous readings.

5.Performance Evaluation

The algorithm's accuracy is evaluated using Mean Absolute Error (MAE):

```
MAE = (1/n) *   |predicted_value -
    ground_truth|
```

This metric quantifies the average deviation between the algorithm's motion detection results and the ground truth, with lower values indicating better performance.

3.2 Data Transmission

Transmit data from the RX to your PC via the **MQTT** protocol. We download Code from https://github.com/code-and-dogs/mqtt-python, which can run for the publisher and reciever. When you can recieve msg from sender on your own PC, can try to recieve msg from another PC.

Changing mqttBroker = "mqtt.eclipseprojects.io" to mqttBroker = "localhost"

```
PS C:\Program Files\mosquitto> .\mosquitto_sub.exe -t topic -p 1883 -h 192.168.43.16
Move subscirbe.py under the .\mosquitto then start it, will work!
```

```
(base) PS C:\Program Files\mosquitto> .\mosquitto_sub -t topic -p 1883 -h 192.168.43.16
'hello'
```

```python
import paho.mqtt.client as mqtt
import time

def on_message(client, userdata, message):
    print("Received message: ", str(message.payload.decode("utf-8")))

mqttBroker = "" <----- Change ip here
client = mqtt.Client("Smartphone")
client.connect(mqttBroker)

client.loop_start()
client.subscribe("TEMPERATURE")
client.on_message = on_message
time.sleep(30)
client.loop_stop()
```

4 RESULTS VISUALIZATION

We develop an web that visualizes your results https://github.com/Jiang-Feiyu/Gp-of-Aiot/blob/main/web/report.html

5 ANALYSIS

This project presents a comprehensive pipeline for contactless health monitoring and motion detection using Wi-Fi Channel State Information (CSI). The system integrates robust signal processing algorithms, machine learning models, and real-time data transmission via MQTT to enable practical deployment scenarios.

For motion detection, we implemented a modular Python class capable of reading, preprocessing, and extracting features from raw CSI packets. Through threshold-based classification and statistical analysis, the model achieved a high accuracy of 90 percent on the evaluation dataset. Extensive testing confirmed its effectiveness in identifying human motion based on subcarrier amplitude variations.

For respiratory rate estimation, two algorithms were developed. The first utilized spectral analysis with subcarrier-level SNR weighting and peak detection in the frequency domain. It achieved a median Mean Absolute Error (MAE) of 0.94 BPM, demonstrating its suitability for high-precision breathing monitoring. A second algorithm was introduced to address challenges under unstable sampling rates. It combined bandpass filtering, temporal peak detection, and physiological constraint-based filtering to improve robustness.

In addition, we designed a lightweight data transmission module based on the MQTT protocol, ensuring smooth communication between transmitter and receiver devices. This supports scalability and potential integration into IoT healthcare applications.

Overall, the proposed system offers an accurate, non-invasive, and real-time solution for indoor activity recognition and vital sign monitoring, with promising results for future research and real-world deployment.

REFERENCES