LOW-COST VIRTUAL FENCING FOR LIVESTOCK MANAGEMENT

A Design Project Report

Presented to the School of Electrical and Computer Engineering of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering, Electrical and Computer Engineering

Submitted by

Ang Chen and Yiyang Zhao

MEng Field Advisor: Dr. Hunter Adams

Degree Date: August 2024 and January 2025

Abstract

Master of Engineering Program

School of Electrical and Computer Engineering

Cornell University

Design Project Report

Project Title: Low-Cost Virtual Fencing for Livestock Management

Author: Ang Chen and Yiyang Zhao

Abstract: This project introduces a revolutionary virtual fencing system designed to enhance livestock management through advanced technology, significantly reducing the need for physical fencing. Utilizing the Raspberry Pi Pico W and the Adafruit Flora GPS Module, the system provides real-time geolocation of livestock across diverse terrains, while also monitoring behavior and health through an integrated Inertial Measurement Unit (IMU) and a temperature sensor. The system employs Wi-Fi and WebSocket for seamless data transmission to a user-friendly web interface, allowing farmers to monitor and manage livestock remotely. Future enhancements include expanding network coverage via LoRa technology, supporting multiple fencing boundaries, and integrating non-invasive guidance mechanisms to steer animals within designated perimeters.

Executive Summary

This project presents an approach to managing livestock using virtual fencing technology, significantly reducing costs and environmental impact. The system leverages the Raspberry Pi Pico W and Adafruit Flora GPS Module to monitor livestock in real-time without physical barriers.

The system integrates GPS tracking, an Inertial Measurement Unit (IMU) for movement analysis, and a temperature sensor for health monitoring, all communicated via Wi-Fi to a user-friendly web interface. This interface enables easy setup of geofences and live tracking of animal locations and health.

Key achievements in this project includes: (1) dynamic geofencing: enables real-time adjustments of livestock boundaries through a web interface; (2) comprehensive monitoring: provides continuous data on animal health and location, improving herd management; (3) field validation: successfully tested at Cornell University, demonstrating reliable functionality and sensor accuracy.

This project illustrates a viable alternative to traditional livestock fencing methods by combining advanced technology with practical applications, thus enhancing agricultural productivity and sustainability. Future improvements will focus on expanding network coverage with LoRa technology, enhancing geofencing capabilities, and broadening health monitoring features to include more vital signs.

Acknowledgement

We are immensely grateful to Dr. Hunter Adams for his profound expertise and invaluable guidance throughout the duration of this project. His contributions have been pivotal to our success. Additionally, we extend our heartfelt thanks to the ECE department of Cornell University for their generous support. The equipment and materials provided by them was crucial in enabling our project, and their backing greatly enhanced the quality of our work. This project would not have reached its potential without the exceptional commitment and support from both our esteemed advisor and the university.

Introduction

The management of livestock has historically been a labor-intensive process fraught with inefficiencies and high costs, particularly when it comes to fencing. Traditional physical fences not only involve significant initial investments and ongoing maintenance costs but also impact the environment negatively by altering natural habitats. Additionally, they do not offer flexibility in managing pastures and monitoring animal behaviors and health.

With advances in technology, particularly in IoT and wireless communication, virtual fencing presents a promising alternative. Virtual fences offer a dynamic solution by allowing for the creation and adjustment of boundaries without physical barriers. This project is dedicated to creating an affordable and effective virtual fencing system for farms, a technology that has the potential to transform livestock management.

The primary objective of this project is to design and implement a virtual fencing system that utilizes GPS for geolocation to monitor livestock. The system is equipped with a wearable device that not only helps in tracking the animals but also monitors their health status via various sensors, including an Inertial Measurement Unit (IMU) for activity and a temperature sensor for health monitoring. By integrating these technologies, the project aims to provide a comprehensive solution that not only keeps the livestock within designated boundaries but also enhances the overall management through behavioral and health monitoring.

The system leverages Wi-Fi and WebSocket technology to wirelessly connect the wearable device on livestock to a central server. This setup enables real-time data transmission and

monitoring, ensuring that all animal movements are tracked accurately. The use of Wi-Fi facilitates a robust and cost-effective communication network that is essential for the deployment of virtual fencing in expansive rural areas.

Moreover, the project introduces a user-friendly web interface that allows farmers to easily configure and monitor the virtual fencing system. Through this interface, farmers can define geofenced areas by setting central points and radiuses. And it displays the location of each animal on this live map, which shows animals within or outside boundaries in blue and red icons. Furthermore, it has line charts that show animals' movement angles and body temperatures. This interface is designed to be intuitive, ensuring that even users with limited technical expertise can effectively manage their livestock.

Background and Problem Statement

2.1 Background

Effective livestock management is crucial for maximizing agricultural productivity and ensuring sustainability. Traditional methods of managing livestock, such as physical fencing, are widely used but come with a host of limitations. These fences restrict the natural movement of animals, require significant maintenance, and lack adaptability in terms of grazing management. Such physical constraints not only inhibit the efficient utilization of pasture but also contribute to environmental degradation through soil erosion and reduced biodiversity. Moreover, physical fences are unable to provide insights into animal behavior and health, which are critical for managing disease outbreaks and optimizing breeding programs.

2.2 Problem Statement

The advent of precision agriculture technologies has ushered in a new era of farming innovations, among which virtual fencing stands out as a transformative approach. Utilizing GPS technology and wireless communication, virtual fencing systems manage livestock movements without the need for physical barriers. This method offers a dramatic reduction in environmental impact and operational costs associated with traditional fencing. It enables farmers to dynamically adjust grazing areas and boundaries with ease, directly from a digital interface, without the physical labor of moving fences.

Despite the advantages, the implementation of virtual fencing systems is not without challenges. The primary hurdles include the complexity of setting up such systems and the robust infrastructure they require. There is also a need for systems that not only prevent animals from

straying beyond their boundaries but also gently guide them back to safety without causing stress or harm. This requires integrating stimulus-response mechanisms that are subtle yet effective in directing animal movement within designated geofences.

Furthermore, the proposed system aims to enhance traditional fencing capabilities by incorporating advanced monitoring features. These include real-time tracking of each animal's location, which aids in theft prevention and swift recovery of stray livestock. Health monitoring sensors can measure vital signs such as body temperature and heart rate, providing early warnings of health issues. Additionally, integrating an Inertial Measurement Unit (IMU) can track and analyze the posture and movement patterns of each animal, offering insights into their physical well-being and behavior.

This project is dedicated to developing a scalable, efficient, and user-friendly virtual fencing system. By leveraging cutting-edge technologies, the system is designed for flexible and precise livestock management. With a focus on continuous improvement through iterative testing and feedback, the project seeks to deliver a comprehensive solution that meets the diverse needs of modern agriculture, enhancing the sustainability and profitability of farming operations across varied geographical landscapes.

System Design and Implementation

3.1 Microcontroller for Integration

A microcontroller is a compact integrated circuit designed to govern specific operations in an embedded system through real-time data processing and control. Serving as the cornerstone in the virtual fencing system, microcontrollers offer several indispensable advantages:

- Real-time Processing: Essential for tasks that require immediate responses, such as localizing each animal's real-time location and activating boundary alerts or health monitoring interventions.
- Versatility: They support multiple functionalities in a single environment, reducing the need for multiple components and thereby simplifying the system architecture.
- Cost and Energy Efficiency: Microcontrollers are generally low-cost and consume minimal power, which is crucial for sustainable, long-term deployment in agricultural settings where resources are often limited.

For this project, the Raspberry Pi Pico W (Figure 1) was selected for its distinctive attributes that align well with virtual fencing system requirements:

- Affordability: It offers substantial computational power at a low cost, making the project economically viable on a larger scale.
- Low Power Consumption: It is designed to operate efficiently on low power, which is critical for devices deployed in the field that may have limited access to power sources.
- Built-in Connectivity: With integrated Wi-Fi and Bluetooth capabilities, the Pico W can communicate wirelessly, enabling data transmission to central servers or directly to users' devices without physical infrastructure.

- Integrated Temperature Sensors: The inclusion of a temperature sensor facilitates direct monitoring of livestock health by measuring external body temperatures, enhancing the system's functionality without additional hardware costs.

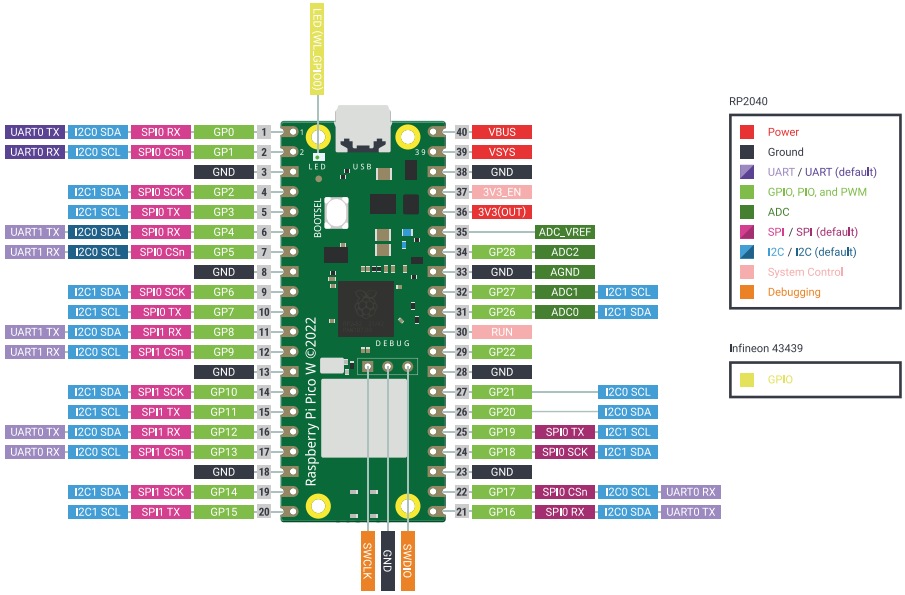

Figure 1. Raspberry Pi Pico W Pinout

(Raspberry Pi Foundation. *Raspberry Pi Pico Documentation*, https://www.raspberrypi.com/documentation/microcontrollers/raspberry-pi-pico.html.)

3.2 GPS for Localization

GPS, or Global Positioning System, is a satellite-based navigation system that allows any user with a GPS receiver to determine their exact location anywhere on Earth. GPS receivers collect signals from satellites in orbit, which are used to precisely pinpoint the receiver's location through a process known as trilateration. In the context of virtual fencing for livestock management, GPS is crucial for continuously monitoring the location of each animal. This technology allows farmers to ensure animals stay within defined geofence boundaries and to

track their movements in real time, enhancing the management of grazing patterns and the overall welfare of the livestock.

The Adafruit Flora Wearable Ultimate GPS Module (Figure 2) was selected for this project due to several advantageous features:

- Compactness and Wearability: Its small size and ease of integration with wearables make it ideal for attaching to livestock without causing discomfort.
- High Sensitivity and Low Power Consumption: This module provides accurate location tracking (-165 dBm sensitivity, less than 3 meters for accuracy, 10 Hz updates, 99 search channels) while consuming minimal power (30 mA current), crucial for long-term field deployment without frequent maintenance.
- Built-in Data Logging: The module can store location data internally, which is useful in areas with intermittent communication or for analyzing movement patterns retrospectively.

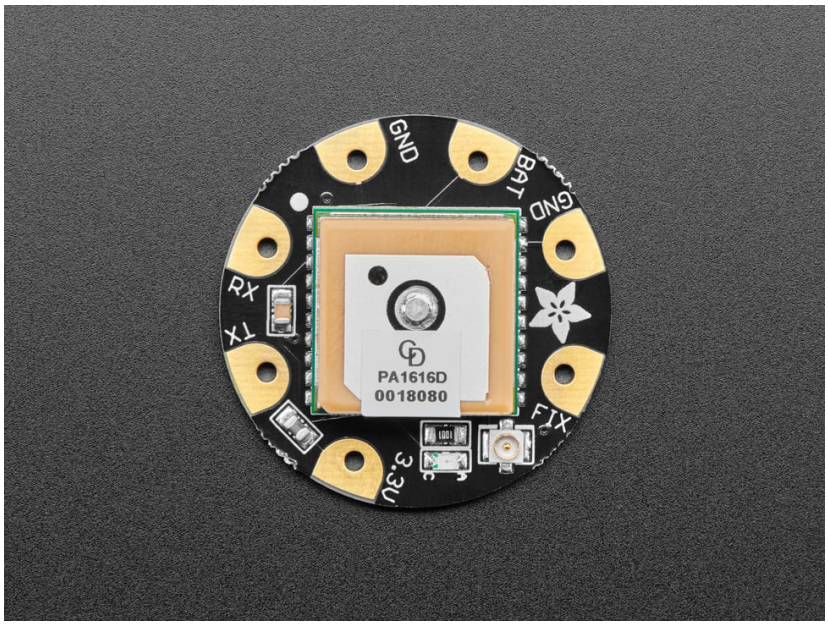

Figure 2. Adafruit Flora Wearable Ultimate GPS Module Pinout

The Adafruit Flora Wearable Ultimate GPS Module is integrated with the Raspberry Pi Pico W through a UART interface, which is essential for real-time geolocation tracking.

For the UART connections, the TX pin of the GPS module is connected to URAT1 RX pin (pin 7) on the Pico W, and the RX pin of the GPS module is connected to URAT1 TX pin (pin 8) on the Pico W. For the power and ground connections, the 3.3V pin of the GPS module is connected to 3V3(OUT) pin (pin 36) on the Pico W, and the GND pin of the GPS module is connected to GND pin (pin 3) on the Pico W.

For software setup using MicroPython, the UART communication on the Pico W is initialized with a baud rate of 9600, which matches the default baud rate of the GPS module. This is done using the “machine.UART” class, specifying the UART channel, baud rate, and the pins used for TX and RX. And a loop is created that continuously reads from the UART. When data is available, it reads the incoming bytes until a newline character is detected, signaling the end of a GPS data sentence.

For data processing, the GPS module transmits NMEA sentences, which provide various pieces of location data. The key sentence for this application is “$GNGGA”, which includes important information such as latitude, longitude, and altitude. Data read from the UART is processed line by line. Each line is checked to ensure it starts with `$GNGGA` to confirm it's a valid GPS fix data sentence. The sentence is split into components, and the latitude and longitude are extracted and converted from the NMEA format (degrees and minutes) to decimal degrees for easier computation and integration with mapping tools. This conversion handles different hemisphere

indicators by adjusting the sign of the result based on whether the coordinate is in the southern or western hemispheres. Once parsed, the latitude, longitude, and altitude data are packaged into a JSON object and transmitted real-time location of each animal over a WebSocket connection to a user interface.

3.3 Inertial Measurement Unit (IMU) for Movement Detection

An Inertial Measurement Unit (IMU) is a critical component in various applications, especially those requiring motion sensing and orientation tracking. It typically comprises accelerometers and gyroscopes, and sometimes magnetometers, to provide comprehensive data on movement and direction. IMUs measure linear acceleration and angular velocity, which can be processed to determine orientation, trajectory, and gravitational forces acting on the device.

To create the virtual fencing system for livestock management, the IMU is indispensable for tracking the orientation and movement patterns of animals. It offers the capability to monitor behavioral patterns and detect unusual activities that could indicate distress, health issues, or potential breaches of geofenced areas. The real-time data generated by IMUs enables proactive management decisions and immediate responses to ensure the safety and well-being of the livestock.

The Adafruit MPU-6050 6-DoF Accel and Gyro Sensor (MPU-6050) (Figure 3)was selected for several reasons:

- Precision: It combines a 3-axis gyroscope and a 3-axis accelerometer on the same silicon die, providing high precision and motion sensitivity.

- Integration: The MPU-6050 supports I2C communication, which simplifies integration with the Raspberry Pi Pico W.
- Cost-Effectiveness: It offers a cost-effective solution with low power consumption (3.6 mA for gyro, 3.8 mA for gyro + accel), which is crucial for deploying devices in field conditions where power efficiency is essential.

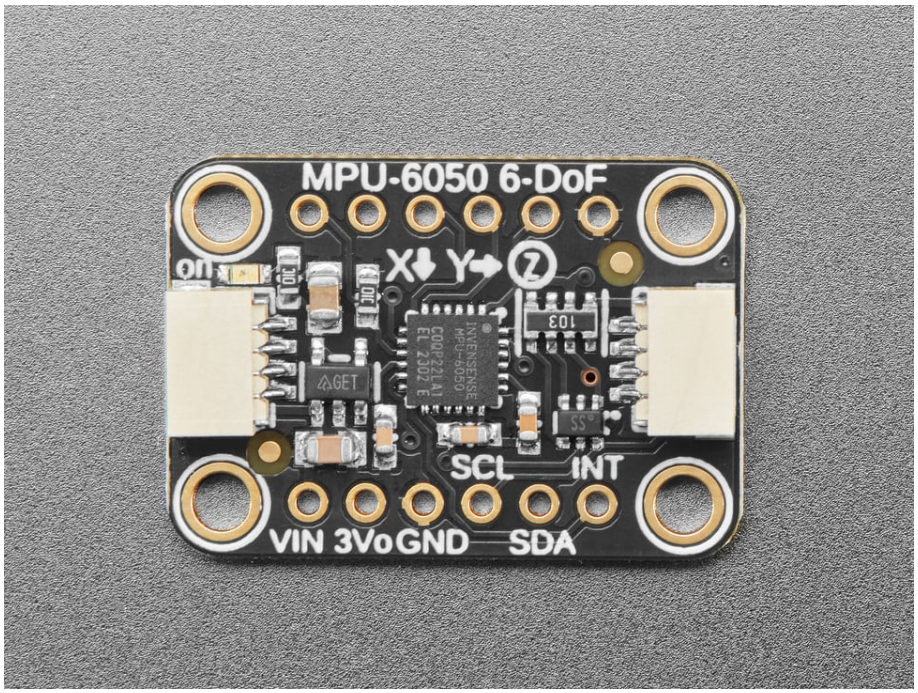

Figure 3. Adafruit MPU-6050 6-DoF Accel and Gyro Sensor Pinout

The IMU module is interfaced with the Raspberry Pi Pico W using the I2C communication protocol, facilitated by MicroPython's “machine” and “busio” libraries. The MPU-6050 is initialized by connecting the SDA and SCL pins of the MPU-6050 to I2C0 SDA (pin 1) and I2C0 SCL (pin 2) pins on the Pico W, and powering the module via the Pico W’s 3V3(OUT) (pin 36) and GND (pin3) pins.

For software initialization, in the IMU class, the I2C bus is initiated on the designated pins, and the MPU-6050 is configured through this bus. This setup allows the MPU-6050 to start sending data about the animal's movements immediately upon powering up.

For data processing, The IMU continuously acquires data on the animal’s orientation by calculating the angle of movement relative to the horizontal plane. This is done using the accelerometer readings to determine the tilt and roll of the device. The angle is then computed using the arctangent of the ratio between the acceleration in the Z-axis and the Y-axis, converted from radians to degrees for usability. The calculated angle also helps in understanding the posture and activity level of the animal, which is crucial for detecting behaviors such as grazing, resting, or potential escape attempts.

3.4 Temperature Sensor for Health Monitoring

The temperature sensor integrated within the Raspberry Pi Pico W plays a crucial role in the virtual fencing system, serving as a primary tool for monitoring the health status of livestock. This sensor, a built-in feature of the Pico W, simplifies the hardware setup and enhances the system's capability to measure critical environmental and biological parameters.

The Raspberry Pi Pico W is equipped with an onboard temperature sensor connected through an Analog-to-Digital Converter (ADC). This sensor operates by measuring the voltage across a thermistor—a type of resistor whose resistance varies significantly with temperature. In our setup, the temperature sensor is accessed via one of the ADC channels specifically reserved for reading the internal temperature.

The temperature sensor is accessed through the ADC channel 4 of the Pico W using MicroPython's machine library with the command "machine.ADC(4)". This channel is specifically mapped to the onboard temperature sensor. The reference voltage of the ADC on the Pico W is 3.3 V. The raw ADC values, which range from 0 to 65535, are converted to an actual voltage value. This conversion is crucial as the raw ADC value itself isn't directly interpretable as a temperature. The conversion formula used is:

$$Voltage = \frac{ADC\ Reading}{65535} \times 3.3\ V$$

The temperature is then calculated from the voltage using the formula derived from the thermistor characteristics, which is embedded within the Pico W's design. Specified in the Pico W datasheet, the formula used to calculate the temperature in degrees Celsius from the voltage reading is:

$$Temperature = 27 - \frac{Voltage - 0.706}{0.001721}$$

The sensor continuously reads the current temperature in real time, which is crucial for monitoring the ambient conditions affecting the livestock. Alongside positional data from the GPS and orientation data from the IMU, temperature readings are formatted into JSON and transmitted over WebSocket to a centralized server or directly to the farmer's monitoring interface.

3.5 Wi-Fi for Wireless Communication

Wi-Fi and Bluetooth are the two primary wireless communication technologies available for this project. Each has distinct characteristics suitable for specific applications within the project. Wi-Fi is preferred for its extensive range and higher data transfer capabilities, crucial for covering

large agricultural fields. Wi-Fi's ability to integrate into existing network infrastructures allows for remote access from various devices, facilitating broader connectivity without additional hardware. Bluetooth offers a convenient short-range connection and is typically easier to configure for simple, point-to-point or small network applications. It consumes less power compared to Wi-Fi, making it suitable for devices where battery life is a concern.

While Bluetooth could be considered for smaller, more localized setups or for specific low-power applications, its limitations in range and data throughput make Wi-Fi the superior choice for the scope of this project:

- Greater Range: Wi-Fi covers a larger area, which is vital for tracking livestock across extensive pastures.
- Higher Data Capacity: Supports streaming of extensive data required for real-time tracking and monitoring.
- Network Flexibility: Easily connects to the internet, enabling cloud data integration and remote monitoring capabilities.

WebSocket is used to establish a persistent, full-duplex communication channel over a single TCP connection, allowing for real-time data exchange between the client’s web interface and the Raspberry Pi Pico W server. This is particularly advantageous for applications that require continuous data flow and minimal latency, such as tracking and managing livestock in real-time.

To configure the Pico W to connect to the local Wi-Fi network, the Wi-Fi module in the Pico W is initialized to operate in station mode, where it connects to an available wireless network.

Relevant network credentials (SSID and password) are programmed into the device to enable automatic connection upon startup.

A WebSocket server is set up using the Microdot library, which provides a lightweight yet robust framework for handling HTTP and WebSocket protocols on constrained devices like the Raspberry Pi Pico W. The “microdot_asyncio_websocket” module is used to define WebSocket routes in the Microdot application. This module listens for incoming WebSocket connections from the client’s browser.

The server continuously reads data from the GPS, IMU, and temperature sensors. This data is formatted into JSON and sent to the client through the WebSocket connection, ensuring that the user interface on the web browser remains updated in real-time. The WebSocket server also listens for commands from the web interface, such as turning on an LED when an animal exits the geofenced area. These commands are processed as soon as they are received to enact immediate responses, enhancing the reactive capabilities of the virtual fencing system.

3.6 Web Interface for Overall Management

In modern agricultural practices, especially in precision livestock management, accessibility and ease of configuration are paramount. A web interface provides a centralized, accessible platform for farmers to monitor and manage their livestock remotely. This approach aligns with the principles of precision agriculture, which emphasizes the use of technology to increase efficiency and effectiveness.

The system features a user-friendly web interface that allows seamless interaction with the virtual fencing system. Through this interface, farmers can:

- Define Geofenced Areas: By setting a central point (latitude and longitude) and specifying its radius, users can easily configure the boundaries within which livestock can roam. This is especially useful for rotational grazing, allowing for flexible management of grazing areas without the need for physical barriers.
- Monitor Animal Locations: The interface dynamically displays the location of each animal on a live map. Animals within the boundary are indicated in blue, while those outside are marked in red (figure 4), providing clear visual feedback on their current status.
- View Real-Time Data: Movement patterns and health statuses are visualized through line charts, offering insights into the wellbeing and behavior of the livestock. This data helps in making informed decisions regarding animal health and pasture management.

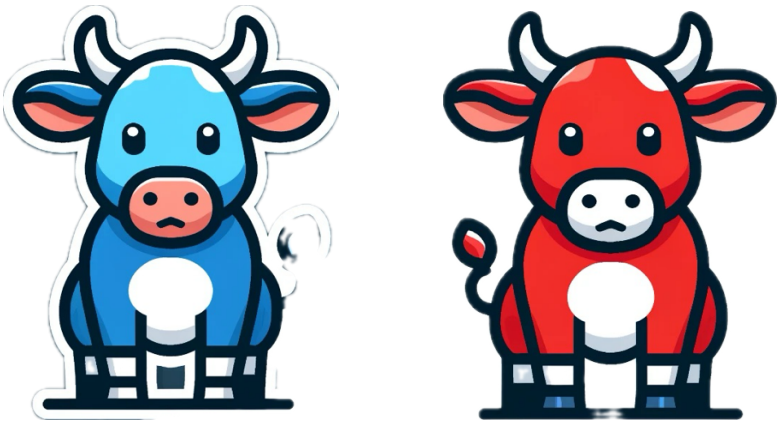

Figure 4. Animal Icons on the Map, Blue for Inside (Left), Red for Outside (Right)

The web interface is developed using modern web technologies that ensure compatibility and responsiveness across devices. Here's how it integrates with the overall system:

- "Leaflet.js" for Mapping: The live map is powered by "Leaflet.js", an open-source JavaScript library for mobile-friendly interactive maps. It is used to display the geographical location of each animal, updating in real-time as data is received.
- "Plotly.js" for Data Visualization: Real-time monitoring of animal movement and health metrics is implemented using "Plotly.js", which provides sophisticated charting tools that update dynamically as new data flows in from the sensors.
- WebSocket Communication: The interface uses WebSocket for real-time, two-way interaction between the server (running on the Raspberry Pi Pico W) and the client's web browser. This setup ensures that data from the GPS and sensors is streamed directly to the web interface without noticeable delays.

Testing and Results

The testing of our virtual fencing system was conducted at Phillips Hall, Cornell University. The setup involved the integration of the Raspberry Pi Pico W, an Adafruit Flora Wearable GPS Module, an IMU module for motion detection, and a LoRa module for potential future enhancements (figure 5). The Raspberry Pi Pico W was connected to the internet via a mobile phone's Wi-Fi hotspot, ensuring both the microcontroller and the laptop monitoring the system were on the same network.

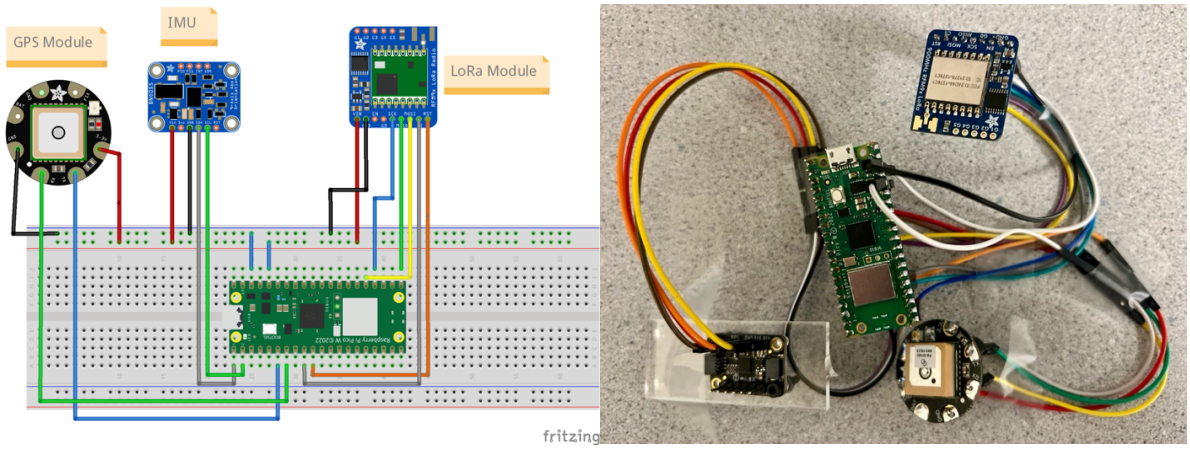

Figure 5. Layout and Wiring for the Whole System

The system's web interface was accessed by navigating to the Wi-Fi IP address on a laptop. This interface dynamically displayed the virtual geofence set around Phillips Hall with adjustable radius settings.

The GPS module required approximately one to two minutes to establish a connection with satellites and retrieve initial data. This slight delay was expected due to the time taken by GPS modules to acquire satellite signals upon startup. Once the connection was established, the module accurately updated the location data on the map in real-time.

Initially, the geofence radius was set to 100 meters. The interface showed a live map where the location of a hypothetical livestock (represented by the blue icon) was indicated in blue when within the boundary (Figure 6). To test the responsiveness of the system to boundary breaches, the geofence radius was reduced to 10 meters. This change was instantly reflected on the map, turning the icon red to signify that the location was out of the designated boundary (Figure 7). This immediate visual feedback was pivotal in demonstrating the system's effectiveness in real-time monitoring and boundary control.

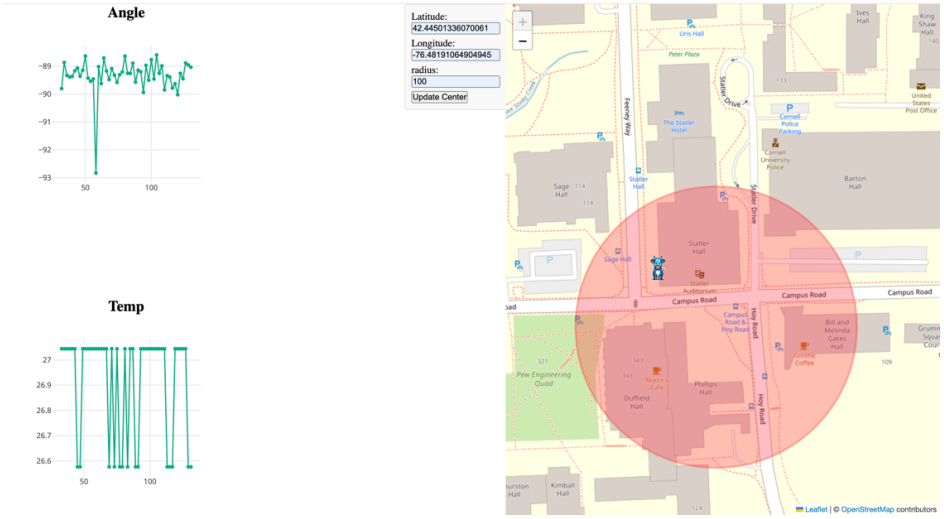

Figure 6. Web Interface When Center Point Is Phillips Hall and Radius Is 100 Meters

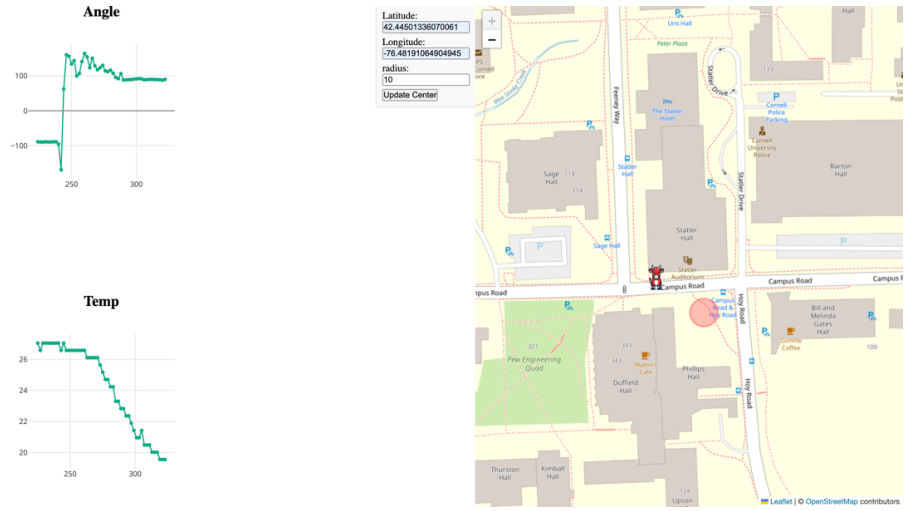

Figure 7. Web Interface When Center Point Is Phillips Hall and Radius Is 10 Meters

The IMU sensor, crucial for detecting the orientation and movement of livestock, was rigorously tested to ensure its accuracy. When the direction of the IMU sensor was altered during the tests, the corresponding changes were instantly reflected on the angle line chart displayed on the web interface (Figure 6). This immediate responsiveness demonstrated the sensor's capability to accurately track and display orientation changes, which is essential for monitoring livestock movements within the geofenced area.

The built-in temperature sensor on the Pico W was tested to assess its sensitivity and accuracy in different environmental conditions. During the tests, moving the system from inside a building to an outdoor setting resulted in significant temperature changes, which were promptly captured and displayed on the temperature line chart on the web interface (Figure 7). This real-time responsiveness underscores the sensor's effectiveness in monitoring the environmental and health-related temperature changes affecting the livestock.

During testing, all components functioned as expected, demonstrating the system's capability to monitor livestock within and beyond predefined boundaries effectively. However, the initial delay in GPS data retrieval highlighted an area for potential improvement, possibly by optimizing the satellite acquisition process or using a more advanced GPS module for quicker startup times.

The tests confirmed that the virtual fencing system could provide robust real-time monitoring and control, essential for modern livestock management. The successful demonstration at Phillips Hall underscored the system's utility in real-world conditions, providing a foundation for further enhancements and wider application.

Conclusion and Future work

6.1 Conclusion

This project has successfully demonstrated the feasibility and effectiveness of a virtual fencing system designed for advanced livestock management. By leveraging a combination of GPS tracking, inertial measurement units (IMU) for motion detection, and temperature monitoring, we have developed a wearable technology that not only minimizes the environmental impact but also reduces the costs associated with traditional fencing methods. The implementation of Wi-Fi technology allows for efficient long-distance data transmission, connecting each wearable device directly to a central user interface. This interface provides farmers with real-time, actionable insights through an interactive map and essential health statistics for each animal, thereby enhancing decision-making and overall herd management.

Our virtual fencing system stands as a testament to the potential of integrating advanced technology into agricultural practices, promoting a more sustainable and productive approach to livestock management. It successfully addresses the challenges of traditional livestock management by offering flexibility in boundary setting, continuous health monitoring, and precise location tracking, thereby ensuring both the safety of the animals and the sustainability of the farming practice.

6.2 Future Work

Looking ahead, the project is poised for several strategic enhancements to further improve its efficiency, scalability, and scope:

- Expand Network Capabilities: Integrate LoRa technology to enable a more robust and extensive wireless sensor network. This will facilitate connectivity over larger farm areas and enhance data transmission efficiency, ensuring reliable communication even in remote or challenging terrains.

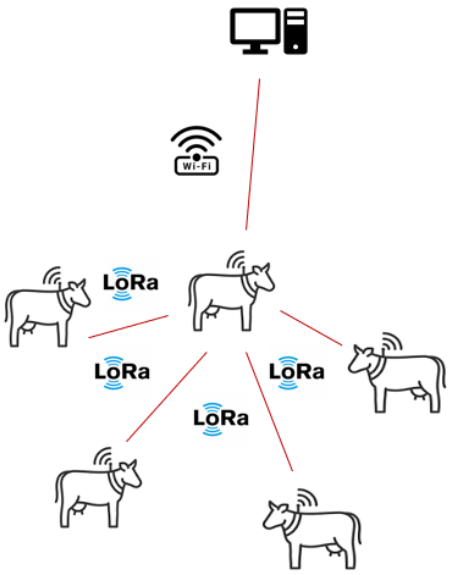

Figure 8. Wireless Communication with Wi-Fi and LoRa

- Advanced Geofencing Features: Develop the system's capabilities to support multiple and diverse geofencing configurations. This will allow for more customized boundary settings tailored to different pasture layouts and farming needs, providing greater control and flexibility in livestock management.

- Enhanced Health Monitoring: Broaden the scope of the health monitoring features to include more vital signs and behavioral indicators. This expansion will enable a more comprehensive assessment of livestock health, aiding in early detection of potential health issues and improving the welfare of the animals.

- Proactive Herd Management: Implement stimulus-response mechanisms that effectively guide animals back within geofence boundaries. These mechanisms will reduce the likelihood of animals straying, ensuring their safety and well-being without causing distress.

By pursuing these future developments, the project aims to set a new standard for technology-driven livestock management, furthering its mission to deliver innovative solutions that meet the evolving demands of modern agriculture.

Appendices

main.py:

```
from microdot_asyncio import Microdot, Response, send_file
from microdot_utemplate import render_template
from microdot_asyncio_websocket import with_websocket
from machine import Pin
from temp_sensor import Temp
from imu import IMU
from GPS import GPS
import time
import json
import asyncio



# Initialize MicroDot
app = Microdot()
Response.default_content_type = 'text/html'

#Initioalize LED
led = machine.Pin("LED", machine.Pin.OUT)

# IMU and temperature sensor
temp = Temp()
imu = IMU()

#Initialize GPS
gps = GPS(tx_pin=4, rx_pin=5)

# root route
@app.route('/')
async def index(request):
    return render_template('index.html')


@app.route('/ws')
@with_websocket
async def read_sensor(request, ws):
    while True:
        time.sleep(1)
        location = gps.location  # Call to get current location
        if location:
            latitude, longitude, altitude = location
            data = {
                'angle': str(imu.get_angle()),
```

```
                'temp': str(temp.get_temperature()),
                'latitude': str(latitude),
                'longitude': str(longitude)
            }
            json_data = json.dumps(data)
            await ws.send(json_data)

            # Handle messages with a timeout to control the LED
            try:
                message = await asyncio.wait_for(ws.receive(), timeout=0.5)  # Timeout
after 1 second
                if message:
                    handle_led_command(message)
            except asyncio.TimeoutError:
                print("No command received within timeout period.")
            except Exception as e:
                print(f"Error while receiving message: {e}")

def handle_led_command(message):
    try:
        command = json.loads(message)
        if command.get("led") == "on":
            led.on()
        elif command.get("led") == "off":
            led.off()
    except Exception as e:
        print(f"Error handling command: {e}")

@app.route("/static/<path:path>")
def static(request, path):
    if ".." in path:
        return "Not found", 404
    return send_file("static/" + path)

# shutdown
@app.get('/shutdown')
def shutdown(request):
    request.app.shutdown()
    return 'The server is shutting down...'


if __name__ == "__main__":
    try:
        app.run()
    except KeyboardInterrupt:
        pass
```

boot.py:

```
import time
import network

ssid = 'Ang'
password = 'bubblebang'

wlan = network.WLAN(network.STA_IF)
wlan.active(True)
wlan.connect(ssid, password)

# Wait for connect or fail
max_wait = 10
while max_wait > 0:
    if wlan.status() < 0 or wlan.status() >= 3:
        break
    max_wait -= 1
    print('waiting for connection...')
    time.sleep(1)

# Handle connection error
if wlan.status() != 3:
    raise RuntimeError('network connection failed')
else:
    print('connected')
    status = wlan.ifconfig()
    print( 'ip = ' + status[0] )
```

GPS.py:

```
import machine

class GPS:
    def __init__(self, tx_pin, rx_pin, baudrate=9600):
        self.uart = machine.UART(1, baudrate=baudrate, tx=machine.Pin(tx_pin),
rx=machine.Pin(rx_pin))
        self.buffer = b''

    def read_line(self):
        while self.uart.any():
            char = self.uart.read(1)
            if char == b'\n':
                line = self.buffer
                self.buffer = b''  # Clear the buffer after reading a line
                return line.strip()
            elif char:
```

```
                self.buffer += char
        return None

    def parse_gngga(self, data):
        try:
            sentence = data.decode('utf-8').strip()
            parts = sentence.split(',')
            if parts[0] == '$GNGGA' and len(parts) > 14 and parts[6] != '0':
                latitude = self.convert_to_decimal(parts[2], parts[3], True)
                longitude = self.convert_to_decimal(parts[4], parts[5], False)
                altitude = parts[9] + ' ' + parts[10]
                return latitude, longitude, altitude
        except ValueError:
            print("Parsing error: invalid data")
        return None, None, None

    def convert_to_decimal(self, coordinate, direction, is_latitude):
        if coordinate:
            # Latitude degrees are two digits, longitude are three
            degree_len = 2 if is_latitude else 3
            degrees = int(coordinate[:degree_len])
            minutes = float(coordinate[degree_len:]) / 60
            decimal = degrees + minutes
            if direction in ['S', 'W']:
                decimal *= -1
            return round(decimal, 6)
        return None

    @property
    def location(self):
        data = self.read_line()
        if data:
            print(self.parse_gngga(data))
            return self.parse_gngga(data)
        return None, None, None
```

imu.py:

```
import board
import busio
import adafruit_mpu6050
import math

class IMU:
    def __init__(self):
        self.i2c = busio.I2C(board.GP1, board.GP0)
```

```
    def get_angle(self):
        self.mpu = adafruit_mpu6050.MPU6050(self.i2c)
        return math.atan2(self.mpu.acceleration[2], self.mpu.acceleration[1]) *
(180/math.pi)
```

temp_sensor.py:

```
import machine

class Temp:
    def __init__(self):
        self.sensor_temp = machine.ADC(4)
        self.conversion_factor = 3.3 / (65535)

    def get_raw_value(self):
        return self.sensor_temp.read_u16()

    def get_temperature(self):
        reading =  self.get_raw_value() * self.conversion_factor
        return (27 - (reading - 0.706)/0.001721)
```

index.css:

```
:root {
  --color-white: #fff;
  --color-dark-variant: #f3f5f8;
  /* --border-radius-1: 0.4rem; */
}

* {
  margin: 0;
  padding: 0;
  outline: 0;
}

/* body {
  width: 100vw;
  height: 100vh;
  overflow-x: hidden;
  background: var(--color-dark-variant);
} */
h1 {
  margin-top: 0.4rem;
  margin-left: 1.6rem;
}
```

```css
/* .container {
  display: grid;
  width: 96%;
  margin: 0 auto;
  gap: 1.8rem;
  grid-template-columns: 14rem auto;
} */

.container {
  position: fixed;
  top: 0;
  left: 0;
  width: 400px;
  height: 350px;
  box-sizing: border-box;
}
.container2 {
  position: fixed;
  bottom: 0;
  left: 0;
  width: 400px;
  height: 350px;
  box-sizing: border-box;
}


.values {
  text-align: center;
  resize: none;
  height: 100%;
  font-size: 1.5rem;
  font-weight: bold;
}

main {
  /* margin-top: 1.4rem; */
  background: var(--color-white);
  /* padding: 0.4rem; */
  /* border: 1px solid red; */
  text-align: center;
}

.wrapper {
  /* display: flex; */
  align-items: center;
  justify-content: center;
}
```

index.js:

```
// const sensorValues = document.querySelector("#sensor-values");

const sensorData = [];

/*
  Plotly.js graph and chart setup code
*/
var sensorChartDiv = document.getElementById("sensor-chart");
var sensorChartDivTemp = document.getElementById("sensor-temp-chart");

// History Data
var sensorTrace = {
  x: [],
  y: [],
  name: "LDR/Photoresistor",
  mode: "lines+markers",
  type: "line",
};
// History Data
var sensorTraceTemp = {
  x: [],
  y: [],
  name: "tmp",
  mode: "lines+markers",
  type: "line",
};

var sensorLayout = {
  autosize: false,
  width: 400,
  height: 300,
  colorway: ["#05AD86"],
  margin: { t: 40, b: 40, l: 80, r: 80, pad: 0 },
  xaxis: {
    gridwidth: "2",
    autorange: true,
  },
  yaxis: {
    gridwidth: "2",
    autorange: true,
  },
};


var config = { responsive: true };
```

```
Plotly.newPlot(sensorChartDiv, [sensorTrace], sensorLayout, config);
Plotly.newPlot(sensorChartDivTemp, [sensorTraceTemp], sensorLayout, config);

// Will hold the sensor reads
let newSensorXArray = [];
let newSensorYArray = [];

let newSensorXArray_temp = [];
let newSensorYArray_temp = [];

// The maximum number of data points displayed on our scatter/line graph
let MAX_GRAPH_POINTS = 50;
let ctr = 0;

function updateChart(sensorRead) {
  if (newSensorXArray.length >= MAX_GRAPH_POINTS) {
    newSensorXArray.shift();
  }
  if (newSensorYArray.length >= MAX_GRAPH_POINTS) {
    newSensorYArray.shift();
  }
  newSensorXArray.push(ctr++);
  newSensorYArray.push(sensorRead);

  var data_update = {
    x: [newSensorXArray],
    y: [newSensorYArray],
  };

  Plotly.update(sensorChartDiv, data_update);
}


function updateChartTemp(sensorRead) {
  if (newSensorXArray_temp.length >= MAX_GRAPH_POINTS) {
    newSensorXArray_temp.shift();
  }
  if (newSensorYArray_temp.length >= MAX_GRAPH_POINTS) {
    newSensorYArray_temp.shift();
  }
  newSensorXArray_temp.push(ctr++);
  newSensorYArray_temp.push(sensorRead);

  var data_update = {
    x: [newSensorXArray_temp],
    y: [newSensorYArray_temp],
  };
```

```
  Plotly.update(sensorChartDivTemp, data_update);
}

// WebSocket support
var targetUrl = `ws://${location.host}/ws`;
var websocket;
window.addEventListener("load", onLoad);

function onLoad() {
  initializeSocket();
}

function initializeSocket() {
  console.log("Opening WebSocket connection MicroPython Server...");
  websocket = new WebSocket(targetUrl);
  websocket.onopen = onOpen;
  websocket.onclose = onClose;
  websocket.onmessage = onMessage;
}
function onOpen(event) {
  console.log("Starting connection to WebSocket server..");
}
function onClose(event) {
  console.log("Closing connection to server..");
  setTimeout(initializeSocket, 2000);
}
function onMessage(event) {
  console.log("WebSocket message received:", event);
  var jsonData = JSON.parse(event.data);
  updateChart(jsonData.angle);
  updateChartTemp(jsonData.temp);
  updateMarker(jsonData.latitude, jsonData.longitude);
}

function updateMarker(lat, lon) {
  var newLatLng = new L.LatLng(lat, lon);
  marker.setLatLng(newLatLng); // Update marker position

  // Calculate the distance from the new marker position to the circle center
  var distance = map.distance(newLatLng, circle.getLatLng());

  // Update marker color based on position relative to the circle
  if (distance <= circle.getRadius()) {
    // Marker is within the circle
    marker.setIcon(new L.Icon({
      iconUrl: 'https://i.imgur.com/z1KaKUV.png',
      iconSize: [25, 41],
      iconAnchor: [12, 41]
```

```
    }));
  } else {
    // Marker is outside the circle
    marker.setIcon(new L.Icon({
      iconUrl: 'https://i.imgur.com/YaYFBZ6.png',
      iconSize: [25, 41],
      iconAnchor: [12, 41]
    }));
    sendCommandToPicoW("turnOnLED");  // Command to turn on the LED
  }
}

//function sendMessage(message) {
  websocket.send(message);
//}

function sendCommandToPicoW(command) {
  if (websocket.readyState === WebSocket.OPEN) {
    websocket.send(JSON.stringify({ command: command }));
    console.log("Command sent to Pico W:", command);
  } else {
    console.log("WebSocket is not open. Cannot send command.");
  }
}
```

map.js:

```
const map = L.map("map").setView([42.444, -76.482], 13);
L.tileLayer("https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png", {
  attribution: "&copy; <a
href='https://www.openstreetmap.org/copyright'>OpenStreetMap</a> contributors"
}).addTo(map);

const marker = L.marker([42.444, -76.482]).addTo(map);

const circle = L.circle([42.444, -76.482], {
  color: 'red',
  opacity: 0.3,
  fillColor: '#f03',
  fillOpacity: 0.3,
  radius: 500
}).addTo(map);

document.getElementById('form').addEventListener('submit', function(event) {
  event.preventDefault();
  const latitude = parseFloat(document.getElementById('latitude').value);
  const longitude = parseFloat(document.getElementById('longitude').value);
```

```
  const radius = parseFloat(document.getElementById('radius').value);

  if (!isNaN(latitude) && !isNaN(longitude) && latitude >= -90 && latitude <= 90 &&
longitude >= -180 && longitude <= 180 && radius > 0) {
    map.setView([latitude, longitude], 13); // Adjust zoom level as needed
    circle.setRadius(radius);
    circle.setLatLng([latitude, longitude]);
  } else {
    alert('Please enter valid latitude, longitude, and radius values.');
  }
});
```

index.html:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <link rel="stylesheet" href="https://unpkg.com/leaflet/dist/leaflet.css" />
    <script src="https://unpkg.com/leaflet/dist/leaflet.js"></script>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>MicroPython WebSocket</title>
    <link rel="stylesheet" href="static/index.css" />
    <link rel="icon" href="./favicon.ico" type="image/x-icon" />
    <script src="https://cdn.plot.ly/plotly-2.16.1.min.js"></script>
  </head>
  <body>
    <div class="container">
      <main class="chart" style="max-width: 100%; max-height: 100%;">
        <h2>Angle</h2>
        <div class="wrapper">
          <div id="sensor-chart" class="sensor-chart"></div>
        </div>
      </main>
    </div>

    <div class="container2">
      <main class="chart" style="max-width: 100%; max-height: 100%;">
        <h2>Temp</h2>
        <div class="wrapper">
          <div id="sensor-temp-chart" class="sensor-temp-chart"></div>
        </div>
      </main>
    </div>
```

```
    <div id="map" style="width: 700px;position: absolute;bottom: 0;right: 0;height:
100%;"></div>
    <div id="form-container" style="position: absolute; left: 649px; background-color:
#f9f9f9; padding: 10px; border: 1px solid #ddd; border-radius: 5px;">
      <form id="form" style="display: block;">
        <label for="latitude">Latitude:</label>
        <input type="text" id="latitude" name="latitude" style="display: block;
margin-bottom: 5px;">
        <label for="longitude">Longitude:</label>
        <input type="text" id="longitude" name="longitude" style="display: block;
margin-bottom: 5px;">
        <label for="radius">radius:</label>
        <input type="text" id="radius" name="radius" style="display: block; margin-
bottom: 5px;">
        <button type="submit">Update Center</button>
      </form>
    </div>
    <script src="static/map.js"></script>

    <script src="static/index.js"></script>
  </body>
</html>
```

Please refer to the full-code package for libraries we use in this project.