# Designing a System for Wearable Gesture Recognition

## IoT Project

Sumukh Prasad

1BG25CS163

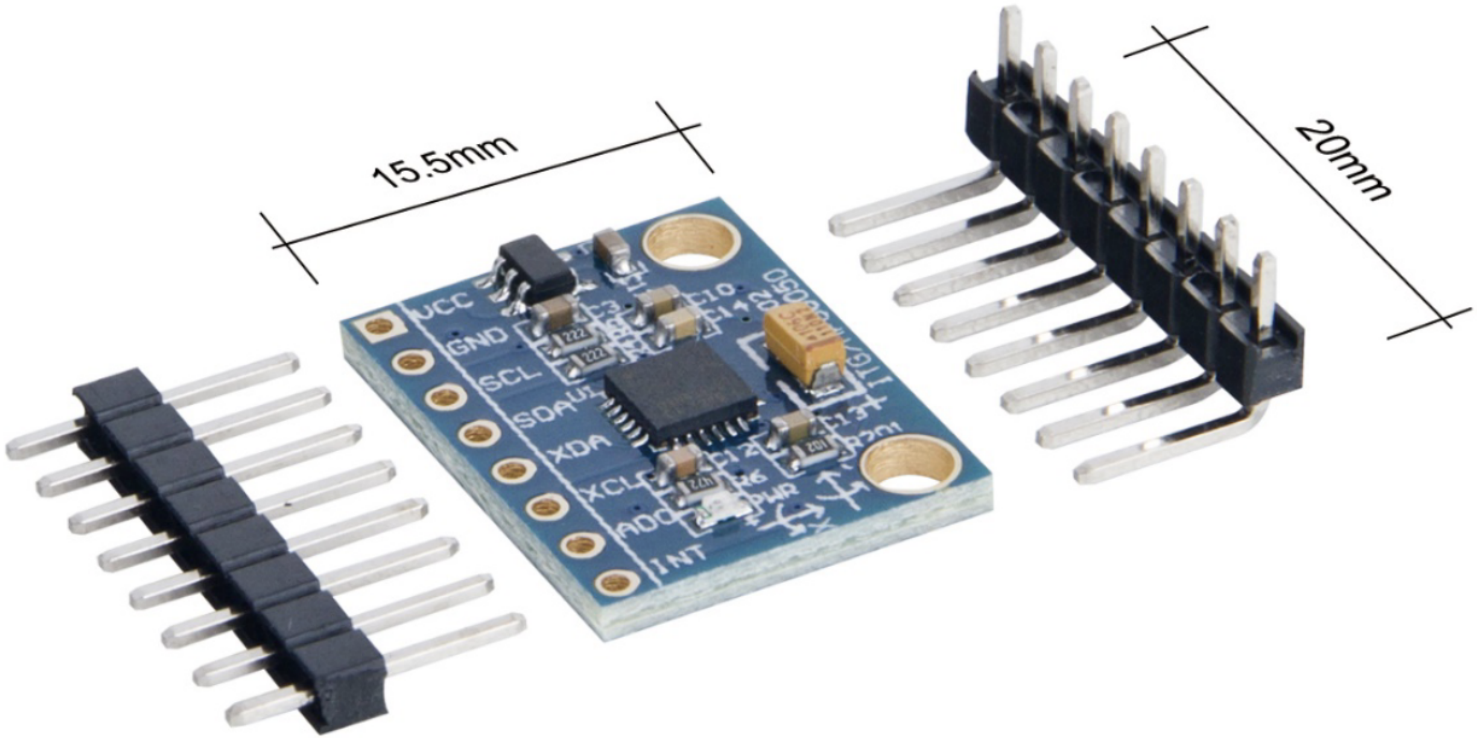
25cse156 (at) bnmit (dot) in

16-06-2026

# Table of Contents

# Table of Contents

# MPU-6050 Accel/Gyro


15.5mm
20mm
VCC
GND
SCL
SDA
XDA
XCL
ADO
INT

# What is an IMU?

## IMU

**Inertial Measurement Unit**s, or IMUs, are electronic devices that continuously track motion-related quantities like acceleration and angular velocity.

Or in simple terms, an IMU is a device that measures

- linear acceleration and gravity, using an **accelerometer**
- angular velocity, using a **gyroscope**

# What does the MPU-6050 measure?

| **Specification** | **Value** |
|---|---|
| Sampling rate | **8kHz** |
| Accelerometer | Triple-axis MEMS accelerometer, +/- **2,48,16g** |
| Gyroscope | Digital-output X-, Y-, and Z-Axis angular rate sensors, +/-**250,500,1000,2000°/sec** |
| Communication | 400kHz $I^2C$ interface |
| Additional | Integrated 16-bit ADCs, 1MHz SPI serial interface, Digital-output temperature sensor, etc. |

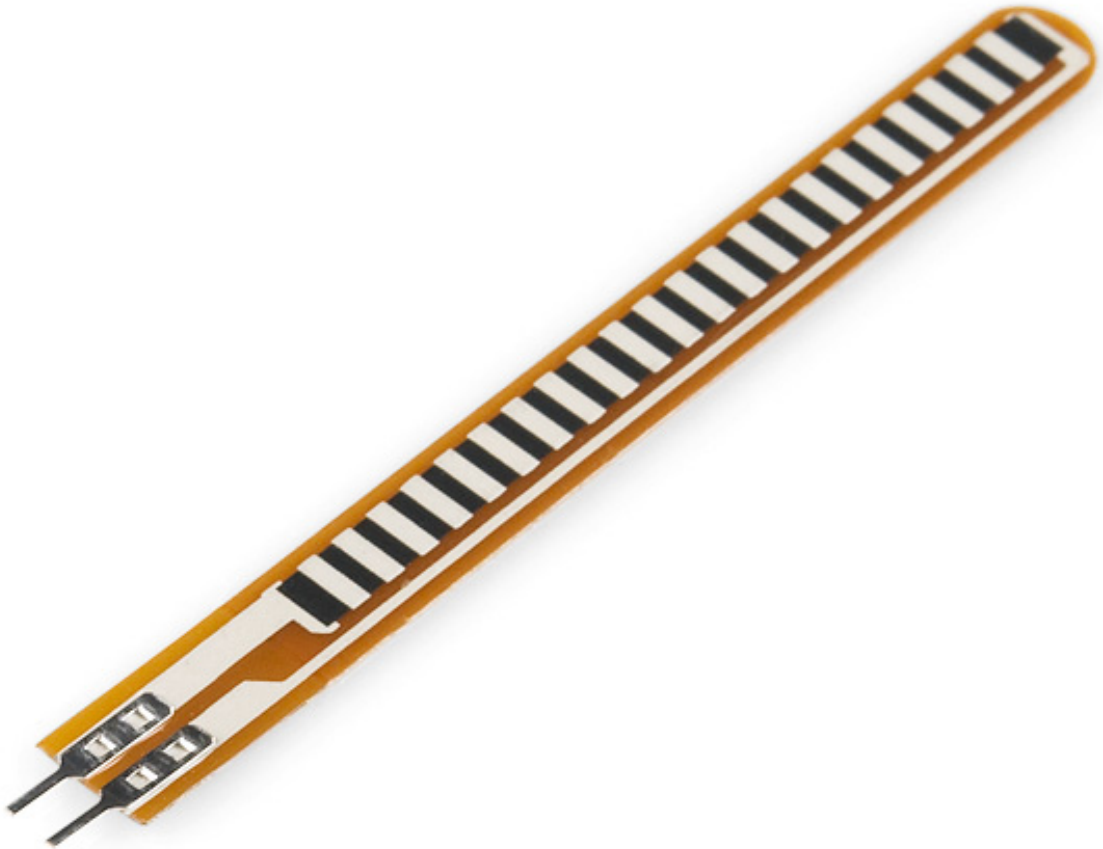
*However, this is not without limitations...*

# Limitations of an IMU

- **Accelerometer** provides absolute reference, but is noisy
- **Gyroscope** is smooth, but drifts over time

*How we solve this is left to be seen later.*

# Flex Sensors



# How does a flex sensor work?

- Essentially a variable resistor.
- Almost always requires a voltage divider circuit.
- Velostat/Linqstat polymer with carbon black, which makes the surface piezoresistive.
- *Piezoresistive: resistance varies with flexing or pressure.*

# Table of Contents

# Why esp-idf?

We would like to execute some tasks in parallel. For example:

- Task 1: Read IMU
- Task 2: Read flex sensors
- Task 3: Process data
- Task 4: Send over UART

These tasks cannot be executed simultaneously when using a single blocking event loop like the Arduino framework provides.

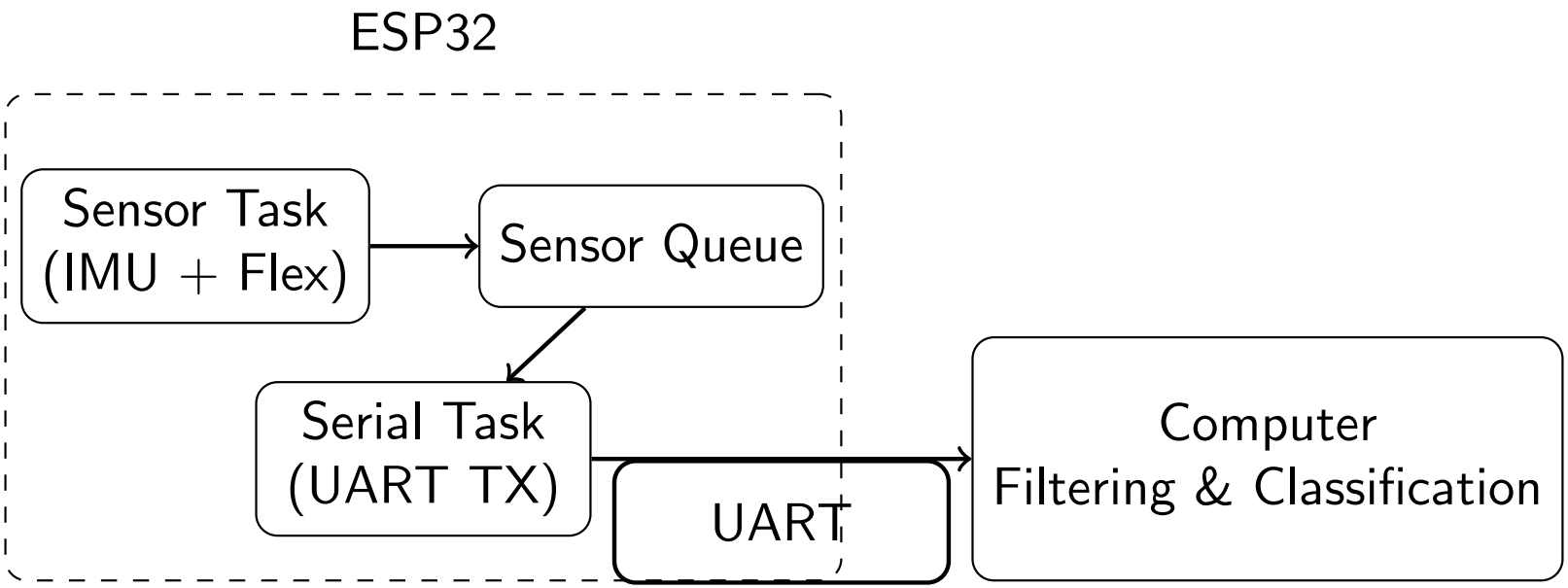
# What is an RTOS?

An **RTOS** or Real Time Operating System is an operating system that allows for...

- multiple tasks to run concurrently
- data to be shared between tasks using queues

# Basic System Architecture

# System Architecture



# Table of Contents

## Signal Processing

**Signal Processing** is a field of electronics that focuses on analyzing, modifying and synthesizing signals.

Let's see how we do that here.

# Calibration and Normalization

# Calibration

Raw sensor values are often biased.

**How do we solve this?**

We can estimate a baseline offset by averaging readings at rest:

$$Z_k = \frac{1}{n} \sum_{i=1}^{n} v(i)$$

This gives us a reference value for each sensor.

# Calibration code

```
sensor_data_t data;
int32_t gyro_sum[3] = {0};
int32_t accel_sum[3] = {0};

int samples = 1000;

for (int i = 0; i < samples; i++) {
  read_raw(&data);

  for (int j = 0; j < 3; j++) {
    gyro_sum[j] += data.gyro[j];
    accel_sum[j] += data.accel[j];
  }
  vTaskDelay(pdMS_TO_TICKS(5));
}

for (int j = 0; j < 3; j++) {
  gyro_offset[j] = gyro_sum[j] / samples;
  accel_offset[j] = accel_sum[j] / samples;
}
```

# Normalization

Subsequently, we can subtract the offset to center the data:

$$V_k^{\text{adj}} = V_k^{\text{raw}} - Z_k$$

Additionally, roll and pitch values are calculated from acceleration:

$$\text{roll} = \arctan\left(\frac{a_y}{a_z}\right)$$

$$\text{pitch} = \arctan\left(\frac{-a_x}{\sqrt{a_y^2 + a_z^2}}\right)$$

# Complementary Filtering and Sensor Fusion

# Why use a complementary filter?

We learnt something earlier about our IMU:

- **Accelerometer** provides absolute reference, but is noisy
- **Gyroscope** is smooth, but drifts over time

How do we fix this?

# Why use a complementary filter?

Notice that these two values are **complementary**:

| Sensor | Accuracy | Smoothness |
|---|---|---|
| **Accelerometer** | High accuracy | Very noisy |
| **Gyroscope** | Drifts | Smooth |

A **complementary filter** uses two complementary data sources to produce a single output value, often called *sensor fusion*.

# Mathematical model for Complementary Filtering

We combine both signals:

$$\theta = \alpha(\theta + \omega \cdot dt) + (1 - \alpha)(\theta_{\text{accel}})$$

- Gyroscope handles short-term motion
- Accelerometer corrects long-term drift

# Complementary Filter code

```
class ComplementaryFilter:
  def __init__(self, alpha=0.9):
    self.alpha = alpha
    self.roll = 0.0
    self.pitch = 0.0

  def update(self, ax, ay, az, gx, gy, gz, dt):
    # accel angles
    accel_roll, accel_pitch = accel_to_angles(ax, ay, az)

    # integrate gyro
    self.roll += gx * dt
    self.pitch += gy * dt

    # fuse
    self.roll = self.alpha * self.roll + (1 - self.alpha) *
        accel_roll
    self.pitch = self.alpha * self.pitch + (1 - self.alpha)
        * accel_pitch

    return self.roll, self.pitch
```

# Table of Contents

# Why Classification is Hard

# Gestures Are Not Identical

- The same gesture can be performed at different speeds
- Timing between movements varies
- Sensor readings are noisy and inconsistent

**But why do we care?**

Two recordings of the **exact same gesture** may look very different as signals. How do we solve this?

# Why not use thresholds?

# Why not use thresholds?

- Thresholds assume fixed timing
- Small variations cause large errors
- Difficult to generalize across users

**Think about this:**
Assume you write the same sentence twice, once very fast, and once very slow.
Do they look the same?

*A gesture performed quickly may never cross the same thresholds as a slower one.*

# Dynamic Time Warping

# Dynamic Time Warping

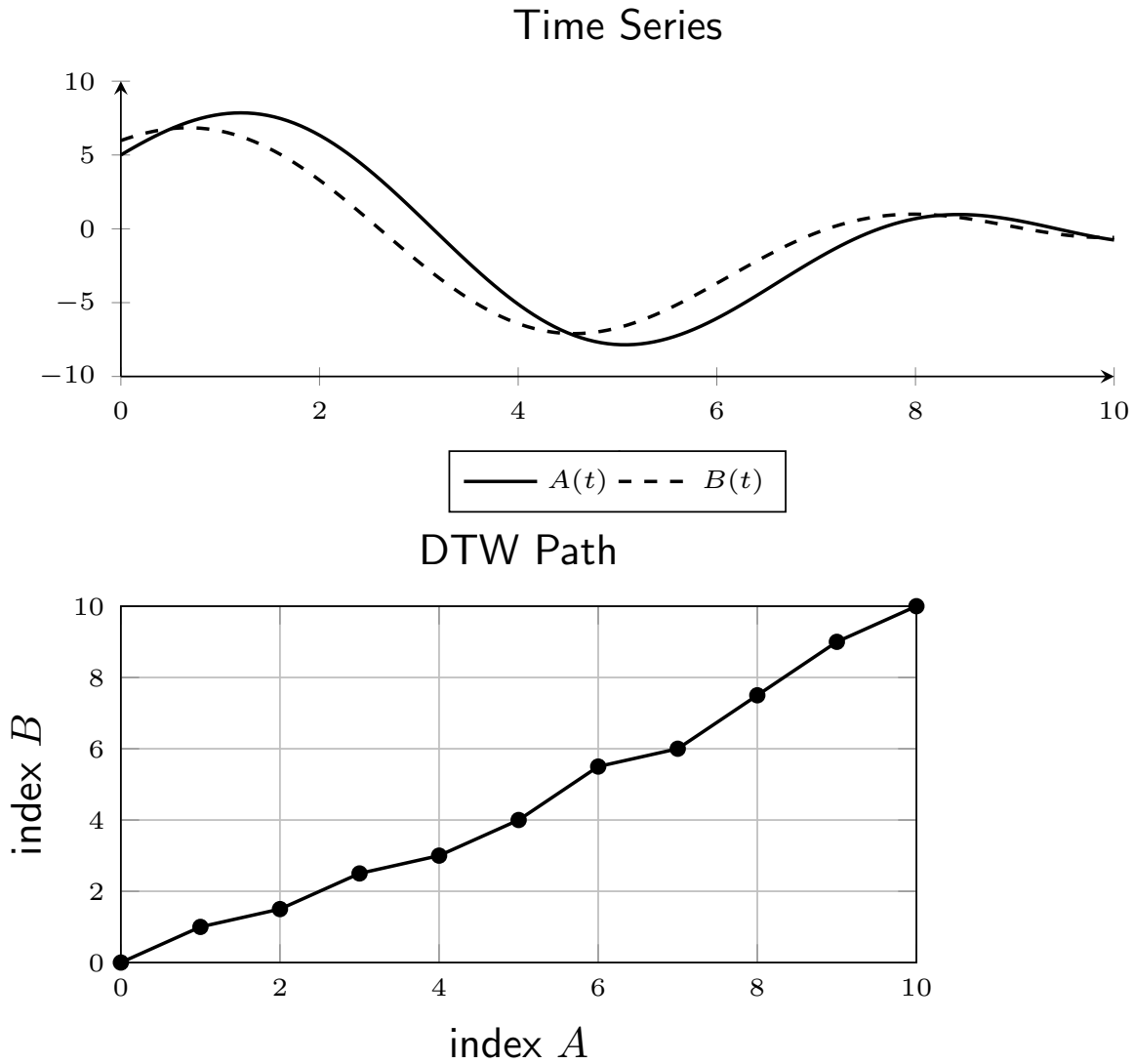
Let's think about a method to classify gestures.

Assuming that gestures can be plotted against time, can we...

- compare two time-series signals,
- stretch and compress along the time axis to find the best fit, and finally,
- find a distance score between them?

This is called **Dynamic Time Warping**.

# DTW Illustration



*Demo*

# Questions and Discussion