Getting Started with Sensor Data Analysis in Rust

Emil Lindfors | January 15, 2024

Modern aquaculture facilities rely heavily on continuous environmental monitoring. In this post, we'll explore how to build robust sensor data processing systems using Rust.

Why Rust for Sensor Data?

Rust offers several advantages for sensor data processing:

1. Memory safety without garbage collection
2. Zero-cost abstractions for high performance
3. Strong type system that catches errors at compile time

According to the official Rust documentation {{ reference(key="rust2024", num=1) }}, these features make it ideal for embedded and IoT applications.

Basic Data Structures

Let's define a simple structure for sensor readings:

```
use chrono::{DateTime, Utc};

#[derive(Debug, Clone)]
pub struct SensorReading {
    pub timestamp: DateTime<Utc>,
    pub sensor_id: String,
    pub value: f64,
    pub unit: MeasurementUnit,
}

#[derive(Debug, Clone, Copy)]
pub enum MeasurementUnit {
    Celsius,
    Ph,
    Ppm,
    Percent,
}
```

Mathematical Foundations

When processing sensor data, we often need to apply statistical methods. The moving average is commonly used for smoothing noisy sensor readings:

$$\bar{x}_n = \frac{1}{n}\sum_{i=1}^{n} x_i$$

For real-time applications, we can use an exponential moving average where α is the smoothing factor:

$$S_t = \alpha \cdot x_t + (1 - \alpha) \cdot S_{t-1}$$

Research in IoT sensor networks {{ reference(key="iot2023", num=2) }} has shown that proper filtering is essential for reliable data.

Implementing a Sensor Buffer

Here's a practical implementation of a ring buffer for sensor data:

```
pub struct SensorBuffer<const N: usize> {
    data: [Option<SensorReading>; N],
    write_index: usize,
    count: usize,
}

impl<const N: usize> SensorBuffer<N> {
    pub fn new() -> Self {
        Self {
            data: [const { None }; N],
            write_index: 0,
            count: 0,
        }
    }

    pub fn push(&mut self, reading: SensorReading) {
        self.data[self.write_index] = Some(reading);
        self.write_index = (self.write_index + 1) % N;
        if self.count < N {
            self.count += 1;
        }
    }

    pub fn average(&self) -> Option<f64> {
        if self.count == 0 {
            return None;
        }
        let sum: f64 = self.data.iter()
            .filter_map(|r| r.as_ref().map(|r| r.value))
```

```
            .sum();
        Some(sum / self.count as f64)
    }
}
```

Environmental Considerations

As described by Bohm {{ reference(key="bohm2021", num=3) }}, environmental monitoring systems must account for:

- Sensor drift over time
- Calibration requirements
- Environmental interference
- Data validation and outlier detection

Conclusion

Rust provides an excellent foundation for building reliable sensor data processing systems. The combination of performance and safety makes it particularly suitable for aquaculture monitoring applications where data integrity is crucial.

In future posts, we'll explore more advanced topics including:

- Async I/O for multiple sensors
- Integration with time-series databases
- Real-time alerting systems