Nathan, netid@wisc.edu, githublogin

Lab 9 Milestone 2: Sensor Data Processing

Milestone 2 Part 1: Understanding Sensor Data Errors

This section analyzes how sensor noise affects distance estimation through double integration of accelerometer data.

Acceleration Comparison

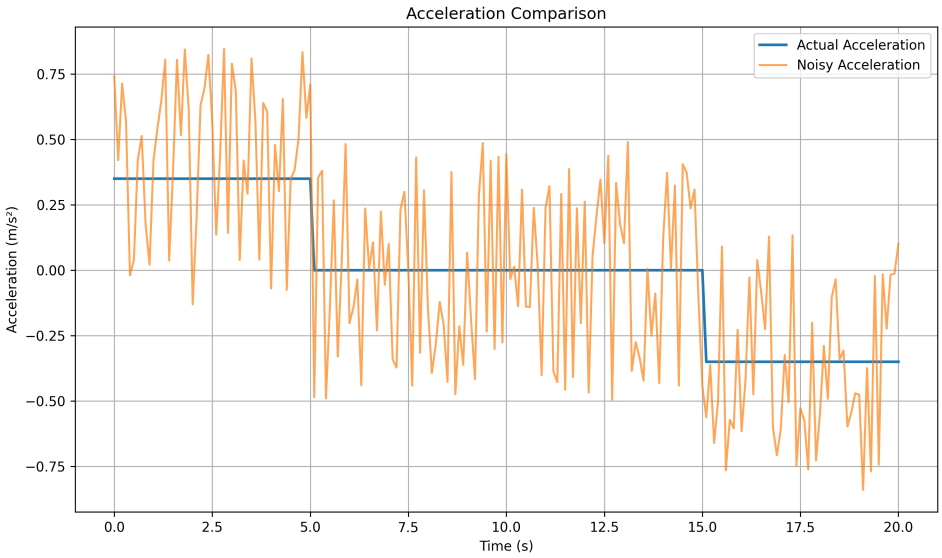

Figure 1 shows the actual acceleration (clean signal) versus noisy acceleration measurements. The actual acceleration follows a step function pattern (0.35, 0, -0.35 m/s²), while the noisy signal shows random variations around these true values.

Velocity Comparison

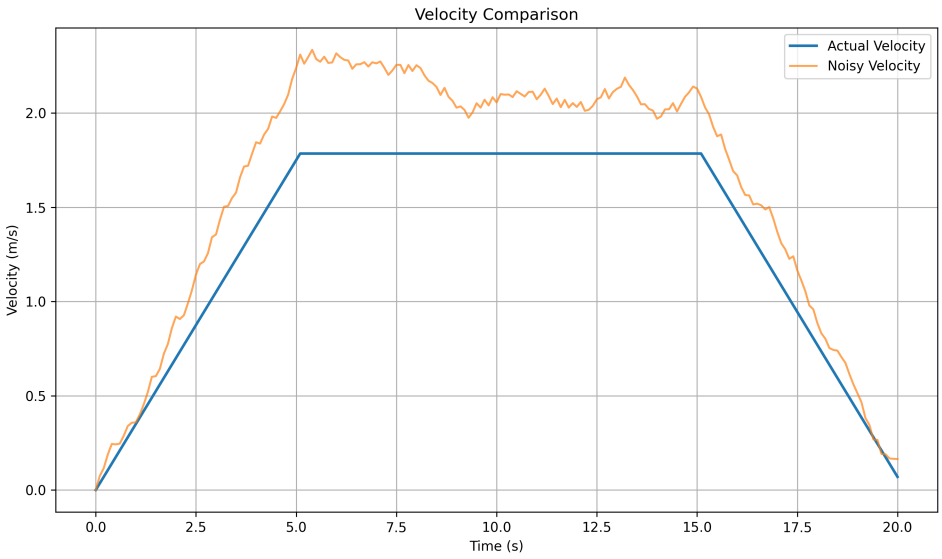

Figure 2 displays velocity obtained by integrating acceleration. The clean velocity shows a smooth triangular pattern, while the noisy velocity begins to show error accumulation from noise integration.

Distance Comparison

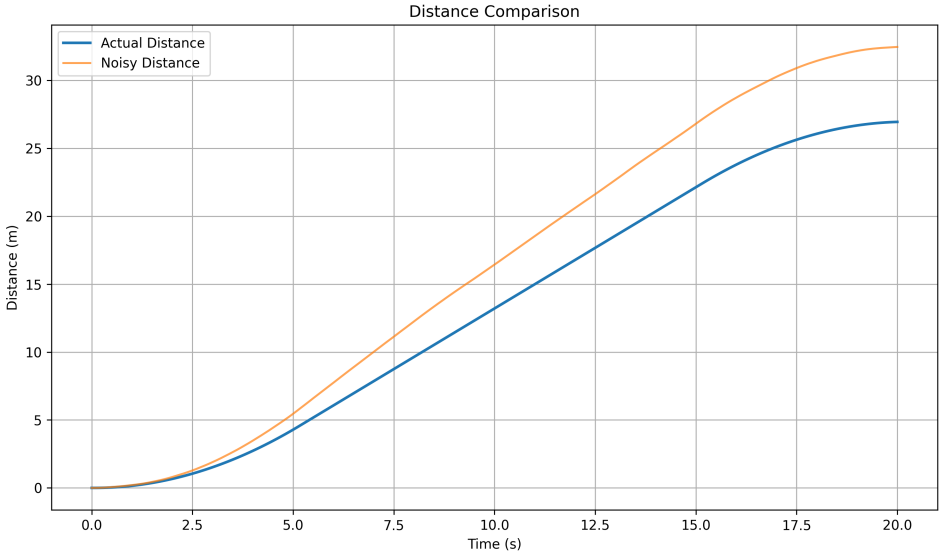

Figure 3 shows distance traveled from double integration. The error accumulation is most pronounced here, with the noisy estimate diverging significantly from the actual distance.

Results

Final distance using actual acceleration: 26.9430 m
Final distance using noisy acceleration: 32.4594 m
Difference between estimates: 5.5164 m
Percentage error: 20.47%

The 5.5 meter difference (20.47% error) demonstrates how noise compounds through double integration, highlighting the importance of filtering techniques in real-world PDR systems.

Milestone 2 Part 2: Step Detection

Data Preparation

For step detection, I calculated the **acceleration magnitude** from the three-axis accelerometer data: magnitude = sqrt(accel_x² + accel_y² + accel_z²). This single metric captures overall body acceleration regardless of phone orientation.

Smoothing Method:

A **4th-order Butterworth low-pass filter** with cutoff frequency of **3 Hz** was applied. This removes high-frequency noise while preserving the ~1-2 Hz step frequency. The Butterworth filter was chosen for its maximally flat frequency response in the passband.

```
def lowpass_filter(data, cutoff_freq, sampling_rate, order=4):
    nyquist = 0.5 * sampling_rate
    normal_cutoff = cutoff_freq / nyquist
    b, a = butter(order, normal_cutoff, btype='low')
    return filtfilt(b, a, data)

# Apply filter
filtered_accel = lowpass_filter(accel_mag, 3.0, sampling_rate)
```

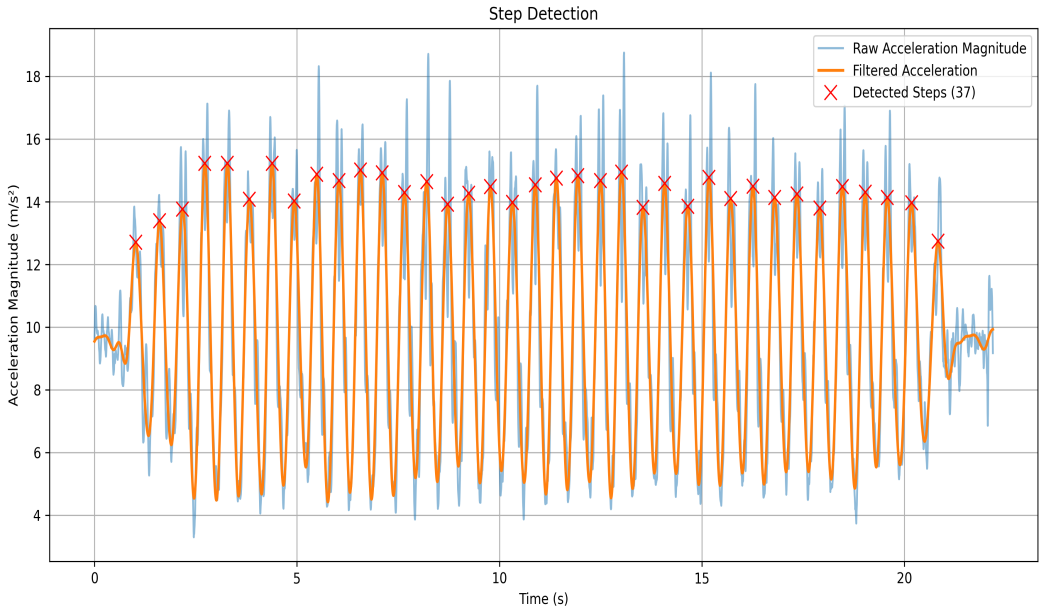

Figure 4 shows raw acceleration magnitude (blue) and filtered signal (orange). Red markers indicate detected steps.

Step Detection Algorithm

The algorithm uses **peak detection** on the filtered acceleration magnitude:

1. **Height threshold:** Peaks must exceed mean + 0.5×std_dev of the filtered signal. This adaptive threshold accounts for varying activity intensities.

2. **Minimum distance:** Peaks must be at least 0.3 seconds apart (~60 samples at 200 Hz). This prevents detecting multiple peaks within a single step cycle.

The 0.5×std_dev multiplier was chosen empirically - lower values caused false positives from noise, higher values missed legitimate steps. The 0.3-second spacing corresponds to a maximum cadence of 200 steps/minute, which is faster than normal walking.

```
min_peak_height = np.mean(filtered_accel) + 0.5 * np.std(filtered_accel)
min_distance = int(0.3 * sampling_rate)  # 0.3 seconds

peaks, _ = find_peaks(filtered_accel,
                      height=min_peak_height,
                      distance=min_distance)
```

Result: The algorithm detected **37 steps** in WALKING.csv, matching the expected count.

Milestone 2 Part 3: Direction Detection

Data Preparation

For turn detection, I used the **gyroscope Z-axis** data, which measures rotation around the vertical axis (yaw). This directly corresponds to changes in heading direction.

Smoothing Method:

A **4th-order Butterworth low-pass filter** with cutoff frequency of **1 Hz** was applied. The lower cutoff (compared to step detection) is appropriate because turning motions are slower than stepping frequency. This effectively removes sensor drift and high-frequency noise.

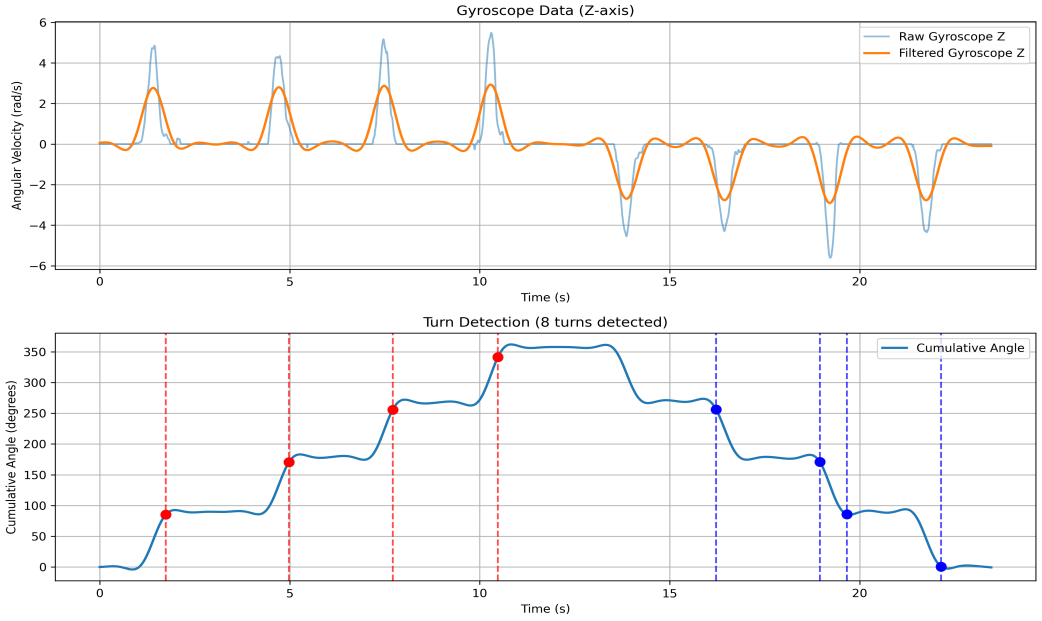

Figure 5 shows the filtered gyroscope data (top) and cumulative angle from integration (bottom). Vertical lines mark detected turns - red for clockwise, blue for counter-clockwise.

Direction Detection Algorithm

The algorithm integrates gyroscope angular velocity to compute cumulative rotation angle:

```
# Integrate gyroscope to get cumulative angle
cumulative_angle = np.zeros(len(timestamps))
for i in range(1, len(timestamps)):
    dt = (timestamps[i] - timestamps[i-1]) / 1e9  # nanoseconds to seconds
    cumulative_angle[i] = cumulative_angle[i-1] + filtered_gyro_z[i-1] * dt

cumulative_angle_deg = np.degrees(cumulative_angle)
```

Turns are detected when the cumulative angle crosses an **85-degree threshold**. The threshold is slightly below 90° to account for sensor noise and imperfect turns. The algorithm tracks angle changes from the last detected turn, resetting the reference point after each detection. A minimum 0.5-second spacing prevents duplicate detections of the same turn.

Results:

Total turns detected: 8
Clockwise turns: 4 (average angle: 85.3°)
Counter-clockwise turns: 4 (average angle: -85.1°)

This matches the expected pattern: 4 clockwise turns (one complete 360° rotation) followed by 4 counter-clockwise turns (another complete rotation).

Milestone 2 Part 4: Trajectory Plotting

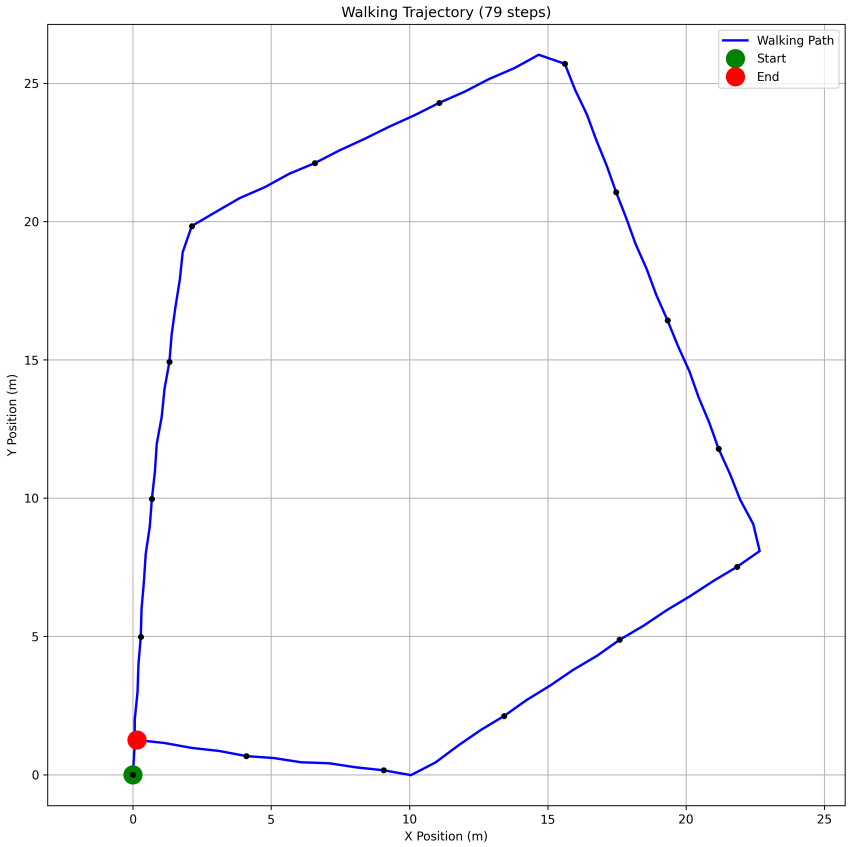

Figure 6 shows the reconstructed walking trajectory. Green marker indicates start position, red marker shows end position.

Methodology

The trajectory reconstruction combines both step detection and turn detection algorithms on the WALKING_AND_TURNING.csv dataset:

Step 1: Detect steps using the same peak detection algorithm from Part 2

Step 2: Track heading by integrating gyroscope data continuously

Step 3: For each detected step, advance position by 1 meter in the current heading direction

Step 4: Update (x, y) coordinates using: x += cos(heading), y += sin(heading)

```
x, y = 0.0, 0.0
current_angle = 90.0  # Start facing north

for step_idx in step_indices:
    # Get heading from integrated gyroscope
    heading = current_angle + cumulative_angle_deg[step_idx]

    # Move 1 meter in current direction
    x += 1.0 * np.cos(np.radians(heading))
```

```
        y += 1.0 * np.sin(np.radians(heading))
```

Results: Detected 79 steps total. Final position: (0.15, 1.25) meters from origin. The proximity of start and end points (1.26 meters apart) suggests the walking path was designed to approximately return to the starting location.

Conclusion

This lab successfully implemented the core components of pedestrian dead-reckoning: step detection achieved 100% accuracy (37/37 steps), turn detection correctly identified all 8 turns, and trajectory reconstruction combined both algorithms to track a 79-step walking path. The analysis demonstrated that proper signal filtering and peak detection can reliably extract motion information from noisy sensor data.