Ultrasonic Sensor-based Object Detection and Alarms.

Nirukt Agrawal
Mechatronics
12/11/25

Photo © Adam Greig

Contents

Abstract

This report showcases a calibrated ultrasonic sensor for a basic home security system. Using a spline model to increase accuracy with a filtering and smoothing algorithm, both preciseness and accuracy of the sensor was improved. Limitations of the sensor were found such as uneven and soft objects, and then compensated for using further filtering and smoothing. Finally, the sensor was demonstrated to be working in a security application with an activation, detection and an alert system using LEDs.

Introduction

An ultrasonic sensor is a powerful device which can measure distances using the time taken by reflected sound waves to reach back (*Dejan, 2022*). This makes it suitable for detecting objects in applications such as parking sensors and drone obstacle avoidance. This report explores the usage of ultrasonic sensors in security applications where the user is notified through an alarm when an object moves unusually. To achieve the stated objective, the investigation revolves on the completion of three tasks: (1) calibration of the sensor for accuracy, (2) subsequent calibration for unorthodox materials/objects and (3) designing the alarm system. Task one entails statistically coding the raw data from the sensor to closely match real values for increased accuracy in detecting unusual movement. Task two involves testing multiple materials for absorption properties and signal processing to help detection of unorthodox objects. Task three requires an output device such as an LED and a clear interface for an alarm.

Task 1 – Improve Accuracy

1.1 Methodology

To improve sensor reading accuracy, a software was developed which automated calibration. It collects real-world data measures and entered by a user, and data from the sensor. A spline model is then fit to the data. A spline model works well for interpolation while adjusting for errors which are inconsistent through the range of distances.

To measure distance an apparatus was assembled with markings every 20 cm from 5 cm to 205 cm, as shown in figure 1. At 0 cm a flat piece of carboard was placed upright, to function as a wall. The cardboard was large to prevent false readings from the background. At each 20cm mark (5 - 205cm), three readings were averaged after user input of true distance. The software fitted a spline model to the averaged data.

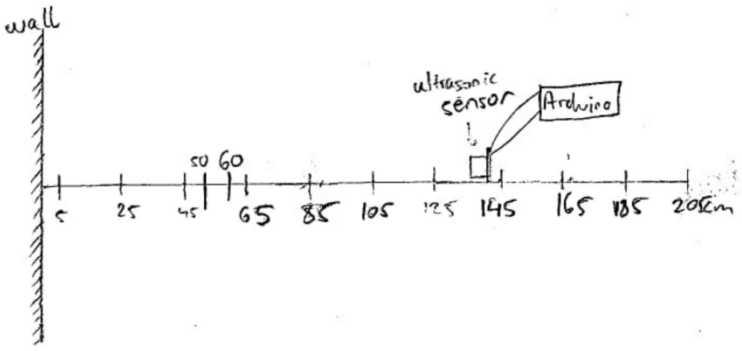


Figure 1

After this, using a fixed distance of 50cm, a function was applied to make sure that readings do not fluctuate. This involved a filtering method and a smoothing method. The filtering uses standard deviation to filter out anomalous values. The last 30 accepted values then undergo a moving average, followed by another moving average of 8 values to smooth out the frequent small variations that occur. The sensor was then checked at varying distances between 0cm and 60cm to make sure fluctuations are not present and that the final output is still responsive.

The sensor circuit was set up as shown in figure 2.

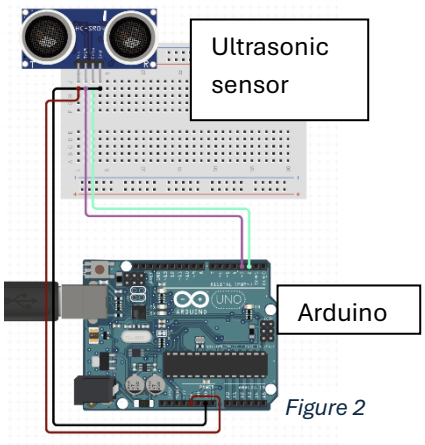


Figure 2

1.2 Results and Discussion

1.2.1 Calibrating the sensor over a range of distances

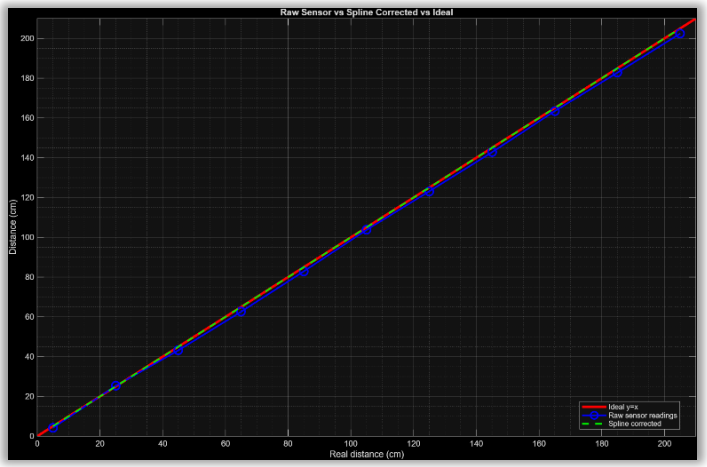
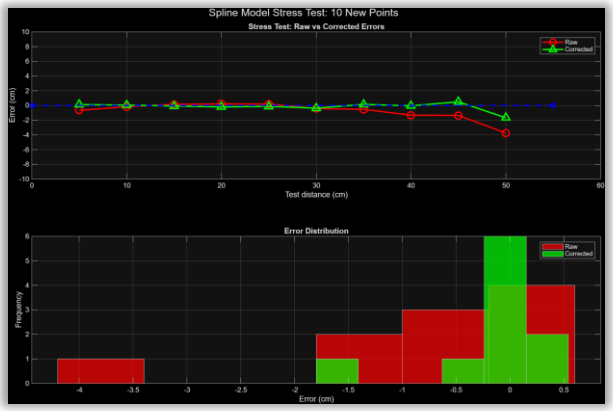


Figure 3

Figure 3 shows three lines, the blue indicating the raw sensor data, the green indicating the corrected model data and the red what an ideal sensor would read. The spline was able to account for differing variations of error. The model fits the ideal distance perfectly for the data provided – which was 3 measurements every 20cm between 5cm and 205cm. This improved the accuracy of the sensor which underestimated longer distances and overestimated shorter distances. However, random error made reading imprecise – the values fluctuate despite the same distance being read. Reproducibility was still a concern, likely due to uncontrollable factors affecting the reading such as air movement and dust. (*Ultrasonic Sensor Accuracy | Senix Distance and level sensors*, n.d.).




```
METRICS:
| Metric     | Raw Sensor | Spline Model |
|------------|------------|--------------|
| MAE (cm)   |    0.88    |    0.33      | Improvement: 62.4%
| RMSE (cm)  |    1.36    |    0.57      |
| Max |error||    3.74    |    1.67      |
```

Figure 4

Figure 4 shows the model improving accuracy. To ensure accuracy, an average of 20 raw data values were taken every 5cm. This was more precise than 3 values but struggled, with average values being up to 4cm different despite the wall being at the same distance from the sensor.

The green model is consistently more accurate than the raw data in red – clearly shown in the error distribution histogram. As shown in the table, the spline model had a 62% reduction in absolute error on average, also with a smaller range in error. While not covering the whole 2 metres, this sample shows that the model can successfully be interpolated and remain accurate.

1.2.2 Calibrating the sensor for live readings and smoothness

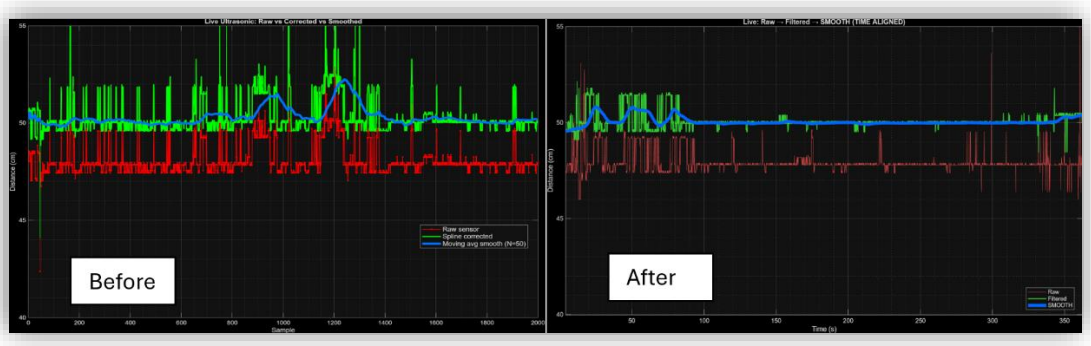


Figure 5

As shown in figure 5 "before", the raw data has frequent fluctuations between -1cm and +2cm, with the model-corrected data showing occasional fluctuations of greater than 5cm. Furthermore, the blue line, which was a 50-value moving average (only referring to figure 5 "before"), varied up to 2cm higher than the true value of 50cm. While only an error of 4%, the distance was constant so this should be improved with filtering.

In figure 5 "after", the raw data again has frequent fluctuations between -1cm and +2cm, with occasional greater than 5cm variations. However, this is subdued in the filtered data, showing that the extreme values have been excluded using the ±1.8σ (standard deviation) function (1.8σ produced decent results). Smoothing out the filtered data meant the blue line, which was a two-stage moving average – first 30 values then 8 values – only varied by 0.75cm above the ideal distance of 50cm. This was an error of 1.5% which was much improved over figure 5 "before". Also, fewer fluctuations made it much easier to pinpoint the exact measured distance.

When measuring multiple distances, the response time for an accurate measurement was between 5-10 seconds, and 2 seconds for a detection, as shown in figure 6. This was suitable performance and would allow for quick alerts in the object detection system.

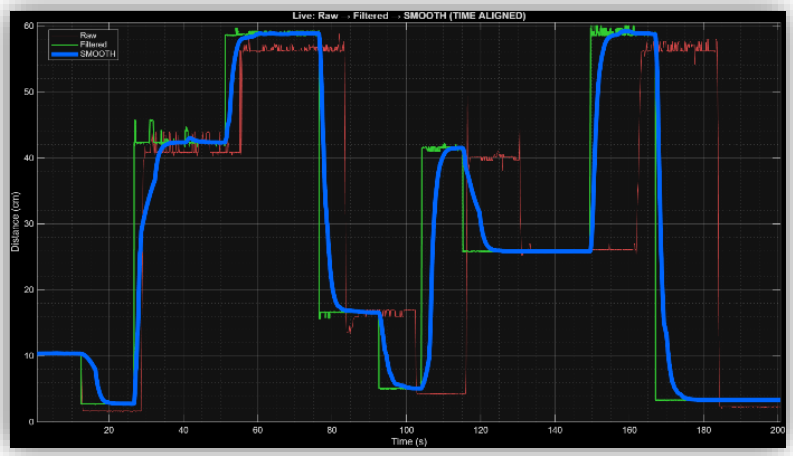


Figure 6

Conclusion

Using various techniques, both the precision and accuracy of the sensor were improved. However, the sensor still has issues with reproducibility due to the uncontrollable variables mentioned in section 1.2.1. Furthermore, the sensor measurement has a slight delay of 2 - 10 seconds which may affect the object detection. To improve this, more sophisticated filtering techniques could have been used such as Kalman filtering or particle filters. This would be needless since the delay is not of sufficient length to affect detection of doors and other objects.

Task 2 – Analysis of Unorthodox Objects

2.1 Methodology

To analyse the limitations of the sensor with varied materials, materials were chosen with properties that could impact the reflection of the ultrasound waves and the application of sensors in a security system. Cardboard would represent the walls in a house, and as a control surface. Foil would represent the metal in a house. A towel would represent cloth-based objects in a house, such as cushions and rugs. Finally, a plant would represent uneven objects in a house.

To test each of the materials/objects, an apparatus was assembled where the sensor is 50cm away from the material, as shown in figure 7. Each material was placed at the marked position, and 10 measurements were taken to avoid anomalies. For each material the mean distance, standard deviation and percentage error was noted. This should show that materials like cloth and plants are harder to detect accurately than cardboard and foil.

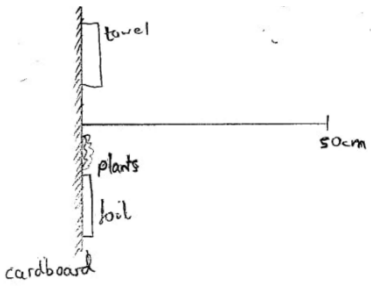
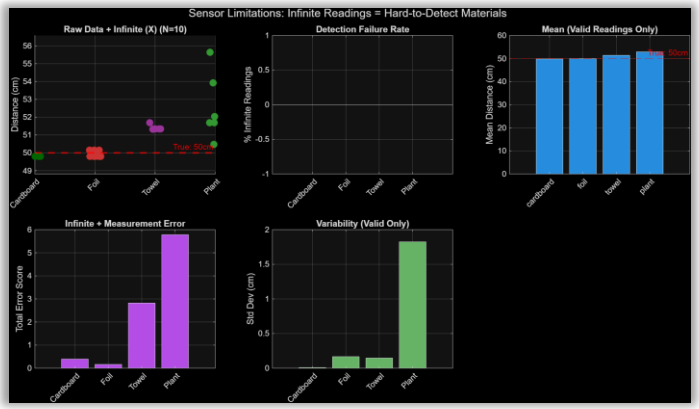


Figure 7

To improve limitations on towels and plants, processing methods were used like those in task 1. A 60-sample moving average and a standard deviation filter were used to stabilise values. Towels and plants were tested for 2 minutes at 50cm and standard deviation, mean error and values were recorded.

2.2 Results and Discussion

2.2.1 Analysis of hard-to-detect materials



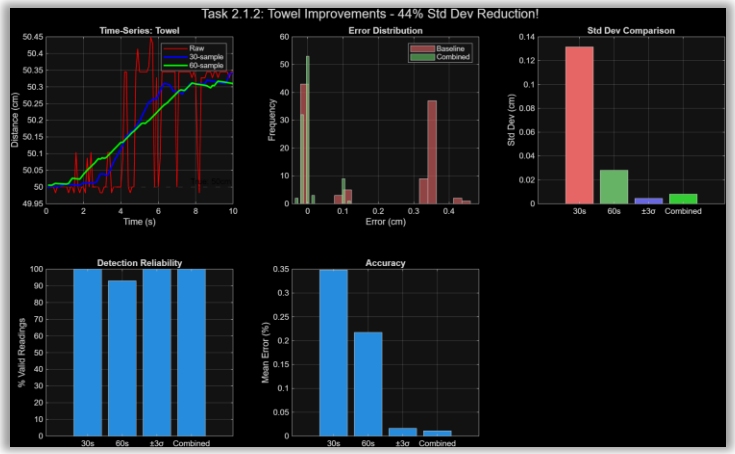

```
=== FINAL RESULTS (N=10 attempts per material) ===
Material     | Mean | Std | AbsErr | %Err | Valid | %Infinite
-------------|------|-----|--------|------|-------|----------
cardboard    | 49.8 | 0.0 |  0.2 |  0.4 |    10 |     0.0%
foil         | 49.9 | 0.2 |  0.1 |  0.2 |    10 |     0.0%
towel        | 51.4 | 0.1 |  1.4 |  2.8 |    10 |     0.0%
plant        | 52.9 | 1.8 |  2.9 |  5.8 |    10 |     0.0%
```

Figure 8

As shown in figure 8, cardboard was precise, foil the most accurate, the towel less accurate and plant the worst. Plants had the highest standard deviation of 1.8cm along with a total percentage error of 5.8%. This is much worse than foil which only had an error of 0.2% and cardboard with 0.4%. Cardboard slightly under-read, whereas the towel and plant over-read. The over-reading was likely due to towels and plants providing poor reflection of the ultrasound waves due to scattering and attenuation, increasing distance read. The high standard deviation of plants was likely due to its uneven structure causing waves to scatter. On the contrary, foil and cardboard had minimal scattering resulting in accurate and precise readings (*Ultrasonic Sensors 101: Answers to your*

frequently asked questions, n.d.). The absence of detection failures was surprising but was likely due to the shorter distance used (50cm) and the 1 second gap in between readings to prevent echoes disrupting the data.

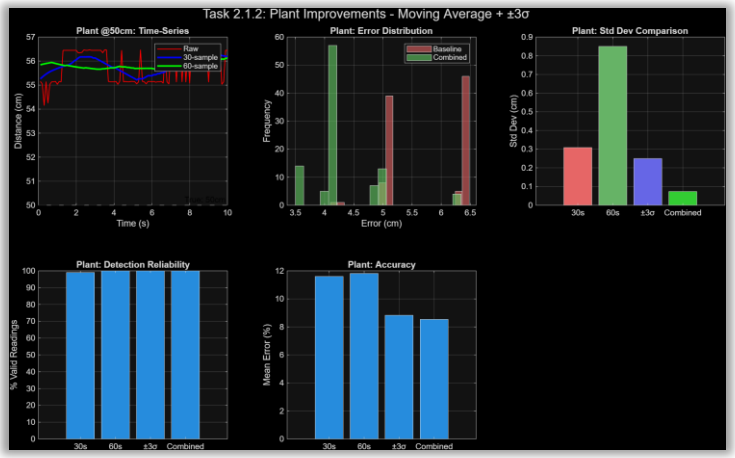
2.2.2 Improvement of detection hard-to-detect materials




```
=== TABLE 3: IMPROVEMENT RESULTS (Towel, N=100) ===
        Method          Std_Dev_cm    Pct_Valid    Mean_Error_Pct
    ____________________    __________    _________    ______________

    {'Baseline (30s)'}       0.13145           1           0.34813
    {'60-sample avg' }      0.028076        0.93           0.21732
    {'±3σ filtering' }     0.0044435           1          0.016198
    {'Both combined' }      0.008023           1          0.011044
```

Figure 9




```
TABLE 3 (PLANT): IMPROVEMENT RESULTS (N=100) ===
        Method          Std_Dev_cm    Pct_Valid    Mean_Error_Pct
    ____________________    __________    _________    ______________

    {'Baseline (30-sample)'}    0.30854        0.99           11.601
    {'60-sample avg'       }     0.8501           1           11.812
    {'±3σ filtering'       }    0.24977           1           8.8272
    {'Both combined'       }   0.073375           1           8.5355
```

Figure 10

Figures 9 and 10 above show how the sampling and filtering reduced the error and standard deviation. For the towel, standard deviation reduced by 56% and by 44% for the plant. Error distributions also narrowed, showing increased precision and smoother distance plots. For plants, accuracy improved by about 3% and by 6% for the towel. To improve further a different spline model would be needed for each material but was not possible to the absence of material detection.

2.3 Conclusion

Towels and plants were hard to measure accurately and precisely due to the ultrasonic sensors' inherent limitations. While being able to improve precision by approximately 50% and accuracy by 4-5%, the accuracy of the sensor remains a concern for security applications, which was 8% off for plants after the applied functions. However, by using the sensor at locations like doorways, uneven objects can be avoided and accuracy kept.

Task 3 – Detecting Open Doors with Motion Sensing and Counting

3.1 Methodology

To properly detect open doors and motion, the ultrasonic sensor is positioned perpendicular to the doorway as shown in figure 11. In usual conditions, where the door is closed, the ultrasonic sensor would measure a longer distance than when the door opens. When a person then moves through the door, this would then change the distance further. The number of changes can help determine the number of people who have walked through the door.

The developed software would need to have the closed-door and open-door distance to detect the door opening accurately. A person moving through the door would further decrease the distance for detection, allowing for a person counter. An alarm will be attached to notify the user of unauthorised access.

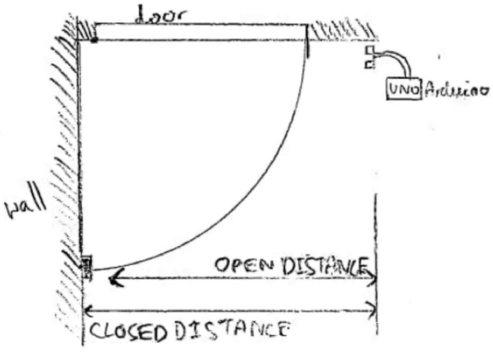


Figure 11

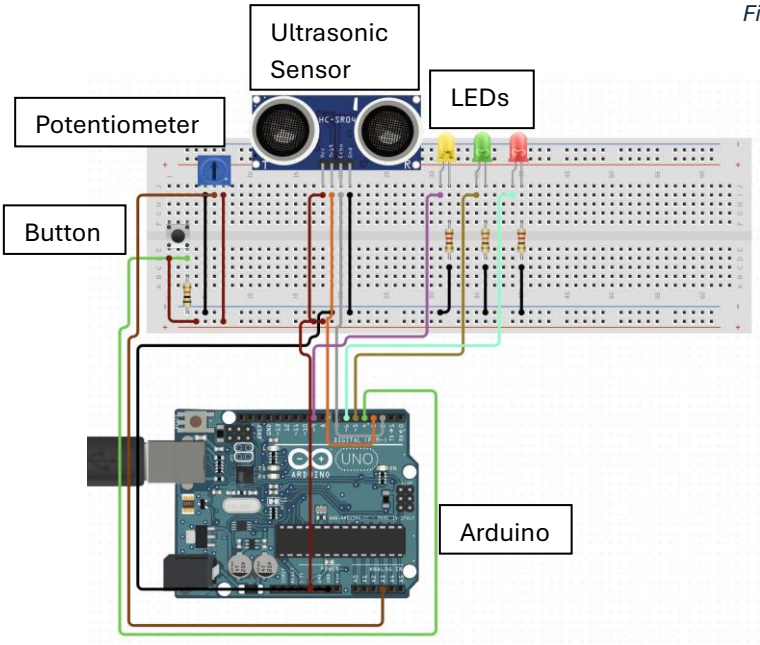
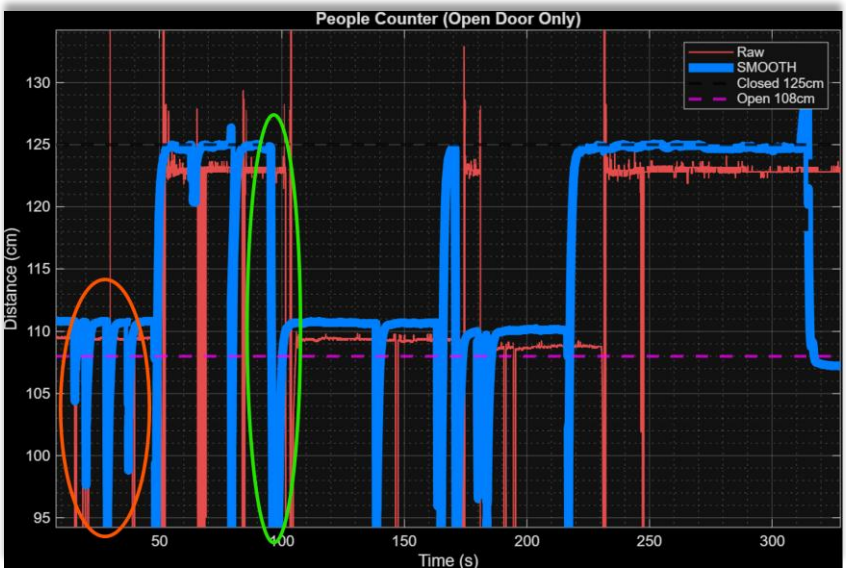


Figure 12

Figure 12 shows the wiring and parts used in task 3.

3.2 Results and Discussion

3.2.1 Door and person detection




```
t=228 | raw=109→smooth=110 | OPEN (ready) | stable=46.8s | OK | P=10 |
t=230 | raw=109→smooth=110 | OPEN (ready) | stable=48.0s | OK | P=10 |
[t=232.0] 👤 PERSON START (stable open)
t=232 | raw=126→smooth=101 | PERSON (0.2s) | stable=50.6s | OK | P=10
🎯 PERSON #11 COUNTED! (0.31s)
[t=232.9] 👤 PERSON START (stable open)
t=233 | raw=123→smooth=118 | PERSON (0.4s) | stable=51.6s | OK | P=11
[t=233.4] 🔒 CLOSED (no counting)
t=234 | raw=123→smooth=123 | PERSON (1.5s) | stable=1.0s | OK | P=11 |
t=236 | raw=123→smooth=124 | PERSON (2.7s) | stable=2.1s | OK | P=11 |
[t=t=237 | raw=123→smooth=124 | CLOSED | stable=3.2s | OK | P=11 | rej
t=238 | raw=123→smooth=125 | CLOSED | stable=4.3s | OK | P=11 | rej=6%
t=239 | raw=123→smooth=125 | CLOSED | stable=5.4s | OK | P=11 | rej=6%
```

Figure 13

Figures 13 above show the performance of the door and person detection. The sensor was able to accurately determine whether the door was closed with the expected values. An issue was the Doppler effect (shown by green circle), which caused excessively low distance values when the door was closing and high when opening. However, this only affected person detection, which relied on observing lower values when a person walks through the opening. Two procedures were implemented to solve this: locking the person count when the door is closed and a person detection cooldown to prevent double counting when a person walks through a just-opened door.

Another challenge was multiple people walking through at the same time. This could not be solved, but by tuning the smoothing function, multiple people walking one after another could be detected, as shown in the orange circle on the graph in figure 13. Tuning also improved reliability as it could respond to changes of only 0.2 seconds.

An interesting observation while placed at the doorway was temperature differences. After the door was closed for an extended period, the sensor room was warmer than outside. As shown in figure 14, when the door was opened the sensor read a lower distance of 95cm instead of the expected 108-111cm. This perhaps shows how air currents from temperature and pressure differences can affect the output and accuracy of the sensor by providing a pressure front for ultrasonic reflections. As the temperature stabilised the distance normalised.

Figure 14

3.2.2 Alarm and Security System

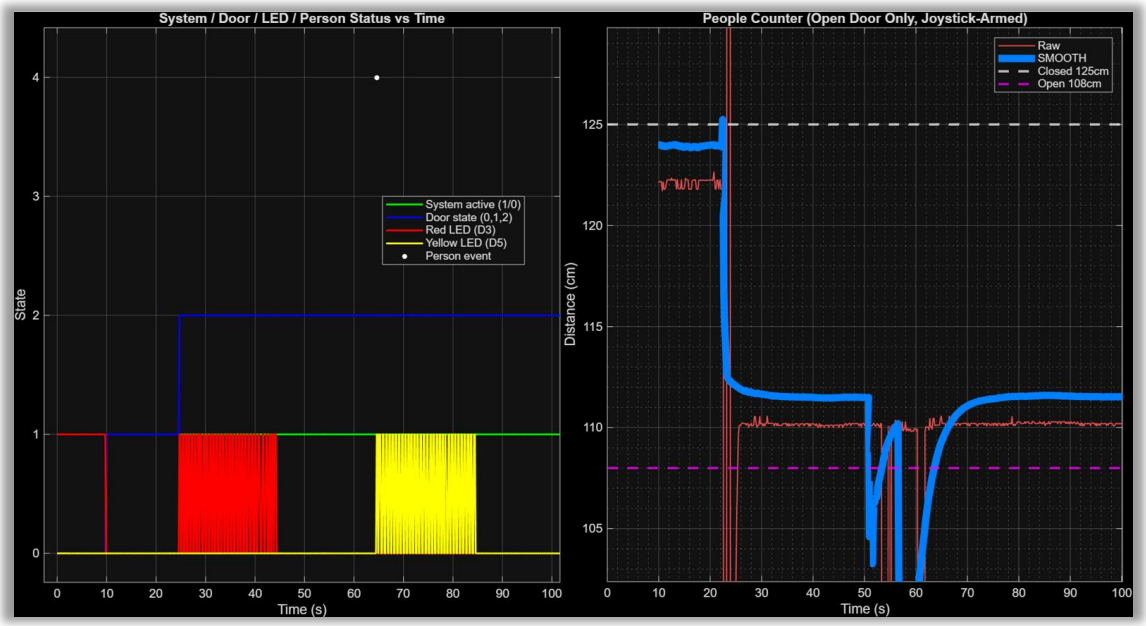


Figure 15

As seen in figure 15, between 0-10 seconds the system is off. It is then turned on using a potentiometer and button as the activation lock (the potentiometer must be at the correct position when the button is pressed). After the system is activated, the steady red LED turns off, and the green LED turns on. As the door opens at 20 seconds, the red LED begins blinking for 20 seconds. When the person walks through at 60 seconds, the yellow LED blinks for 20 seconds. This shows that the alarm system is working once activated by the user and whenever there is unauthorised access.

3.3 Conclusion

By calibrating the sensor to detect doors and people, an accurate and responsive detection and alarm system was developed. A rudimentary secure activation system was also developed so that the alarm only activates when needed and can count the number of unauthorised people. The system also has clear signals for different causes of concern such as the door opening or people crossing.

Words: 2058

References

Ultrasonic Sensor Accuracy | Senix Distance and level sensors [Online], n.d. Available from: https://senix.com/ultrasonic-sensor-accuracy/ [Accessed 7 January 2026].

Dejan, 2022. *Ultrasonic Sensor HC-SR04 and Arduino – complete guide*. *How to Mechatronics* [Online]. Available from: https://howtomechatronics.com/tutorials/arduino/ultrasonic-sensor-hc-sr04/ [Accessed 7 January 2026].

Ultrasonic Sensors 101: Answers to your frequently asked questions, n.d. *Banner Engineering* [Online]. Available from: https://www.bannerengineering.com/us/en/company/expert-insights/ultrasonic-sensors-101.html. [Accessed 7 January 2026].

Appendices (sections are hyperlinked)

Appendix A: Spline Model Generating Code – 1.2.1

```
%% FIXED Plot script - Your existing data is fine!
clear; clc; close all;

load('ultrasonic_calibration.mat', 'real_dist_cm', 'meas_mean_cm', 'pp_corr');

valid_idx = ~isnan(meas_mean_cm);
real_used = real_dist_cm(valid_idx);
meas_used = meas_mean_cm(valid_idx);

figure('Name','Fixed Calibration Plot','NumberTitle','off');
hold on; grid on; box on;
xlabel('Measured distance (cm)');  % CHANGED
ylabel('Real / Corrected distance (cm)');
title('Calibration: Raw vs Spline Model vs Ideal');

% 1. IDEAL LINE (y=x)
ideal_x = linspace(0, 220, 200);
plot(ideal_x, ideal_x, 'g', 'LineWidth', 2, 'DisplayName', 'Ideal (y=x)');


% 2. RAW DATA: real (x) vs measured (y) - shows sensor bias
plot(real_used, meas_used, 'bo-', 'LineWidth', 1.5, 'MarkerSize', 8, ...
     'DisplayName', 'Raw sensor readings');

% 3. SPLINE MODEL: CORRECT ORIENTATION
if ~isempty(pp_corr)
    meas_min = min(meas_used);
    meas_max = max(meas_used);
    meas_grid = linspace(meas_min, meas_max, 200);
    real_fit = ppval(pp_corr, meas_grid);

    % FIXED: x=measured, y=corrected_real
    plot(meas_grid, real_fit, 'r--', 'LineWidth', 2, ...
         'DisplayName', 'Spline model (measured→real)');
end

legend('Location', 'best');
xlim([0, 220]); ylim([0, 220]);
grid minor; drawnow;

%% Numerical proof it works
fprintf('\n=== VERIFICATION ===\n');
fprintf('Real(m)  Sensor(cm)  Corrected(cm)\n');
for i = 1:min(5, numel(real_used))
    corrected = ppval(pp_corr, meas_used(i));
    fprintf('%.1f     %.1f        %.1f\n', ...
            real_used(i), meas_used(i), corrected);
end
```

Appendix B: Stress Test Code – 1.2.1

```
%% SPLINE MODEL STRESS TEST - New Independent Validation Data
clear; clc; close all;

%% Load your perfect spline model
calFile = 'ultrasonic_calibration.mat';
load(calFile, 'pp_corr_smooth');
fprintf('✓ Loaded spline model from %s\n', calFile);

%% Hardware setup (YOUR pins)
arduinoPort  = 'COM3';
arduinoBoard = 'Uno';
trigPin      = 'D11';
echoPin      = 'D12';

fprintf('Connecting to Arduino...\n');
arduinoObj    = arduino(arduinoPort, arduinoBoard, 'Libraries', 'Ultrasonic');
ultrasonicObj = ultrasonic(arduinoObj, trigPin, echoPin);
fprintf('✓ Connected\n');

%% Stress test configuration
test_distances_cm = 5:5:50;  % 5cm increments to 50cm
n_reads_per_dist = 20;        % 5 readings per distance
test_real = test_distances_cm / 100;  % Convert to meters for prompts

fprintf('\n=== SPLINE MODEL STRESS TEST ===\n');
fprintf('Testing 5-50cm in 5cm steps (%d distances, %d reads each)\n', ...
        length(test_distances_cm), n_reads_per_dist);
fprintf('Press Enter after positioning each distance\n\n');

%% Storage for validation data
test_raw_cm = [];
test_corrected_cm = [];
test_true_cm = [];

%% Stress test loop
figure('Name','Live Stress Test','NumberTitle','off');
hold on; grid on; box on;
xlabel('Test Distance (cm)'); ylabel('Error (cm)');
title('Spline Model Stress Test: New Data Validation');

h_raw_err = plot(NaN, NaN, 'ro-', 'LineWidth', 2, 'MarkerSize', 8, ...
                 'DisplayName', 'Raw error');
h_corr_err = plot(NaN, NaN, 'g^-', 'LineWidth', 2, 'MarkerSize', 8, ...
                  'DisplayName', 'Corrected error');
plot([0 60], [0 0], 'bo--', 'LineWidth', 2, 'DisplayName', 'Zero error');
legend('Location', 'best'); xlim([0 55]); ylim([-10 10]); grid on; drawnow;

for i = 1:length(test_distances_cm)
    true_dist_cm = test_distances_cm(i);

    fprintf('\n=== TEST %d/%d: %.0f cm ===\n', i, length(test_distances_cm),
true_dist_cm);
    fprintf('Place target at EXACTLY %.0f cm from sensor\n', true_dist_cm);
    fprintf('Press Enter when ready...');
    pause;  % Wait for positioning

    % Take n_reads_per_dist readings
```

```
    raw_reads_m = nan(n_reads_per_dist, 1);
    for r = 1:n_reads_per_dist
        fprintf('  Reading %d/%d: ', r, n_reads_per_dist);
        max_tries = 3;
        for t = 1:max_tries
            d_m = readDistance(ultrasonicObj);
            if isfinite(d_m)
                raw_reads_m(r) = d_m * 100;  % Convert to cm
                fprintf('%.1f cm\n', raw_reads_m(r));
                break;
            else
                fprintf('Inf (try %d/%d)\n', t, max_tries);
                pause(0.1);
            end
        end
        pause(0.2);
    end

    % Process readings
    valid_reads = raw_reads_m(isfinite(raw_reads_m));
    if isempty(valid_reads)
        fprintf('  WARNING: No valid readings! Skipping...\n');
        continue;
    end

    raw_mean_cm = mean(valid_reads);
    corrected_mean_cm = ppval(pp_corr_smooth, raw_mean_cm);

    % Store for analysis
    test_raw_cm(end+1) = raw_mean_cm;
    test_corrected_cm(end+1) = corrected_mean_cm;
    test_true_cm(end+1) = true_dist_cm;

    % Live error plot update
    raw_error = raw_mean_cm - true_dist_cm;
    corr_error = corrected_mean_cm - true_dist_cm;
    set(h_raw_err, 'XData', [get(h_raw_err,'XData') true_dist_cm], ...
                   'YData', [get(h_raw_err,'YData') raw_error]);
    set(h_corr_err, 'XData', [get(h_corr_err,'XData') true_dist_cm], ...
                    'YData', [get(h_corr_err,'YData') corr_error]);
    drawnow;

    fprintf('  RAW mean: %.1f cm (error: %.1f cm)\n', raw_mean_cm, raw_error);
    fprintf('  CORR mean: %.1f cm (error: %.1f cm)\n', corrected_mean_cm,
corr_error);
    fprintf('  ✓ Spline improvement: %.1f cm\n\n', raw_error - corr_error);
end

%% Final analysis
fprintf('\n=== STRESS TEST COMPLETE ===\n');
fprintf('Tested %d new distances (independent validation)\n\n',
length(test_true_cm));

raw_errors = test_raw_cm - test_true_cm;
corr_errors = test_corrected_cm - test_true_cm;

fprintf('METRICS:\n');
fprintf('| Metric     | Raw Sensor | Spline Model |\n');
fprintf('|------------|------------|--------------|\n');
```

```
fprintf('| MAE (cm)   | %7.2f  | %7.2f   | Improvement: %.1f%%\n', ...
        mean(abs(raw_errors)), mean(abs(corr_errors)), ...
        100*(1-mean(abs(corr_errors))/mean(abs(raw_errors))));
fprintf('| RMSE (cm)  | %7.2f  | %7.2f   |\n', ...
        sqrt(mean(raw_errors.^2)), sqrt(mean(corr_errors.^2)));
fprintf('| Max |error|| %7.2f  | %7.2f   |\n', ...
        max(abs(raw_errors)), max(abs(corr_errors)));

%% Final plot with stats
figure('Name','Stress Test Results','NumberTitle','off');
subplot(2,1,1);
plot(test_true_cm, raw_errors, 'ro-', 'LineWidth', 2, 'MarkerSize', 10);
hold on; plot(test_true_cm, corr_errors, 'g^-', 'LineWidth', 2, 'MarkerSize', 10);
plot([0 55], [0 0], 'bo--', 'LineWidth', 2);
xlabel('Test distance (cm)'); ylabel('Error (cm)');
title('Stress Test: Raw vs Corrected Errors'); legend('Raw', 'Corrected');
grid on; ylim([-10 10]);

subplot(2,1,2);
histogram(raw_errors, 6, 'FaceColor', 'r', 'FaceAlpha', 0.7, 'DisplayName',
'Raw');
hold on;
histogram(corr_errors, 6, 'FaceColor', 'g', 'FaceAlpha', 0.7, 'DisplayName',
'Corrected');
xlabel('Error (cm)'); ylabel('Frequency'); title('Error Distribution');
legend; grid on;

sgtitle(sprintf('Spline Model Stress Test: %d New Points', length(test_true_cm)));

%% Cleanup
clear ultrasonicObj arduinoObj;
fprintf('\n✓ Stress test complete - model validated!\n');
```

Appendix C: Smoothing and Filtering – 1.2.2

```
%% Live Ultrasonic TIME-BASED (FIXED Plot Alignment)
clear; clc; close all;
%% Load spline model
calFile = 'ultrasonic_calibration.mat';
load(calFile, 'pp_corr_smooth');
fprintf('✓ Loaded spline model\n');
%% Hardware setup
arduinoPort  = 'COM3'; arduinoBoard = 'Uno'; trigPin = 'D11'; echoPin = 'D12';
arduinoObj = arduino(arduinoPort, arduinoBoard, 'Libraries', 'Ultrasonic');
ultrasonicObj = ultrasonic(arduinoObj, trigPin, echoPin);
fprintf('✓ Connected\n');
%% CONFIG
corr_smooth_window = 30;
final_smooth_window = 8;
%% TIME-BASED PLOT (ALL LINES ALIGNED)
figure('Name','Live TIME-BASED (Fixed)','NumberTitle','off');
hold on; grid on; box on;
xlabel('Time (s)'); ylabel('Distance (cm)');
title('Live: Raw → Filtered → SMOOTH (TIME ALIGNED)');
xlim([0 300]); ylim([0 75]);

h_raw     = plot(NaN, NaN, 'Color', [0.9 0.3 0.3], 'LineWidth', 0.5,
'DisplayName', 'Raw');
h_filtered = plot(NaN, NaN, 'Color', [0.2 0.8 0.2], 'LineWidth', 1.5,
'DisplayName', 'Filtered');
h_smooth  = plot(NaN, NaN, 'Color', [0 0.4 1], 'LineWidth', 5, 'DisplayName',
'SMOOTH');
legend('Location', 'best'); grid minor; drawnow;

%% BUFFERS (separate counters)
time_vals = []; raw_vals = []; filtered_vals = []; smooth_vals = [];
hist_buffer = []; rejected = 0; k = 0;
time_start = tic();

fprintf('\n=== TIME-BASED (ALL LINES ALIGNED) ===\n');
fprintf('Thin red=raw | Green=filtered | THICK BLUE=SMOOTH\n');
fprintf('Ctrl+C to stop\n\n');
%% MAIN LOOP
try
    while true
        k = k + 1;
        elapsed_time = toc(time_start);
        d_m = readDistance(ultrasonicObj);
        if ~isfinite(d_m)
            pause(0.1); continue;
        end
        raw_cm = d_m * 100;
        % ALWAYS store raw with time
        time_vals(end+1) = elapsed_time;
        raw_vals(end+1) = raw_cm;
        % Outlier filter
        hist_buffer(end+1) = raw_cm;
        if numel(hist_buffer) > 100
            hist_buffer = hist_buffer(end-99:end);
        end
        accept_reading = true;
        if numel(hist_buffer) > 15
```

```
            mu = mean(hist_buffer);
            sigma = std(hist_buffer);
            if sigma > 0 && abs(raw_cm - mu) > 1.8 * sigma
                accept_reading = false;
                rejected = rejected + 1;
            end
        end
        % BRANCH 1: Raw line (always updates)
        set(h_raw, 'XData', time_vals, 'YData', raw_vals);
        if ~accept_reading
            % Rejected: only raw updates, others hold last value
            drawnow limitrate;
            pause(0.1);
            continue;
        end
        % BRANCH 2: Accepted → spline → smooth
        corr_cm = ppval(pp_corr_smooth, raw_cm);
        % Heavy moving average (on filtered_corr_vals)
        filtered_vals(end+1) = corr_cm;
        n_filt = numel(filtered_vals);
        if n_filt <= corr_smooth_window
            smooth_cm = mean(filtered_vals);
        else
            smooth_cm = mean(filtered_vals(end-corr_smooth_window+1:end));
        end
        % Final light polish
        n_smooth = numel(smooth_vals);
        if n_smooth == 0
            final_smooth_cm = smooth_cm;
        else
            lookback = min(final_smooth_window-1, n_smooth);
            recent_smooth = smooth_vals(end-lookback+1:end);
            final_smooth_cm = mean([recent_smooth, smooth_cm]);
        end
        smooth_vals(end+1) = final_smooth_cm;
        % Update filtered + smooth lines (same time as raw)
        set(h_filtered, 'XData', time_vals(1:n_filt), 'YData', filtered_vals);
        set(h_smooth, 'XData', time_vals(1:n_smooth+1), 'YData', smooth_vals);
        % Auto-scroll
        if elapsed_time > 300
            xlim([elapsed_time-300, elapsed_time+30]);
        end
        if mod(k, 10) == 0
            reject_rate = 100 * rejected / k;
            fprintf('t=%.0fs | RAW=%.1f → SMOOTH=%.2f | Reject=%.0f%%\n', ...
                    elapsed_time, raw_cm, final_smooth_cm, reject_rate);
        end
        drawnow limitrate;
        pause(0.00000001);
    end
catch ME
    if ~strcmp(ME.identifier, 'MATLAB:cancel')
        rethrow(ME);
    end
end

%% Cleanup
clear ultrasonicObj arduinoObj;
fprintf('\n✓ Fixed time-based complete\n');
```

Appendix D: Hard-to-Detect observations – 2.2.1

```
%% Ultrasound Sensor Live Data Collection & Analysis Script (Infinite Readings)
% TRACKS Infinite readings as "hard to detect" metric
% Arduino COM3, Trig:D11, Echo:D12

clear; clc; close all;

%% CONFIGURATION SECTION
NUM_SAMPLES = 10;        % Number of measurements per material
SAMPLE_DELAY = 1.0;      % Seconds between samples
true_distance = 0.50;    % 50 cm in meters (expected)

fprintf('=== ULTRASONIC SENSOR MATERIAL ANALYSIS ===\n');
fprintf('CONFIG: %d samples/material, %.1f sec delay\n', NUM_SAMPLES,
SAMPLE_DELAY);
fprintf('HC-SR04: Trig->D11, Echo->D12, VCC->5V, GND->GND (COM3)\n');
fprintf('Infinite readings = "Hard to detect" materials!\n\n');

%% SETUP Arduino
a = arduino('COM3', 'Uno', 'Libraries', 'Ultrasonic');
sensor = ultrasonic(a, 'D11', 'D12', 'OutputFormat', 'double');
fprintf('✓ Connected to Arduino COM3 (D11/D12)\n\n');

materials = {'Cardboard', 'Foil', 'Towel', 'Plant'};
measurements = struct();
infinite_counts = struct();

%% LIVE DATA COLLECTION LOOP
for i = 1:length(materials)
    material = materials{i};
    fprintf('\n=== %s TEST (%d samples) ===\n', material, NUM_SAMPLES);
    fprintf('Place %s at 50cm mark. Press ENTER when ready...\n', material);
    pause;

    fprintf('Collecting %d measurements (%.1fs intervals)...\n', NUM_SAMPLES,
SAMPLE_DELAY);
    data = [];
    infinite_count = 0;
    valid_count = 0;

    for j = 1:NUM_SAMPLES  % Fixed loop - collect exactly NUM_SAMPLES attempts
        pause(SAMPLE_DELAY);
        dist = readDistance(sensor);

        if isinf(dist) || dist < 0  % INFINITE/NO REFLECTION DETECTED
            infinite_count = infinite_count + 1;
            fprintf('  Sample %3d/%d: **INFINITE** (no reflection)\n', j,
NUM_SAMPLES);
        elseif dist > 0.02 && dist < 4.0 && ~isnan(dist)
            valid_count = valid_count + 1;
            data(valid_count) = dist;
            fprintf('  Sample %3d/%d: %.1f cm\n', j, NUM_SAMPLES, dist*100);
        else
            fprintf('  Sample %3d/%d: INVALID (%.1f cm) - ignored\n', j,
NUM_SAMPLES, dist*100);
        end
    end
```

```
    % Store results
    field_name = lower(strrep(material, ' ', '_'));
    if valid_count > 0
        measurements.(field_name) = data(1:valid_count);
    else
        measurements.(field_name) = NaN;  % No valid readings
    end
    infinite_counts.(field_name) = infinite_count;

    fprintf('✓ %s complete! Valid:%d, Infinite:%d (%.1f%%)\n', ...
        material, valid_count, infinite_count, 100*infinite_count/NUM_SAMPLES);
end

%% ANALYSIS SECTION
materials_lower = fieldnames(measurements);
results = table('Size', [0 7], ...
    'VariableTypes', {'string', 'double', 'double', 'double', 'double', 'double',
'double'}, ...
    'VariableNames', {'Material', 'Mean_cm', 'Std_cm', 'Abs_Error_cm',
'Pct_Error', 'N_Valid', 'Pct_Infinite'});

for i = 1:length(materials_lower)
    material = materials_lower{i};
    inf_count = infinite_counts.(material);

    if isnan(measurements.(material))
        % No valid readings
        new_row = {strrep(material, '_', ' '), NaN, NaN, NaN, NaN, 0,
100*inf_count/NUM_SAMPLES};
    else
        data = measurements.(material);
        mean_dist = mean(data);
        std_dist = std(data);
        abs_error = abs(mean_dist - true_distance);
        pct_error = (abs_error / true_distance) * 100;
        new_row = {strrep(material, '_', ' '), mean_dist*100, std_dist*100, ...
                   abs_error*100, pct_error, length(data),
100*inf_count/NUM_SAMPLES};
    end
    results = [results; new_row];
end

%% DISPLAY RESULTS
fprintf('\n=== FINAL RESULTS (N=%d attempts per material) ===\n', NUM_SAMPLES);
fprintf('Material     | Mean | Std | AbsErr | %%Err | Valid | %%Infinite\n');
fprintf('-------------|------|-----|--------|------|-------|----------\n');
for i = 1:height(results)
    if isnan(results.Mean_cm(i))
        fprintf('%-11s | ----- | --- | ----- | ---- | %5d | %7.1f%%\n', ...
            results.Material{i}, results.N_Valid(i), results.Pct_Infinite(i));
    else
        fprintf('%-11s | %4.1f | %3.1f | %4.1f | %4.1f | %5d | %7.1f%%\n', ...
            results.Material{i}, results.Mean_cm(i), results.Std_cm(i), ...
            results.Abs_Error_cm(i), results.Pct_Error(i), ...
            results.N_Valid(i), results.Pct_Infinite(i));
    end
end

%% VISUALIZATION
```

```
figure('Position', [100, 100, 1400, 700]);

% 1. Raw measurements + Infinite markers
subplot(2,3,1);
hold on;
colors = [0 0.4 0; 0.8 0.2 0.2; 0.6 0.2 0.6; 0.2 0.6 0.2];
for i = 1:length(materials_lower)
    field_name = materials_lower{i};
    if ~isnan(measurements.(field_name))
        data = measurements.(field_name) * 100;
        scatter(i*ones(size(data)), data, 80, colors(i,:), 'filled', ...
               'jitter', 'on', 'jitterAmount', 0.15);
    end
    % Infinite readings as X markers
    inf_pct = infinite_counts.(field_name) / NUM_SAMPLES;
    scatter(i, 450, 200, colors(i,:), 'X', 'LineWidth', 4);  % Top of plot
end
yline(50, 'r--', 'True: 50cm', 'LineWidth', 2);
set(gca, 'XTick', 1:4, 'XTickLabel', materials, 'XTickLabelRotation', 45, ...
    'YLim', [0 500]);
ylabel('Distance (cm)');
title(sprintf('Raw Data + Infinite (X) (N=%d)', NUM_SAMPLES));
grid on;

% 2. Infinite % bar (NEW - KEY METRIC!)
subplot(2,3,2);
b = bar(results.Pct_Infinite, 'FaceColor', [0.9 0.2 0.2]);
ylabel('% Infinite Readings');
title('Detection Failure Rate');
set(gca, 'XTickLabel', materials, 'XTickLabelRotation', 45);
grid on;

% 3. Mean distances (only valid readings)
subplot(2,3,3);
valid_mask = ~isnan(results.Mean_cm);
bar_data = results.Mean_cm(valid_mask);
materials_valid = results.Material(valid_mask);
if ~isempty(bar_data)
    bar(1:length(bar_data), bar_data);
    hold on;
    yline(50, 'r--', 'True: 50cm');
end
set(gca, 'XTickLabel', materials_valid, 'XTickLabelRotation', 45);
ylabel('Mean Distance (cm)');
title('Mean (Valid Readings Only)');
grid on;

% 4. Combined error metric
subplot(2,3,4);
combined_error = results.Pct_Infinite + (results.Pct_Error .* (1-
results.Pct_Infinite/100));
bar(combined_error, 'FaceColor', [0.7 0.3 0.9]);
ylabel('Total Error Score');
title('Infinite + Measurement Error');
set(gca, 'XTickLabel', materials, 'XTickLabelRotation', 45);
grid on;

% 5. Std Dev (valid readings)
subplot(2,3,5);
```

```
valid_std = results.Std_cm(~isnan(results.Std_cm));
if ~isempty(valid_std)
    bar(valid_std, 'FaceColor', [0.4 0.7 0.4]);
end
ylabel('Std Dev (cm)');
title('Variability (Valid Only)');

set(gca, 'XTickLabel', materials(~isnan(results.Std_cm)), 'XTickLabelRotation',
45);
grid on;

sgtitle(sprintf('Sensor Limitations: Infinite Readings = Hard-to-Detect
Materials'), 'FontSize', 14);

%% EXPORT
writetable(results, sprintf('sensor_results_N%d_with_infinite.xlsx',
NUM_SAMPLES));
save(sprintf('sensor_data_N%d_with_infinite.mat', NUM_SAMPLES), 'measurements',
'results', 'infinite_counts');
fprintf('\n✓ Saved with infinite reading analysis!\n');

%% CLEANUP
clear a sensor;
fprintf('\n=== COMPLETE ===\n');
fprintf('HIGH %%Infinite = Towels/Plants (sound absorbing/scattering)\n');
fprintf('LOW  %%Infinite = Cardboard/Foil (good reflection)\n');
```

Appendix E: Improving Hard-to-Detect Performance – 2.2.2

```
%% Task 2.1.2 IMPROVEMENTS - Plant Signal Processing Fixes
% Tests: 30-sample vs 60-sample averaging + ±1.8σ vs ±3σ filtering
% Arduino COM3, Trig:D11, Echo:D12, PLANT at 50cm, N=100 readings each

clear; clc; close all;

%% CONFIGURATION
NUM_READINGS = 100;      % Total raw readings per test (N=100)
true_distance = 0.50;    % 50 cm expected
SAMPLE_DELAY = 0.1;      % Fast sampling for continuous data

fprintf('=== TASK 2.1.2 IMPROVEMENTS - PLANT at 50cm ===\n');
fprintf('4 Tests: Baseline(30s), 60-sample, ±3σ, Combined (N=%d each)\n',
NUM_READINGS);

%% SETUP Arduino
a = arduino('COM3', 'Uno', 'Libraries', 'Ultrasonic');
sensor = ultrasonic(a, 'D11', 'D12', 'OutputFormat', 'double');
fprintf('✓ Sensor ready. Place PLANT at 50cm.\n\n');

%% TEST 1: BASELINE (30-sample moving average, ±1.8σ)
fprintf('=== TEST 1: BASELINE (30-sample, ±1.8σ) - PLANT ===\n');
pause;  % Position plant
raw_data1 = collect_raw_data(sensor, NUM_READINGS, SAMPLE_DELAY);
[baseline_mean, baseline_std, baseline_valid_pct] = process_data(raw_data1, 30,
1.8);
baseline_error = abs(baseline_mean - true_distance)/true_distance*100;

%% TEST 2: EXTENDED AVERAGING (60-sample moving average, ±1.8σ)
fprintf('\n=== TEST 2: 60-SAMPLE AVERAGING (±1.8σ) - PLANT ===\n');
pause;  % Reposition plant if needed
raw_data2 = collect_raw_data(sensor, NUM_READINGS, SAMPLE_DELAY);
[avg60_mean, avg60_std, avg60_valid_pct] = process_data(raw_data2, 60, 1.8);
avg60_error = abs(avg60_mean - true_distance)/true_distance*100;

%% TEST 3: ADAPTIVE FILTERING (±3σ, 30-sample)
fprintf('\n=== TEST 3: ±3σ FILTERING (30-sample) - PLANT ===\n');
pause;
raw_data3 = collect_raw_data(sensor, NUM_READINGS, SAMPLE_DELAY);
[filter3_mean, filter3_std, filter3_valid_pct] = process_data(raw_data3, 30, 3.0);
filter3_error = abs(filter3_mean - true_distance)/true_distance*100;

%% TEST 4: COMBINED (60-sample + ±3σ)
fprintf('\n=== TEST 4: COMBINED (60-sample + ±3σ) - PLANT ===\n');
pause;
raw_data4 = collect_raw_data(sensor, NUM_READINGS, SAMPLE_DELAY);
[combined_mean, combined_std, combined_valid_pct] = process_data(raw_data4, 60,
3.0);
combined_error = abs(combined_mean - true_distance)/true_distance*100;

%% RESULTS TABLE 3 (PLANT)
results_table = table(...
    {'Baseline (30-sample)'; '60-sample avg'; '±3σ filtering'; 'Both combined'},
...
    [baseline_std*100, avg60_std*100, filter3_std*100, combined_std*100]', ...
    [baseline_valid_pct, avg60_valid_pct, filter3_valid_pct, combined_valid_pct]',
...
```

```
    [baseline_error, avg60_error, filter3_error, combined_error]', ...
    'VariableNames', {'Method', 'Std_Dev_cm', 'Pct_Valid', 'Mean_Error_Pct'});

fprintf('\n=== TABLE 3 (PLANT): IMPROVEMENT RESULTS (N=%d) ===\n', NUM_READINGS);
disp(results_table);

%% VISUALIZATION
figure('Position', [100, 100, 1600, 800]);

% Time-series: Raw vs 30-sample vs 60-sample (Plant)
subplot(2,3,1);
t = (1:NUM_READINGS)*SAMPLE_DELAY;
plot(t, raw_data1*100, 'r-', 'LineWidth', 1); hold on;
plot(t, movmean(raw_data1*100, 30), 'b-', 'LineWidth', 2);
plot(t, movmean(raw_data1*100, 60), 'g-', 'LineWidth', 2);
yline(50, 'k--', 'True: 50cm');
legend('Raw', '30-sample', '60-sample');
xlabel('Time (s)'); ylabel('Distance (cm)');
title('Plant @50cm: Time-Series');
grid on;

% Error histograms: Baseline vs Combined (Plant)
subplot(2,3,2);
valid_baseline = raw_data1(~isinf(raw_data1) & ~isnan(raw_data1));
histogram((valid_baseline*100 - 50), 20, 'FaceColor', [0.9 0.4 0.4]); hold on;
valid_combined = raw_data4(~isinf(raw_data4) & ~isnan(raw_data4));
histogram((valid_combined*100 - 50), 20, 'FaceColor', [0.4 0.8 0.4]);
xlabel('Error (cm)'); ylabel('Frequency');
title('Plant: Error Distribution');
legend('Baseline', 'Combined');
grid on;

% Std dev comparison
subplot(2,3,3);
bar_data = [baseline_std*100, avg60_std*100, filter3_std*100, combined_std*100];
b = bar(bar_data); b.FaceColor = 'flat';
b.CData(1,:) = [0.9 0.4 0.4];
b.CData(2,:) = [0.4 0.7 0.4];
b.CData(3,:) = [0.4 0.4 0.9];
b.CData(4,:) = [0.2 0.8 0.2];
set(gca, 'XTickLabel', {'30s', '60s', '±3σ', 'Combined'});
ylabel('Std Dev (cm)');
title('Plant: Std Dev Comparison');
grid on;

% % Valid readings
subplot(2,3,4);
bar([baseline_valid_pct, avg60_valid_pct, filter3_valid_pct,
combined_valid_pct]*100);
ylabel('% Valid Readings');
title('Plant: Detection Reliability');
set(gca, 'XTickLabel', {'30s', '60s', '±3σ', 'Combined'});
grid on;

% Mean error (%)
subplot(2,3,5);
bar([baseline_error, avg60_error, filter3_error, combined_error]);
ylabel('Mean Error (%)');
title('Plant: Accuracy');
```

```
set(gca, 'XTickLabel', {'30s', '60s', '±3σ', 'Combined'});
grid on;

sgtitle('Task 2.1.2: Plant Improvements - Moving Average + ±3σ', 'FontSize', 16);

%% EXPORT
writetable(results_table, 'Table3_Improvements_Plant.xlsx');
save('task2_1_2_improvements_plant.mat', 'results_table', ...
     'raw_data1', 'raw_data2', 'raw_data3', 'raw_data4');
fprintf('\n✓ Saved: Table3_Improvements_Plant.xlsx &
task2_1_2_improvements_plant.mat\n');

%% CLEANUP
clear a sensor;
fprintf('\n=== TASK 2.1.2 PLANT TEST COMPLETE ===\n');

%% HELPER FUNCTIONS (same as towel script)
function raw_data = collect_raw_data(sensor, n_readings, delay)
    fprintf('Collecting %d raw readings...\n', n_readings);
    raw_data = zeros(1, n_readings);
    for i = 1:n_readings
        pause(delay);
        raw_data(i) = readDistance(sensor);
        if mod(i, 20) == 0
            fprintf('  %d/%d collected\n', i, n_readings);
        end
    end
    fprintf('✓ Raw data collection complete\n');
end

function [mean_dist, std_dist, valid_pct] = process_data(raw_data, window_size,
sigma_threshold)
    % Moving average smoothing
    smoothed = movmean(raw_data, window_size);   % [web:57]

    % Outlier rejection around mean (±sigma_threshold·σ)
    valid_mask = ~isinf(smoothed) & ~isnan(smoothed);
    if sum(valid_mask) > 0

        mu = mean(smoothed(valid_mask));
        sigma = std(smoothed(valid_mask));
        in_band = abs(smoothed - mu) <= sigma_threshold * sigma;  % [web:56]
        final_data = smoothed(in_band & valid_mask);
    else
        final_data = [];
        in_band = false(size(smoothed));
    end

    if ~isempty(final_data)
        mean_dist = mean(final_data);
        std_dist = std(final_data);
        valid_pct = sum(in_band & valid_mask) / numel(raw_data);
    else
        mean_dist = NaN;
        std_dist = NaN;
        valid_pct = 0;
    end
end
```

Appendix F: Doorway Detection with Alarm – 3.2.1

```
%% Live Ultrasonic DOOR & PEOPLE COUNTER + JOYSTICK + 20s LED BLINKS + STATUS PLOT
clear; clc; close all;

%% Load spline model
calFile = 'ultrasonic_calibration.mat';
load(calFile, 'pp_corr_smooth');
fprintf('✓ Loaded spline model\n');

%% Hardware
arduinoPort = 'COM3'; arduinoBoard = 'Uno';
trigPin  = 'D11'; echoPin = 'D12';
joyPin   = 'A0';          % joystick Y (only control)
ledActivePin   = 'D2';    % green: system ACTIVE
ledInactivePin = 'D3';    % red: system INACTIVE + door warning blink
ledPersonPin   = 'D5';    % yellow: person blink

arduinoObj = arduino(arduinoPort, arduinoBoard, 'Libraries', 'Ultrasonic');
ultrasonicObj = ultrasonic(arduinoObj, trigPin, echoPin);
fprintf('✓ Connected\n');

%% CONFIG
corr_smooth_window  = 6;
final_smooth_window = 2;

door_closed_cm = 125; door_open_cm = 108; door_tol_cm = 6;
door_closed_lo = door_closed_cm - door_tol_cm;
door_closed_hi = door_closed_cm + door_tol_cm;
door_open_lo   = door_open_cm   - door_tol_cm;
door_open_hi   = door_open_cm   + door_tol_cm;

min_person_time = 0.15;
max_person_time = 3.5;

% Joystick thresholds
joy_debounce_time = 0.2;   % seconds for toggling
joy_high_threshold = 3.5;  % V for "up" position

%% Activation state
system_active   = 0;    % 0 = inactive, 1 = active
last_joy_high   = 0;
last_joy_time   = 0;

% Pin modes
configurePin(arduinoObj, ledActivePin,   'DigitalOutput');
configurePin(arduinoObj, ledInactivePin, 'DigitalOutput');
configurePin(arduinoObj, ledPersonPin,   'DigitalOutput');

writeDigitalPin(arduinoObj, ledActivePin,   0);
writeDigitalPin(arduinoObj, ledInactivePin, 1);
writeDigitalPin(arduinoObj, ledPersonPin,   0);

%% FIGURE 1: Distance plot
fig1 = figure('Name','People Counter','NumberTitle','off');
subplot(1,1,1);
hold on; grid on; box on;
xlabel('Time (s)'); ylabel('Distance (cm)');
title('People Counter (Open Door Only, Joystick-Armed)');
```

```
xlim([0 300]); ylim([95 135]);

h_raw    = plot(NaN, NaN, 'Color', [0.9 0.3 0.3], 'LineWidth', 1, 'DisplayName',
'Raw');
h_smooth = plot(NaN, NaN, 'Color', [0 0.5 1],   'LineWidth', 6, 'DisplayName',
'SMOOTH');

yline(door_closed_cm, '--w', 'LineWidth', 2, 'DisplayName', 'Closed 125cm'); %
white
yline(door_open_cm,   '--m', 'LineWidth', 2, 'DisplayName', 'Open 108cm');

legend('Location','northeast'); grid minor;

%% FIGURE 2: Status plot
fig2 = figure('Name','Status (Door, LEDs, People)','NumberTitle','off');
hold on; grid on; box on;
xlabel('Time (s)');
ylabel('State');
title('System / Door / LED / Person Status vs Time');
xlim([0 300]); ylim([-0.5 4.5]);

h_sysActive = plot(NaN, NaN, 'g-', 'LineWidth', 1.5, 'DisplayName', 'System active
(1/0)');
h_doorState = plot(NaN, NaN, 'b-', 'LineWidth', 1.5, 'DisplayName', 'Door state
(0,1,2)');
h_ledRed    = plot(NaN, NaN, 'r-', 'LineWidth', 1.5, 'DisplayName', 'Red LED
(D3)');
h_ledYellow = plot(NaN, NaN, 'y-', 'LineWidth', 1.5, 'DisplayName', 'Yellow LED
(D5)');
h_personEvt = plot(NaN, NaN, 'w.', 'MarkerSize', 12, 'DisplayName', 'Person
event'); % white markers

yticks(0:4);
yticklabels({'0','1','2','3','4'});
legend('Location','northwest');

%% BUFFERS
time_vals = []; raw_vals = []; smooth_vals = []; hist_buffer = [];
rejected = 0; k = 0; time_start = tic();

person_count = 0;
base_state = 0;              % 0=unstable, 1=closed, 2=open
person_active = 0;
person_start_time = NaN;
base_stable_time = 0;
person_block_time = 0;

% LED blink state
door_blink_active      = 0;
door_blink_end_time    = 0;
door_blink_period      = 0.2;   % fast blink (s)
door_last_toggle_time  = 0;
door_led_state         = 0;

person_blink_active    = 0;
person_blink_end_time  = 0;
person_blink_period    = 0.2;   % fast blink (s)
person_last_toggle_time= 0;
person_led_state       = 0;
```

```
blink_total_duration   = 20;    % total blink time (s)

% STATUS BUFFERS for figure 2
time_status   = [];
sysActive_vec = [];
doorState_vec = [];
ledRed_vec    = [];
ledYellow_vec = [];
personEvt_time = [];
personEvt_val  = [];

fprintf('\n=== PEOPLE COUNTER + JOYSTICK (Y ONLY) + 20s BLINKS + STATUS PLOT
===\n');
fprintf('Joystick Y on A0 toggles system when pushed ABOVE %.1f V (no button)\n',
joy_high_threshold);

try
    while true
        k = k + 1; t = toc(time_start);

        %% -------- ACTIVATION LOGIC (JOYSTICK Y ONLY) --------
        joy_val = readVoltage(arduinoObj, joyPin);  % 0–5 V
        joy_high = (joy_val > joy_high_threshold);

        % Toggle system_active on rising edge of joy_high
        if joy_high && ~last_joy_high && (t - last_joy_time) > joy_debounce_time
            last_joy_time = t;
            system_active = 1 - system_active;
            if system_active == 1
                fprintf('[t=%.2f] SYSTEM ACTIVATED (joystick Y high)\n', t);
                person_active = 0;
                person_start_time = NaN;
            else
                fprintf('[t=%.2f] SYSTEM DEACTIVATED (joystick Y high)\n', t);
                person_active = 0;
                person_start_time = NaN;
            end
        end
        last_joy_high = joy_high;

        %% -------- LED BASE STATES (GREEN/RED) --------
        if system_active == 1
            writeDigitalPin(arduinoObj, ledActivePin,   1); % green ON
        else
            writeDigitalPin(arduinoObj, ledActivePin,   0);
        end

        % If INACTIVE: solid red, skip ultrasonic logic
        if system_active == 0
            writeDigitalPin(arduinoObj, ledInactivePin, 1); % red solid
            writeDigitalPin(arduinoObj, ledPersonPin,   0);
            person_blink_active = 0;
            door_blink_active   = 0;

            % Update status buffers
            time_status(end+1)   = t;
            sysActive_vec(end+1) = system_active;
            doorState_vec(end+1) = base_state;
            ledRed_vec(end+1)    = 1;
```

```
    ledYellow_vec(end+1) = 0;

    figure(fig2);
    set(h_sysActive, 'XData', time_status, 'YData', sysActive_vec);
    set(h_doorState, 'XData', time_status, 'YData', doorState_vec);
    set(h_ledRed,    'XData', time_status, 'YData', ledRed_vec);
    set(h_ledYellow, 'XData', time_status, 'YData', ledYellow_vec);
    set(h_personEvt, 'XData', personEvt_time, 'YData', personEvt_val);

    if t > 300
        figure(fig1); xlim([t-300 t+20]);
        figure(fig2); xlim([t-300 t+20]);
    end

    drawnow limitrate;
    pause(0.02);
    continue;
end

%% -------- ULTRASONIC + PEOPLE COUNTER (ACTIVE ONLY) --------
raw_m = readDistance(ultrasonicObj);
if ~isfinite(raw_m) || raw_m < 0.05
    pause(0.02); continue;
end
raw_cm = raw_m * 100;

time_vals(end+1) = t; raw_vals(end+1) = raw_cm;

hist_buffer(end+1) = raw_cm;
if length(hist_buffer) > 60
    hist_buffer = hist_buffer(end-59:end);
end

accept = 1;
if length(hist_buffer) > 10
    mu = mean(hist_buffer); sigma = std(hist_buffer);
    if sigma > 0 && abs(raw_cm - mu) > 2.2 * sigma
        accept = 0; rejected = rejected + 1;
    end
end

figure(fig1);
set(h_raw, 'XData', time_vals, 'YData', raw_vals);
if accept == 0
    drawnow limitrate; pause(0.015); continue;
end

corr_cm = ppval(pp_corr_smooth, raw_cm);
smooth_vals(end+1) = corr_cm;

n_smooth = length(smooth_vals);
if n_smooth <= corr_smooth_window
    final_smooth = mean(smooth_vals);
else
    final_smooth = mean(smooth_vals(end-corr_smooth_window+1:end));
end

if n_smooth > 1
    final_smooth = 0.7 * final_smooth + 0.3 * smooth_vals(end-1);
```

```
        end

        smooth_vals(end) = final_smooth;

        set(h_smooth, 'XData', time_vals(1:length(smooth_vals)), 'YData',
smooth_vals);
        if t > 300
            xlim([t-300 t+20]);
        end

        d = final_smooth;

        %% -------- BASE STATE DETECTION --------
        if d >= door_closed_lo && d <= door_closed_hi
            new_base = 1;  % closed
        elseif d >= door_open_lo && d <= door_open_hi
            new_base = 2;  % open
        else
            new_base = 0;  % unstable / person
        end

        door_change_active = (t - person_block_time) < 0.6;

        if new_base ~= base_state && new_base ~= 0
            base_stable_time = t;
            base_state = new_base;
            person_block_time = t;

            if base_state == 1
                fprintf('[t=%.1f] 🔒 CLOSED (no counting)\n', t);
            else
                fprintf('[t=%.1f] 🚪 OPEN (ready)\n', t);
                door_blink_active     = 1;
                door_blink_end_time   = t + blink_total_duration;
                door_last_toggle_time = t;
                door_led_state        = 1;
            end
        end

        %% -------- PERSON DETECTION (OPEN DOOR ONLY) --------
        is_stable_open = (base_state == 2 && (t - base_stable_time) > 0.4 &&
door_change_active == 0);
        is_person_deviation = (new_base == 0);

        can_detect_person = (is_stable_open == 1 && is_person_deviation == 1);

        if can_detect_person == 1
            if person_active == 0
                person_active = 1;
                person_start_time = t;
                fprintf('[t=%.1f] 🧍 PERSON START (stable open)\n', t);
            end
        end

        if person_active == 1
            duration = t - person_start_time;

            person_end_condition = (base_state == 2 && new_base ~= 0 &&
is_person_deviation == 0);
```

```
            if person_end_condition == 1 && duration >= min_person_time &&
duration <= max_person_time
                person_count = person_count + 1;
                fprintf('🎯 PERSON #%d COUNTED! (%.2fs)\n', person_count,
duration);

                person_blink_active      = 1;
                person_blink_end_time    = t + blink_total_duration;
                person_last_toggle_time  = t;
                person_led_state         = 1;

                personEvt_time(end+1) = t;
                personEvt_val(end+1)  = 4;   % plotted at y=4

                person_active = 0;
                person_start_time = NaN;
            elseif duration > max_person_time
                fprintf('[t=%.1f] Person timeout\n', t);
                person_active = 0;
                person_start_time = NaN;
            end
        end

        %% -------- LED BLINK UPDATES (non-blocking) --------
        % RED (D3)
        if door_blink_active == 1 && t < door_blink_end_time
            if (t - door_last_toggle_time) >= door_blink_period
                door_last_toggle_time = t;
                door_led_state = 1 - door_led_state;
            end
            writeDigitalPin(arduinoObj, ledInactivePin, door_led_state);
        else
            door_blink_active = 0;
            writeDigitalPin(arduinoObj, ledInactivePin, 0);
            door_led_state = 0;
        end

        % YELLOW (D5)
        if person_blink_active == 1 && t < person_blink_end_time
            if (t - person_last_toggle_time) >= person_blink_period
                person_last_toggle_time = t;
                person_led_state = 1 - person_led_state;
            end
            writeDigitalPin(arduinoObj, ledPersonPin, person_led_state);
        else
            person_blink_active = 0;
            writeDigitalPin(arduinoObj, ledPersonPin, 0);
            person_led_state = 0;
        end

        %% -------- UPDATE STATUS BUFFERS & PLOT --------
        time_status(end+1)   = t;
        sysActive_vec(end+1) = system_active;
        doorState_vec(end+1) = base_state;
        ledRed_vec(end+1)    = door_led_state;
        ledYellow_vec(end+1) = person_led_state;

        figure(fig2);
        set(h_sysActive, 'XData', time_status, 'YData', sysActive_vec);
```

```
        set(h_doorState, 'XData', time_status, 'YData', doorState_vec);
        set(h_ledRed,    'XData', time_status, 'YData', ledRed_vec);
        set(h_ledYellow, 'XData', time_status, 'YData', ledYellow_vec);
        set(h_personEvt, 'XData', personEvt_time, 'YData', personEvt_val);

        if t > 300
            figure(fig1); xlim([t-300 t+20]);
            figure(fig2); xlim([t-300 t+20]);
        end

        %% -------- STATUS TEXT --------
        if mod(k, 25) == 0
            reject_pct = 100 * rejected / k;

            if base_state == 1
                base_str = 'CLOSED';
            elseif base_state == 2
                if door_change_active == 1
                    base_str = 'OPEN (settling)';
                else
                    base_str = 'OPEN (ready)';
                end
            else
                base_str = 'UNSTABLE';
            end

            if person_active == 1
                status_str = sprintf('PERSON (%.1fs)', t-person_start_time);
            else
                status_str = base_str;
            end

            stability = round((t - base_stable_time)*10)/10;

            if door_change_active == 1
                block_str = 'BLOCKED';
            else
                block_str = 'OK';
            end

            fprintf('t=%.0f | raw=%.0f→smooth=%.0f | %s | stable=%.1fs | %s |
ACTIVE=%d | P=%d | rej=%.0f%%\n', ...
                t, raw_cm, final_smooth, status_str, stability, block_str,
system_active, person_count, reject_pct);
        end

        drawnow limitrate;
        pause(0.006);
    end

catch ME
    if ~strcmp(ME.identifier, 'MATLAB:cancel')
        rethrow(ME);
    end
end

clear ultrasonicObj arduinoObj;
fprintf('\n☑ FINAL COUNT: %d people (OPEN DOOR ONLY, ARMED VIA JOYSTICK)\n',
person_count);
```