MSD Assignment – II

Assignment Drive link: https://drive.google.com/drive/folders/1mEj16NZq5kF1QJ2dV5GkceqfvOpFxjEO?usp=sharing

Indira C

2023102070

Part 1

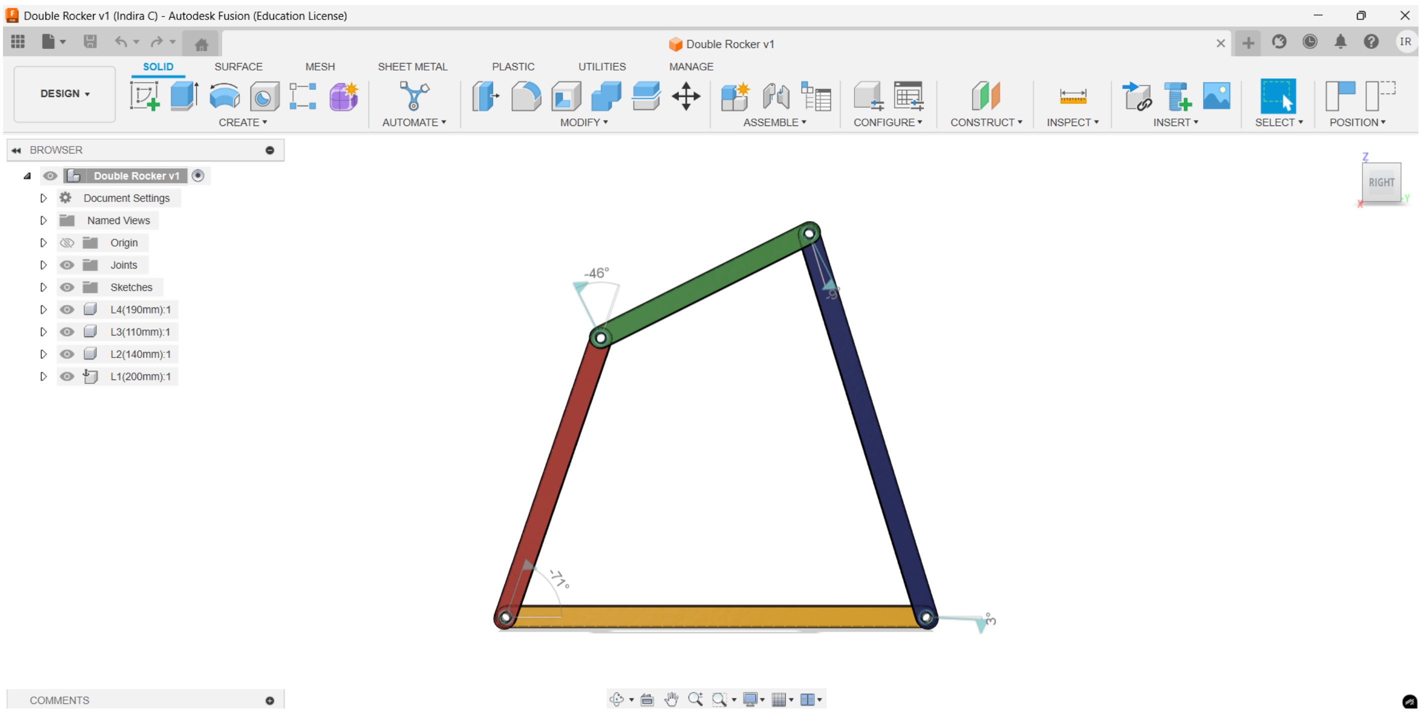
Double Rocker:

The double rocker four-bar mechanism is a type of four-bar linkage where both the input and output links (rockers) oscillate instead of completing full rotations. It consists of:

1. Frame (fixed link) – The stationary base.
2. Input rocker – The driving link that oscillates.
3. Coupler – Connects the two rockers.
4. Output rocker – The driven link that also oscillates.

Key Characteristics:

- Occurs when the sum of the shortest and longest links is greater than the sum of the other two (Grashof's condition).
- Longest link is the grounded one and the shortest link is the coupler.
- No complete rotation for any moving link.
- Used in valve gears, pumping systems, and robotic mechanisms for controlled oscillatory motion.



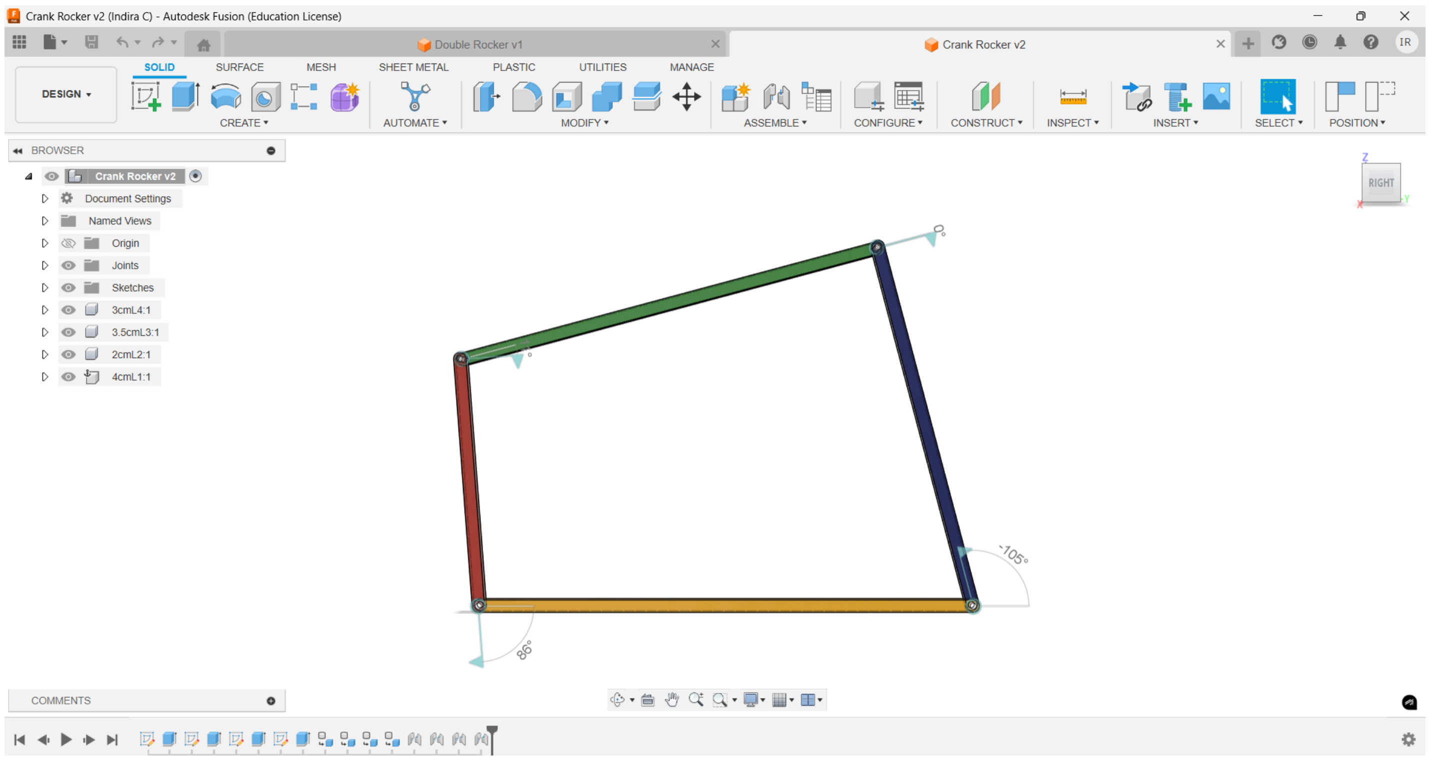
Crank Rocker:

The crank-rocker four-bar mechanism is a type of four-bar linkage where one link (crank) completes a full rotation while the other (rocker) oscillates. It consists of:

1. Frame (fixed link) – The stationary base.
2. Crank (input link) – Rotates continuously.
3. Coupler – Connects the crank and rocker.
4. Rocker (output link) – Oscillates back and forth.

Key Characteristics:

- Follows Grashof's condition:
- The shortest link (crank) must be adjacent to the fixed link.
- Converts rotary motion into oscillatory motion.
- Commonly found in engines, shapers, and mechanical presses for controlled movement.



Part 2

Dataset Description

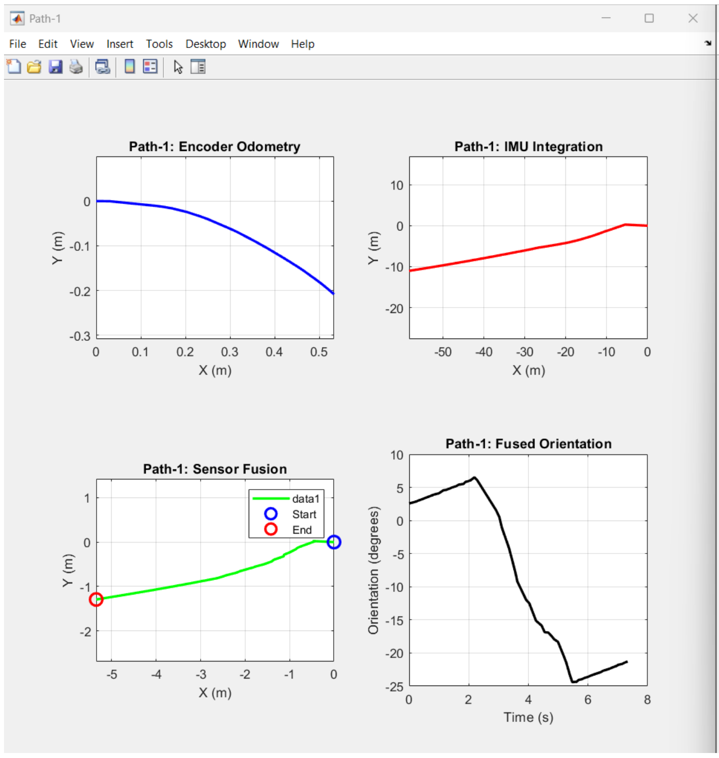
- Encoders: Measure track movement (0.095 mm per pulse).
- IMU: Provides acceleration and angular velocity.
- Magnetometer: Provides heading angle.

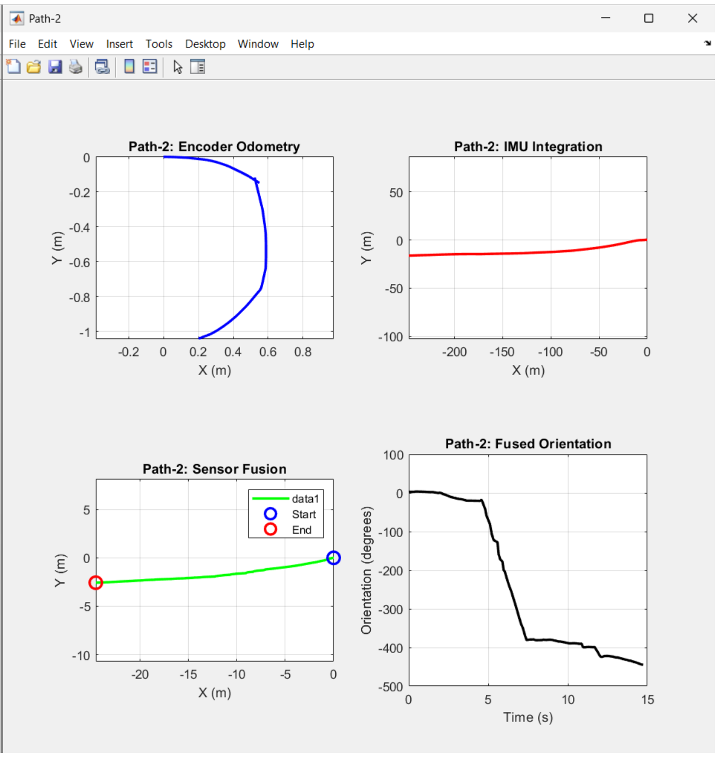
Methodology

1. Preprocessing: Load CSV, convert timestamps, extract relevant data.
2. Wheel Encoder Odometry: Estimate displacement and heading using differential track motion.
3. IMU-Based Estimation: Integrate accelerometer & gyroscope readings for velocity and position.
4. Sensor Fusion: Combine encoder, IMU, and magnetometer data using weighted averaging.
5. Visualization: Plot individual sensor trajectories and fused path.

Observations

- Encoders provide reliable distance but drift in orientation.
- IMU corrects rapid changes but accumulates noise over time.
- Magnetometer stabilizes heading but is affected by interference.
- Sensor fusion improves accuracy by compensating for individual weaknesses.





Matlab Code:

```
clear;
close all;
clc;
data1 = readtable('path_1_telemetry.csv');
data2 = readtable('path_2_telemetry.csv');
disp('Column names in path_1_telemetry.csv:');
disp(data1.Properties.VariableNames);
pulse_to_distance = 0.095e-3;
track_width = 0.105;
[x1_enc, y1_enc, theta1_enc, x1_imu, y1_imu, theta1_imu, x1_fused, y1_fused, theta1_fused] = ...
reconstruct_trajectory(data1, pulse_to_distance, track_width);
[x2_enc, y2_enc, theta2_enc, x2_imu, y2_imu, theta2_imu, x2_fused, y2_fused, theta2_fused] = ...
reconstruct_trajectory(data2, pulse_to_distance, track_width);
time1 = get_time(data1.timestamp);
time2 = get_time(data2.timestamp);
plot_trajectory(x1_enc, y1_enc, x1_imu, y1_imu, x1_fused, y1_fused, theta1_fused, time1, 'Path-
1');
plot_trajectory(x2_enc, y2_enc, x2_imu, y2_imu, x2_fused, y2_fused, theta2_fused, time2, 'Path-
2');
function time = get_time(timestamps)
timestamps = datetime(timestamps, 'InputFormat', 'yyyy-MM-dd''T''HH:mm:ss.SSSSSS');
time = seconds(timestamps - timestamps(1));
end
function [x_enc, y_enc, theta_enc, x_imu, y_imu, theta_imu, x_fused, y_fused, theta_fused] = ...
reconstruct_trajectory(data, pulse_to_distance, track_width)
left_counts = data.left_encoder_count;
right_counts = data.right_encoder_count;
accel_x = data.accel_x;
accel_y = data.accel_y;
gyro_z = data.gyro_z;
heading = deg2rad(data.heading);
timestamps = datetime(data.timestamp, 'InputFormat', 'yyyy-MM-dd''T''HH:mm:ss.SSSSSS');
dt = seconds(diff(timestamps));
dt = [dt(1); dt];
n = length(left_counts);
x_enc = zeros(n, 1); y_enc = zeros(n, 1); theta_enc = zeros(n, 1);
x_imu = zeros(n, 1); y_imu = zeros(n, 1); theta_imu = zeros(n, 1);
x_fused = zeros(n, 1); y_fused = zeros(n, 1); theta_fused = zeros(n, 1);
vx_imu = 0; vy_imu = 0;
for i = 1:n-1
d_left = (left_counts(i+1) - left_counts(i)) * pulse_to_distance;
d_right = (right_counts(i+1) - right_counts(i)) * pulse_to_distance;
d_center = (d_left + d_right) / 2;
d_theta = (d_right - d_left) / track_width;
theta_enc(i+1) = theta_enc(i) + d_theta;
x_enc(i+1) = x_enc(i) + d_center * cos(theta_enc(i+1));
y_enc(i+1) = y_enc(i) + d_center * sin(theta_enc(i+1));
end
for i = 1:n-1
```

```
theta_imu(i+1) = theta_imu(i) + gyro_z(i) * dt(i);
vx_imu = vx_imu + accel_x(i) * dt(i);
vy_imu = vy_imu + accel_y(i) * dt(i);
x_imu(i+1) = x_imu(i) + vx_imu * dt(i);
y_imu(i+1) = y_imu(i) + vy_imu * dt(i);
end
alpha = 0.9;
beta = 0.6;
for i = 1:n
theta_fused(i) = alpha * theta_enc(i) + (1-alpha) * (beta * theta_imu(i) + (1-beta) * heading(i));
x_fused(i) = alpha * x_enc(i) + (1-alpha) * x_imu(i);
y_fused(i) = alpha * y_enc(i) + (1-alpha) * y_imu(i);
end
end
function plot_trajectory(x_enc, y_enc, x_imu, y_imu, x_fused, y_fused, theta_fused, time,
title_str)
figure('Name', title_str, 'NumberTitle', 'off', 'Position', [100, 100, 800, 600]);
subplot(2, 2, 1);
plot(x_enc, y_enc, 'b-', 'LineWidth', 2);
title([title_str ': Encoder Odometry']); xlabel('X (m)'); ylabel('Y (m)'); grid on; axis equal;
subplot(2, 2, 2);
plot(x_imu, y_imu, 'r-', 'LineWidth', 2);
title([title_str ': IMU Integration']); xlabel('X (m)'); ylabel('Y (m)'); grid on; axis equal;
subplot(2, 2, 3);
plot(x_fused, y_fused, 'g-', 'LineWidth', 2); hold on;
plot(x_fused(1), y_fused(1), 'bo', 'MarkerSize', 10, 'LineWidth', 2, 'DisplayName', 'Start');
plot(x_fused(end), y_fused(end), 'ro', 'MarkerSize', 10, 'LineWidth', 2, 'DisplayName', 'End');
title([title_str ': Sensor Fusion']); xlabel('X (m)'); ylabel('Y (m)'); legend; grid on; axis
equal;
subplot(2, 2, 4);
plot(time(1:end-1), rad2deg(theta_fused(1:end-1)), 'k-', 'LineWidth', 2);
title([title_str ': Fused Orientation']); xlabel('Time (s)'); ylabel('Orientation (degrees)');
grid on;
end
```

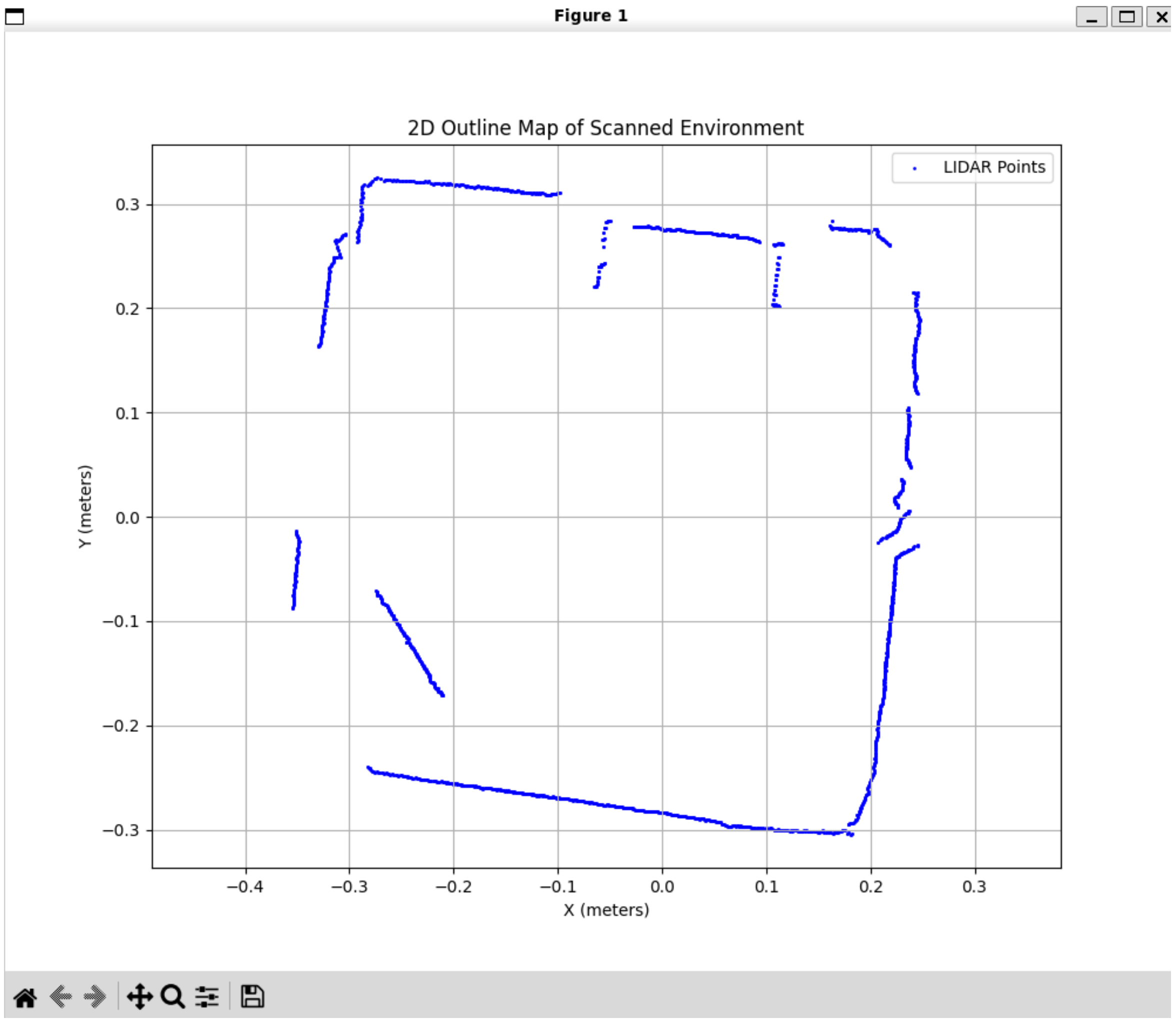
Part 3

LIDAR

Methodology

- Load Data: Read the .pcd (point cloud) and .pkl files.
- Combine Points: Merge both datasets.
- Project to 2D: Ignore z-coordinates to get a 2D view.
- Plot & Save: Visualize the outline using Matplotlib.

Results



Observations on Black Object Behavior

- Edge Detection: Black objects may lack clear boundaries, making detection challenging.
- Noise Sensitivity: Black objects with low contrast blend into the background.
- LIDAR Reflection: Dark surfaces absorb laser pulses, leading to missing points in scans.

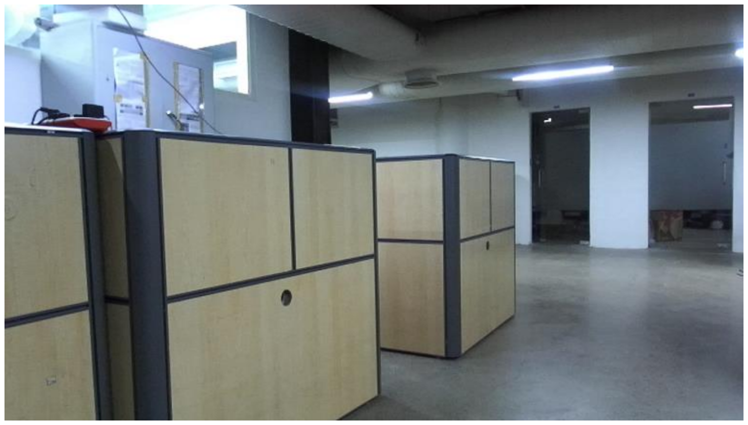
Python Code:

```
import numpy as np
import open3d as o3d
import pickle
```

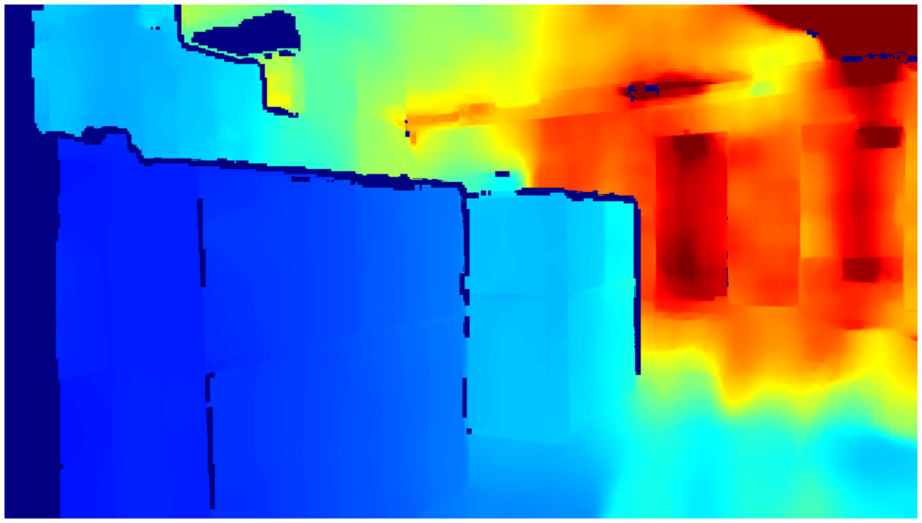
```python
import matplotlib.pyplot as plt

pcd = o3d.io.read_point_cloud("scan_team_common.pcd")
points = np.asarray(pcd.points)

with open("scan_vector_team_common.pkl", "rb") as f:
    pkl_data = pickle.load(f)

if isinstance(pkl_data, np.ndarray):
    points_pkl = pkl_data
else:
    points_pkl = np.array(pkl_data)

all_points = np.vstack((points, points_pkl))

x = all_points[:, 0]
y = all_points[:, 1]

plt.figure(figsize=(10, 8))
plt.scatter(x, y, s=1, c='blue', label='LIDAR Points')
plt.title("2D Outline Map of Scanned Environment")
plt.xlabel("X (meters)")
plt.ylabel("Y (meters)")
plt.grid(True)
plt.legend()
plt.axis('equal')
plt.savefig("2d_outline_map.png")
plt.show()
```

ZED

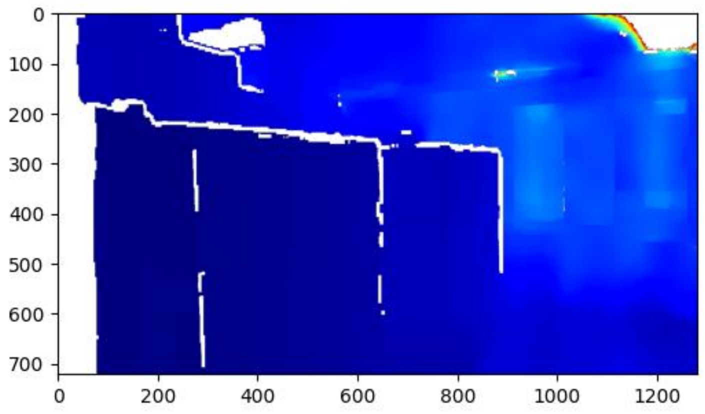
Results:

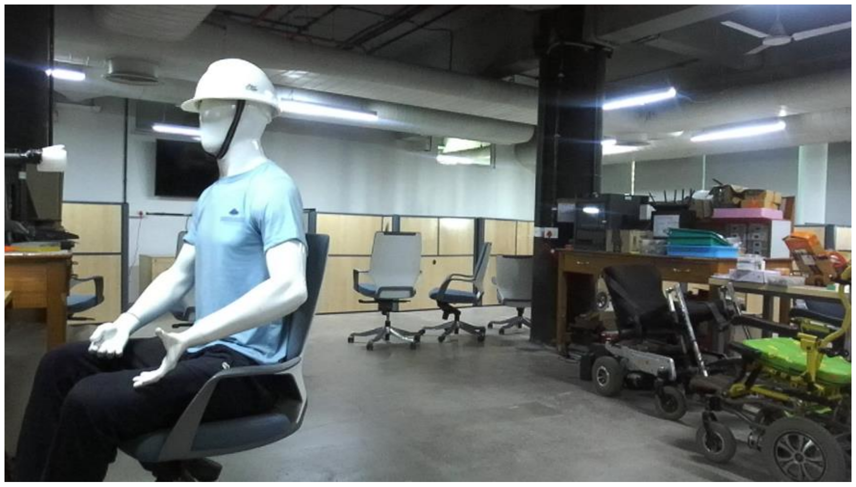


Lab depth heatmap output

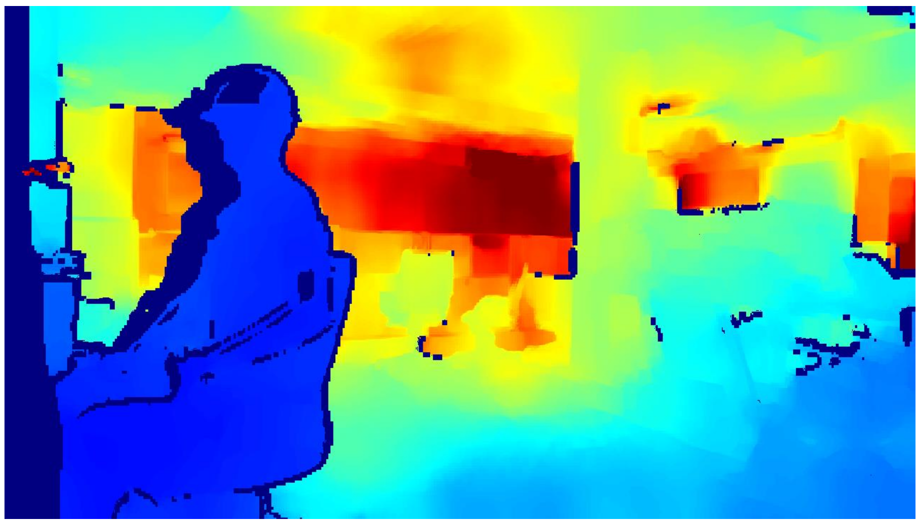


Lab depth matrix output

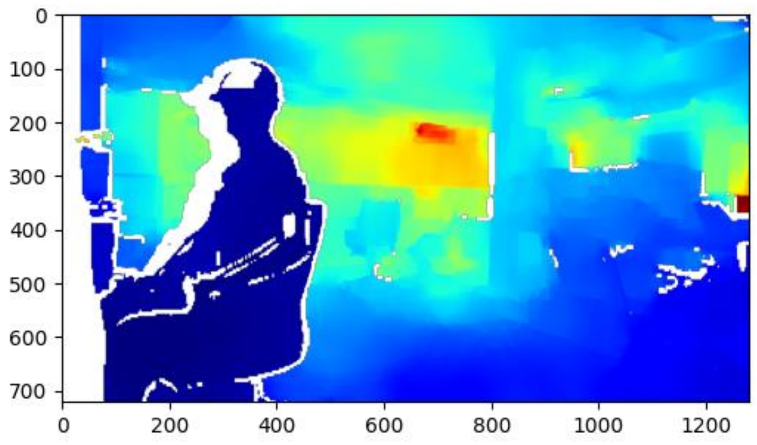




Lab2 depth heatmap output



Lab2 depth matrix output



Reconstruction Methodology:

- Method 1 (Depth Matrix): Used raw depth matrix (.npy) with formulas X=fx (u−cx)· Z ,Y=fy (v−cy)· Z , Z=depth value, and color from image to create PLY point cloud.
- Method 2 (Depth Heatmap): Converted heatmap to grayscale, mapped back to depth using depth=255gray ×(max_depth−min_depth)+min_depth (min=0.1m, max=10.0m), then computed 3D coordinates and PLY file.

Challenges Faced:

- Depth heatmap conversion caused accuracy loss due to quantization.

Comparison of Outcomes:

- Method 1: More accurate, detailed point cloud with precise shapes.
- Method 2: Noisier, less accurate due to heatmap approximation, but recognizable shape.

Method 1 preferred for precision; Method 2 viable with limitations.

Python Code:

```python
import numpy as np
import cv2
import open3d as o3d
import matplotlib.pyplot as plt

fx = 957.08
fy = 957.08
cx = 649.15
cy = 370.98

min_depth = 0.1
max_depth = 10.

def heatmap_to_depth(heatmap):
    gray = cv2.cvtColor(heatmap, cv2.COLOR_BGR2GRAY)
    depth = (gray / 255.0) * (max_depth - min_depth) + min_depth
    return depth

def reconstruct_point_cloud(depth, color_image, fx, fy, cx, cy):
    height, width = depth.shape
    points = []
    colors = []

    color_image = cv2.cvtColor(color_image, cv2.COLOR_BGR2RGB)

    for v in range(height):
        for u in range(width):
            Z = depth[v, u]
            if Z < min_depth or Z > max_depth:
                continue
            X = (u - cx) * Z / fx
            Y = (v - cy) * Z / fy
            points.append([X, Y, Z])
            colors.append(color_image[v, u] / 255.0)
    pcd = o3d.geometry.PointCloud()
    pcd.points = o3d.utility.Vector3dVector(np.array(points))
    pcd.colors = o3d.utility.Vector3dVector(np.array(colors))
    return pcd

def visualize_depth(depth, output_heatmap_path, output_matrix_path):
    # Normalize depth for visualization (heatmap)
    depth_norm = (depth - min_depth) / (max_depth - min_depth) * 255
    depth_norm = np.clip(depth_norm, 0, 255).astype(np.uint8)
    depth_heatmap = cv2.applyColorMap(depth_norm, cv2.COLORMAP_JET)

    cv2.imwrite(output_heatmap_path, depth_heatmap)

    plt.figure(figsize=(6, 4))
    plt.imshow(depth, cmap='jet')
    plt.axis('on')
    plt.savefig(output_matrix_path)
    plt.close()

def process_dataset(color_path, heatmap_path, depth_matrix_path, output_ply_path,
output_heatmap_path, output_matrix_path):
    # 1. Load the data
    color_image = cv2.imread(color_path)
    depth_heatmap = cv2.imread(heatmap_path)
    depth_matrix = np.load(depth_matrix_path)

    # Method 1: Reconstruct point cloud using depth matrix
    pcd_matrix = reconstruct_point_cloud(depth_matrix, color_image, fx, fy, cx, cy)
    o3d.io.write_point_cloud(output_ply_path + "_method1.ply", pcd_matrix)

    # Method 2: Reconstruct point cloud using depth heatmap
    depth_from_heatmap = heatmap_to_depth(depth_heatmap)
    pcd_heatmap = reconstruct_point_cloud(depth_from_heatmap, color_image, fx, fy, cx, cy)
    o3d.io.write_point_cloud(output_ply_path + "_method2.ply", pcd_heatmap)

    visualize_depth(depth_matrix, output_heatmap_path, output_matrix_path)

    plt.figure(figsize=(15, 5))

    plt.subplot(1, 3, 1)
    plt.title("Color Image")
    plt.imshow(cv2.cvtColor(color_image, cv2.COLOR_BGR2RGB))
    plt.axis('on')

    plt.subplot(1, 3, 2)
    plt.title("Depth Heatmap")
    plt.imshow(cv2.cvtColor(cv2.imread(output_heatmap_path), cv2.COLOR_BGR2RGB))
    plt.axis('on')

    plt.subplot(1, 3, 3)
    plt.title("Depth Matrix")
    plt.imshow(depth_matrix, cmap='jet')
    plt.axis('on')

    plt.tight_layout()
    plt.show()

if __name__ == "__main__":
    # Dataset 1: lab
    process_dataset(
        color_path="lab_color.png",
        heatmap_path="lab_depth_heatmap.png",
        depth_matrix_path="lab_depth_matrix.npy",
        output_ply_path="lab_point_cloud",
        output_heatmap_path="lab_depth_heatmap_output.png",
        output_matrix_path="lab_depth_matrix_output.png"
    )

    # Dataset 2: lab2
    process_dataset(
        color_path="lab2_color.png",
        heatmap_path="lab2_depth_heatmap.png",
        depth_matrix_path="lab2_depth_matrix.npy",
        output_ply_path="lab2_point_cloud",
        output_heatmap_path="lab2_depth_heatmap_output.png",
        output_matrix_path="lab2_depth_matrix_output.png"
    )
```

TOF

Observations

- Distance Variation: The plot shows step changes as objects are removed.

- Noise Reduction: A moving average filter smooths the raw data.
- Sensor Accuracy: Distance readings align with expected values but show minor fluctuations.
- Response Time: The sensor quickly updates distances, capturing object removal moments clearly.

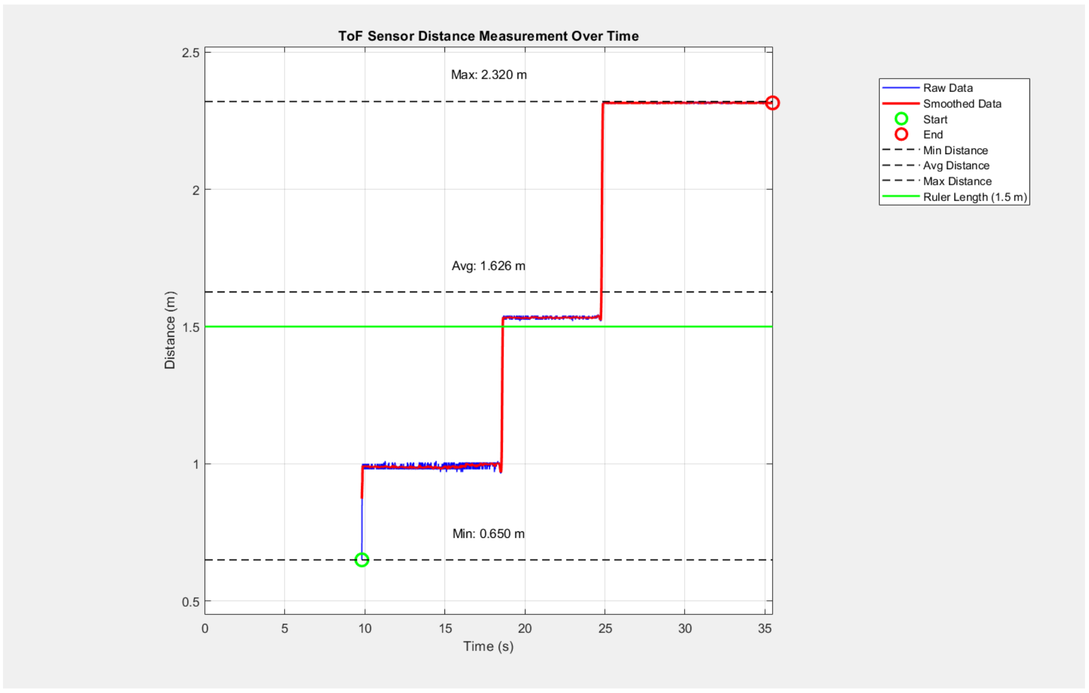
Accuracy Analysis

```
Column names in the table:
    {'Timestamp'}    {'Distance(mm)'}

Total Duration: 35.48 seconds
Minimum Distance: 0.650 meters
Maximum Distance: 2.320 meters
Average Distance: 1.626 meters
```

- Error Sources: Minor deviations due to sensor noise and environmental factors.

Plot Overview

- Raw vs. Smoothed Data: Blue (raw), Red (smoothed).
- Key Events: Start (Green), End (Red).
- Reference Line: 1.5m ruler for comparison.



Code Summary

- Reads and processes ToF sensor data.
- Converts timestamps to relative time.

- Filters valid (non-zero) distances.
- Plots raw and smoothed data.
- Marks key events and reference values.

Matlab Code:

```
clear all;
close all;
clc;
opts = detectImportOptions('tof_benchmark.csv');
opts.VariableNamingRule = 'preserve';
data = readtable('tof_benchmark.csv', opts);
disp('Column names in the table:');
disp(data.Properties.VariableNames);
timestamps = data.Timestamp;
distances_mm = data.('Distance(mm)');
time_start = datetime(timestamps(1), 'InputFormat', 'yyyy-MM-dd HH:mm:ss.SSSSSS');
time_seconds = seconds(datetime(timestamps, 'InputFormat', 'yyyy-MM-dd HH:mm:ss.SSSSSS') -
time_start);
distances_m = distances_mm / 1000;
valid_indices = distances_m > 0;
time_valid = time_seconds(valid_indices);
distances_valid = distances_m(valid_indices);
window_size = 10;
distances_smoothed = movmean(distances_valid, window_size);
total_time = time_seconds(end);
max_distance = max(distances_valid);
min_distance = min(distances_valid);
avg_distance = mean(distances_valid);
figure('Name', 'ToF Sensor Distance Over Time', 'NumberTitle', 'off', 'Position', [100, 100, 1000,
600]);
plot(time_valid, distances_valid, 'b-', 'LineWidth', 1, 'DisplayName', 'Raw Data');
hold on;
plot(time_valid, distances_smoothed, 'r-', 'LineWidth', 2, 'DisplayName', 'Smoothed Data');
plot(time_valid(1), distances_valid(1), 'go', 'MarkerSize', 10, 'LineWidth', 2, 'DisplayName',
'Start');
plot(time_valid(end), distances_valid(end), 'ro', 'MarkerSize', 10, 'LineWidth', 2, 'DisplayName',
'End');
plot([0 total_time], [min_distance min_distance], 'k--', 'LineWidth', 1, 'DisplayName', 'Min
Distance');
plot([0 total_time], [avg_distance avg_distance], 'k--', 'LineWidth', 1, 'DisplayName', 'Avg
Distance');
plot([0 total_time], [max_distance max_distance], 'k--', 'LineWidth', 1, 'DisplayName', 'Max
Distance');
% Display text annotations for min, avg, and max distances
text(total_time/2, min_distance + 0.1, sprintf('Min: %.3f m', min_distance),
'HorizontalAlignment', 'center');
text(total_time/2, avg_distance + 0.1, sprintf('Avg: %.3f m', avg_distance),
'HorizontalAlignment', 'center');
```

```
text(total_time/2, max_distance + 0.1, sprintf('Max: %.3f m', max_distance),
'HorizontalAlignment', 'center');
plot([0 total_time], [1.5 1.5], 'g-', 'LineWidth', 1.5, 'DisplayName', 'Ruler Length (1.5 m)');
title('ToF Sensor Distance Measurement Over Time');
xlabel('Time (s)');
ylabel('Distance (m)');
legend('Location', 'best');
grid on;
axis([0 total_time min_distance - 0.2 max_distance + 0.2]);
axis square;
fprintf('Total Duration: %.2f seconds\n', total_time);
fprintf('Minimum Distance: %.3f meters\n', min_distance);
fprintf('Maximum Distance: %.3f meters\n', max_distance);
fprintf('Average Distance: %.3f meters\n', avg_distance);
```