RANSAC-Based Plane Detection: Improving Precision in 3D Mapping

Ana Pinto[1,2], João Monteiro[1,2], Tomás Rodrigues[1,2]

[1] FCUP - Faculty of Sciences, University of Porto
[2] FEUP - Faculty of Engineering, University of Porto
up202105085@up.pt, up202108347@up.pt, up202107937@up.pt

Abstract. Mobile robots are used in several applications and features like the identification of 3D primitives is fundamental in the mapping process. This paper presents a method to identify accurately the pose and parameters of 3D primitives in a map from a LiDAR sensor. In this sense, the main goal is to get a mobile robot with basic capabilities for navigating a rectangular map filled with 3D uniformly distributed obstacles. This methodology uses the LiDAR sensor to acquire a point cloud of the environment. Then that is processed in the Random Sample Consensus (RANSAC) algorithm since it is robust against outliers and can fit multiple models. The robot has a Webots simulation environment and is simulated using an e-puck model featuring the LiDAR sensor for motion while using predefined paths. This approach enables the robot to find properties that define the centroid and orientation of the detected primitive. Experiments in the simulated environment showed high reliability in determining the pose and parameters of different 3D primitives and possible incorporation into fully automated systems for mapping and navigation of mobile robots.

Keywords: mapping, simulation, obstacle detection, pose estimation, robot vision, RANSAC

1 Introduction

Mapping and identifying 3D primitives are crucial in mobile robotics, especially in industrial applications such as manufacturing and agriculture. These tasks, which include distinguishing the pose, orientation, and size of objects in a specific space, are fundamental functions for mobile robots. Hence, reducing the failure rate in these processes makes this study significant.

Automating the detection of 3D primitives can lead to significant efficiency gains across various applications. However, this task is complicated by the diversity of shapes and the potential for occlusion and overlapping spatial scenes. The goal of this project is to design a mobile robot equipped with a LiDAR sensor that can accurately detect the pose and parameters of 3D primitives, including planes, polyhedra and any solid whose phases are all planar, in a mapped environment.

The proposed method leverages the Random Sample Consensus (RANSAC) algorithm, chosen for its resistance to outliers and ability to handle multiple shapes within the same dataset, making it suitable for detecting various 3D forms. The mobile robot navigates using the Webots simulation platform with the e-puck robot model, carrying a LiDAR sensor. The robot's movement follows predefined paths to ensure accurate and consistent data collection.

The remainder of this paper is organized as follows: Section 2 reviews the literature on pose estimation for 3D primitive mapping. Section 3 describes the proposed method, detailing the experimental setup, the algorithms. Additionally, it reports the performance metrics. Section 4 concludes with a discussion of the findings and suggestions for future research directions.

2 Literature Review

Mobile robots are increasingly being utilized in various applications, with mapping and navigation being fundamental functionalities. One crucial aspect of mapping is the identification of 3D primitives within the environment. The use of LiDAR sensors for this purpose has gained significant attention due to their ability to provide detailed point cloud data of the surroundings. By processing this data using robust algorithms like Random Sample Consensus (RANSAC), mobile robots can accurately identify and locate 3D primitives such as spheres, cubes, and other shapes within the mapped area [1].

The integration of computer vision techniques with mobile robots has enabled advancements in mapping and navigation capabilities. Vision-based global localization and mapping systems have been developed to create 3D maps of environments using visual landmarks, enhancing the autonomy of mobile robots [2]. Additionally, information-theoretic planning approaches have been proposed to facilitate the autonomous construction of dense 3D maps efficiently, further improving the mapping capabilities of mobile robots [3].

Efficient algorithms like RANSAC have been pivotal in processing point cloud data for shape detection and segmentation tasks. RANSAC has been widely adopted in the computer vision community due to its robustness against outliers, making it suitable for tasks like primitive shape fitting in point clouds [4]. Furthermore, the development of hybrid approaches like H-RANSAC, which combine 2D and 3D data for point cloud segmentation, showcases the continuous efforts to enhance the accuracy and efficiency of mapping processes.

The extraction of geometric primitives from point clouds has been a focus of research, aiming to bridge the gap between low-level 3D data and high-level structural information. Methods such as supervised fitting of geometric primitives to point clouds have been developed to provide detailed information on the shapes present in the environment, facilitating better mapping and navigation for mobile robots [5].

In conclusion, the utilization of computer vision techniques, robust algorithms like RANSAC, and advancements in point cloud processing have significantly enhanced the mapping and navigation capabilities of mobile robots. By

accurately identifying and locating 3D primitives within the environment, mobile robots can navigate complex terrains autonomously, paving the way for the integration of fully automated systems for mapping and navigation tasks.

3 Algorithm and experimental validation

3.1 Experimental Setup

In our experimental setup using Webots, we configured an e-puck robot with specific sensor parameters to achieve precise data acquisition.

A LIDAR sensor was chosen due to its ability to provide detailed depth information with default settings, except for two key adjustments: the number of layers is set to 8, and the vertical field of view is 0.4. This configuration allows for a balanced trade-off between vertical resolution and coverage, ensuring accurate scanning of the environment.

The e-puck robot, named EPUCK, operates under default parameters with the exception of the supervisor mode being enabled. This setting allows for enhanced monitoring and control during the simulation, facilitating better data collection and analysis.

Further into the experiments, we changed the number of noise levels in the LIDAR and adjusted the number of beams emitted. These changes were implemented to assess the robustness of the code and to determine the extent to which the methods can cope with variations in the quality of the sensor input.

3.2 Random Map Generator

To automate the generation of multiple maps, a script was developed that randomly places geometric primitives from a predefined set of shapes. Each primitive is assigned a random location, size, and rotation. The sides of each primitive are constructed using the Webots object **Wall** [6].

The implementation is divided into four main steps: randomly generate the parameters, apply the rotation, verify legality (no overlaps or out-of-bound placements), and place the primitive. Algorithm 1 presents an overview of the solution. Figure 1 shows an example of the rotation and bounding box.

Algorithm 1 Generating Map Matrix with Random Primitives

Input: None

Output: *matrix* - a 3D array representing the final map, *mask* - a 2D array indicating which cells are occupied and which are not, *shapes* - a list detailing the placed shapes, including their type, vertex coordinates, rotation angle, and center.

size ← the size of each side of the map
matrix ← a 3D array of zeros with dimensions $size \times size \times 3$
mask ← a 2D array of zeros with dimensions $size \times size$
shapes ← list that will store the primitives information
num_shapes ← a random integer between a predefined interval
shape_functions ← a list containing the shape generating functions DrawTriangle,DrawSquare,DrawRectangle,DrawPentagon
shape_sizes ← randomly sets a size to each primitive
for *i* from 1 to *num_shapes* **do**
    shape_fn ← the random primitive chosen
    angle_range ← rotation angle range for the respective primitive
    angle ← random angle value
    shapes ← PlaceShape(*matrix*, *mask*, *shapes*, *shape_fn*, *shape_sizes*[*i*], *angle*)
return *matrix*, *mask*, *shapes*

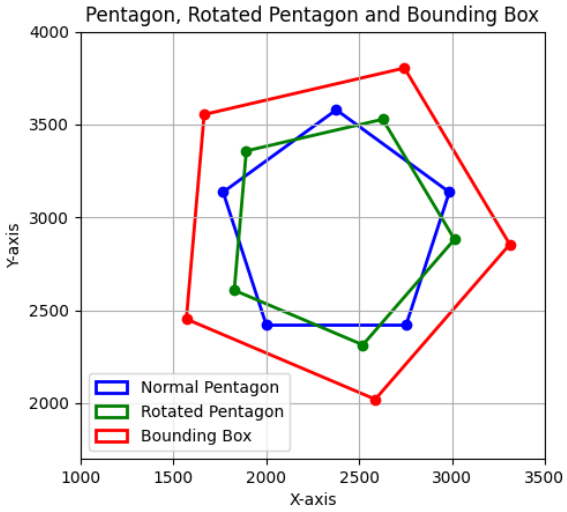


Fig. 1. Visual representation of the rotation and bounding box

In the process of placing a primitive shape, it's essential to verify that it fits within the map boundaries and doesn't overlap with existing shapes. This

validation occurs within the PlaceShape method. It checks that all vertices of the shape are within the map and maintain a minimum distance from the walls. Additionally, a bounding box is created around the shape to detect overlaps with existing figures on the map. These measures ensure that new shapes integrate seamlessly into the map without causing conflicts or extending beyond its limits.

To apply rotation, each point (p_x, p_y) of every generated primitive is rotated around a centroid (o_x, o_y) (as defined in Equation 1) by an angle θ. This angle is converted to radians. The resulting points are denoted as (q_x, q_y) (as defined in Equation 2).

$$G_{(x,y)} = \frac{1}{n}\sum_{i=1}^{n}(x_i, y_i) \tag{1}$$

$$q_{(x,y)} = o_{(x,y)} + \cos(\theta_{\text{rad}}) \cdot (p_x - o_x) - \sin(\theta_{\text{rad}}) \cdot (p_y - o_y) \tag{2}$$

After the rotation and ensuring the primitive's containment within the map, it becomes imperative to verify the absence of overlaps upon placement. To achieve this, a bounding box is generated to maintain separation between primitives. This process involves calculating the direction vector from the centroid (G_x, G_y) for each vertex (x_i, y_i)3. Subsequently, utilizing the distance d_i4 from each vertex to the centroid, a scaling factor s_i5 is determined to extend the distance by a predefined pixel value p. Utilizing the scaled direction vectors6, the new position (x'_i, y'_i)7 of each vertex can be computed. Next, it is verified whether the bounding box overlaps with any existing primitives. While an overlap is detected, a new pair of coordinates is generated, and the process is repeated, keeping the angle and primitive type unchanged.

$$\Delta(x_i, y_i) = (x_i, y_i) - G_{(x_i, y_i)} \tag{3}$$

$$d_i = \sqrt{(\Delta x_i)^2 + (\Delta y_i)^2} \tag{4}$$

$$s_i = \frac{d_i + p}{d_i} \tag{5}$$

$$\Delta(x'_i, y'_i) = \Delta(x'_i, y'_i) \cdot s_i \tag{6}$$

$$(x'_i, y'_i) = G_{(x,y)} + \Delta(x'_i, y'_i) \tag{7}$$

3.3 Point Cloud Processing

Point cloud processing is a fundamental aspect of various applications in robotics, computer vision, and 3D modeling. A point cloud is a collection of data points defined in a three-dimensional coordinate system. These points represent the external surface of an object or scene, captured by sensors such as LIDAR or stereo cameras.

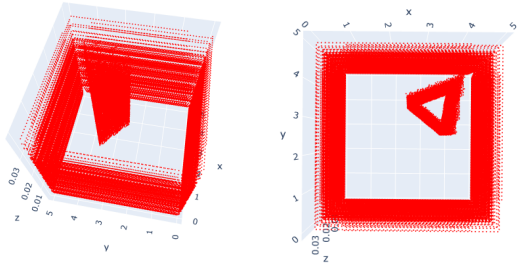
In the context of robotics, point cloud data is invaluable for navigation, object recognition, and environment mapping. It enables robots to perceive their surroundings in 3D, making it possible to perform tasks such as obstacle avoidance, path planning, and interaction with objects.

To obtain the point cloud data from our maps, we implemented the following code:

Algorithm 2 Point Cloud Processing Algorithm

Input: Devices (LIDAR, GPS, Compass), Movement Coordinates
Output: Processed Point Cloud Data

Initialize supervisor and devices
Enable LIDAR, GPS, and Compass with timestep
Retrieve initial GPS and compass readings for robot position and orientation
Warp robot to initial position
Create movement coordinates (x, y)
Load map mask from file
for each *new_position* in *moves* **do**
    if *new_position* is valid based on mask **then**
        Step the simulation
        Warp robot to *new_position*
        Retrieve updated GPS and compass readings
        Get point cloud data from LIDAR
        Transform and filter point cloud data
        Append transformed data to result list
Save the collected point cloud data to a file
Filter the saved data based on z-coordinate threshold
Create and visualize the point cloud using the filtered data



(a) Point Cloud 3D (b) Point Cloud Top View

Fig. 2. Visual representation of a point cloud

Plane Fitting It should be noted that, in this implementation, the robot does not scan the point cloud when it is inside any solid.

In this phase, the point cloud data needs to be filtered to identify planes. This is crucial because planes are fundamental geometric structures that define boundaries and surfaces of objects in the environment.

To identify planes, we use RANSAC (Random Sample Consensus) to iteratively fit plane models to random subsets of data, selecting the model with the most inliers. DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is then used to refine these inliers, grouping points that are close together and identifying the largest cluster as the best-fitting plane.

Here's a detailed overview of the plane fitting process using RANSAC and DBSCAN:

Algorithm 3 Plane Intersection Algorithm with RANSAC

Input: data_arr (point cloud data), min_points, threshold, max_iteration
Output: planes, intersection_points, intersection_edges, inliers_all

Initialize lists for planes and inliers_all
Set outliers_before to data_arr
for each orientation in ["vertical"] **do**
    while there are inliers_plane **do**
        Create and fit a Plane object 4 with outliers_before
        Update inliers_plane and outliers_before
        if there are enough inliers **then**
            Add plane and inliers to planes
            Add inliers_plane to inliers_all
Set z to the maximum z-coordinate in data_arr
for each pair of planes (plane_a, plane_b) **do**
    if plane_a and plane_b are different **then**
        Calculate intersection points using plane_intersect
        if intersection points are close to inliers of both planes **then**
            Add intersection point to intersection_points and (a, b) to intersection_edges
for each plane in planes **do**
    for each intersection point in intersection_points **do**
        if point is on the plane **then**
            Add point to edge_points
    if there are 2 edge_points **then**
        Add edge_points to edges
Return planes, intersection_points, intersection_edges, inliers_all

Algorithm 4 Plane Fitting Process Using DBSCAN

Input: pts - 3D point cloud, thresh - distance threshold, minPoints - minimum number of inlier points, maxIteration - maximum number of iterations, orientation - plane orientation **Output**: equation - plane parameters, inliers - inlier points

if random_seed $\neq$ None **then**
    Set random seeds with random_seed
Initialize DBSCAN with eps = 0.1 and min_samples = 1
$n\_points \leftarrow$ number of points in pts
$best\_eq \leftarrow []$
$best\_inliers \leftarrow []$
for *it* from 0 to maxIteration **do**
    Select random points from pts according to orientation
    Compute vectors A, B, and normal vector C
    if $\|C\| = 0$ **then**
        Continue to the next iteration
    Normalize C
    Compute *plane_eq* and distances to the plane
    Select inliers within thresh
    if $len(inliers) > len(best\_inliers)$ and $len(inliers) >$ minPoints **then**
        Update *best_eq* and *best_inliers*
$self.inliers \leftarrow best\_inliers$
$self.equation \leftarrow best\_eq$
$inliers\_plane \leftarrow pts[self.inliers, :]$
$inliers\_indices \leftarrow self.inliers$
if $len(inliers\_plane) > 0$ **then**
    Apply DBSCAN on *inliers_plane*
    Find the largest cluster
    inliers_plane_indices $\leftarrow$ indices of the largest cluster
else
    inliers_plane_indices $\leftarrow []$
$self.inliers\_indices \leftarrow inliers\_plane\_indices$
return $self.equation$, $self.inliers\_indices$

In the process of adjusting planes, several mathematical equations are essential to determine the plane that best fits a 3D point cloud. To simplify the code, the z coordinate was taken as the z of the point in the point cloud with the highest z value. The meaning of each equation used in this context is explained below.

Vectors between Selected Points: To define a plane, we first need to identify two vectors that lie on the plane. These vectors are calculated from three selected points pt_1, pt_2, and pt_3 in the point cloud:

$$\vec{A} = \mathrm{pt}_2 - \mathrm{pt}_1$$

$$\vec{B} = \mathrm{pt}_3 - \mathrm{pt}_1$$

These vectors represent directions along the plane and are fundamental in determining the plane's orientation.

Normal Vector to the Plane: The normal vector $\vec{C}$ is crucial as it is perpendicular to the plane. It is found using the cross product of vectors $\vec{A}$ and $\vec{B}$:

$$\vec{C} = \vec{A} \times \vec{B}$$

The normal vector provides the coefficients for the plane equation and helps in calculating distances from other points to the plane.

Normalization of the Normal Vector: For numerical stability and standardization, the normal vector $\vec{C}$ is normalized:

$$\vec{C} = \frac{\vec{C}}{\|\vec{C}\|}$$

Normalization ensures that the magnitude of $\vec{C}$ is 1, simplifying subsequent calculations and making the plane equation more robust.

Plane Equation: The equation of the plane can be derived using the normal vector and a point on the plane. The constant k is calculated as follows:

$$k = -(\vec{C} \cdot \mathrm{pt}_1)$$

Thus, the plane equation in the form $Ax + By + Cz + D = 0$ is given by:

$$\text{plane\_eq} = [C_x, C_y, C_z, k]$$

This equation represents the plane in 3D space and is used to evaluate how well other points fit this plane.

Distance of Points to the Plane: To determine if a point (x_i, y_i, z_i) is an inlier (i.e., it lies close to the plane), we calculate the perpendicular distance d_i from the point to the plane:

$$d_i = \frac{C_x \cdot x_i + C_y \cdot y_i + C_z \cdot z_i + k}{\sqrt{C_x^2 + C_y^2 + C_z^2}}$$

This distance helps in identifying points that are within a certain threshold from the plane, indicating they are part of the plane.

Selection of Inliers: Inliers are points whose distance to the plane is within a specified threshold. These points are selected using the following condition:

$$\text{pt\_id\_inliers} = \{i \mid |d_i| \leq \text{thresh}\}$$

By selecting inliers, we can refine our plane model, ensuring it accurately represents the major surface within the point cloud while excluding outliers.

Intersection of Two Planes:

To find the line of intersection between two planes, was solved the system of equations for the two plane equations. Suppose we have two planes P_1 and P_2 with the equations:

$$a_1x + b_1y + c_1z + d_1 = 0$$

$$a_2x + b_2y + c_2z + d_2 = 0$$

The line of intersection of these planes can be found by solving this system of equations. Typically, this involves finding a common point on both planes and the direction vector of the line of intersection. The intersection of two planes generally results in a line, which can be represented by two points (x_1, y_1, z_1) and (x_2, y_2, z_2).

Euclidean Distance Calculation:

To determine if the intersection points are close enough, was used the Euclidean distance formula:

$$\text{distance} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

This formula is used to check if the intersection points are within a specific distance, which helps to identify if two planes intersect effectively.

After executing the RANSAC function, several additional steps are carried out to identify and process geometric shapes from the plane intersection data.

Identification of Cycles (Shapes): The Find Cycles function 5 is used to identify cycles in the intersection edge data. Intersecting edges represent lines where planes intersect. Finding cycles on these edges makes it possible to identify closed geometric shapes, such as polygons, which are formed by the intersection points of the planes.

Algorithm 5 Cycle Finding Algorithm

Input: edges
Output: cycles

Create an empty graph G
Add edges to G from edges list
Initialize an empty list *cycles*
function DFS_CYCLE(start, current, visited, stack, cycles)
    $visited[current] \leftarrow True$
    $stack.append(current)$
    for all neighbor in neighbors of *current* in G **do**
        if not $visited[neighbor]$ **then**
            DFS_CYCLE(start, neighbor, visited, stack, cycles)
        else if neighbor == start and length of $stack > 2$ **then**
            $cycle \leftarrow stack + [start]$
            $cycles.append(cycle)$
    $stack.pop()$
    $visited[current] \leftarrow False$
for all node in nodes of G **do**
    Initialize *visited* as a dictionary with all nodes set to $False$
    DFS_CYCLE(node, node, visited, [], cycles)
Initialize an empty list *unique_cycles*
for all cycle in *cycles* **do**
    $cycle \leftarrow cycle[:-1]$ ▷ Remove the duplicate start/end node
    normalized_cycle ← sorted tuple of *cycle*
    if *normalized_cycle* not in *unique_cycles* **then**
        unique_cycles.append(*normalized_cycle*)
return [*list*(*cycle*) for *cycle* in *unique_cycles*]

For each cycle identified, the corresponding intersection points are collected. This involves going through each cycle and selecting the intersection points that are part of that cycle. The result is a list of point sets, where each set represents the points that form a specific cycle.

For each cycle of intersecting points, the geometric centre is calculated. This is done by averaging the coordinates of the points that form the cycle. The geometric centre is essentially the centre point of the cycle and is useful for characterising the location of each geometric shape detected.

In addition to calculating the geometric centre, other metrics are used for characterisation, such as the name of the shape in the cycle.

To do this, a function was made for shape identification that consisted of classifying geometric shapes based on their vertices and centroid. First, it determines the number of vertices and sorts them by the angle to the centroid. If the vertices don't form a cycle, it returns 'Unknown with num vertices vertices'.

For three vertices, it calculates the lengths of the sides and returns 'Regular Triangle' if they are equal, otherwise 'Triangle'. For four vertices, if all the sides are equal, it returns 'Square'; if the opposite sides are equal, it returns 'Rectan-

gle'; otherwise, 'Polygon with 4 vertices'. For five vertices, it returns 'Pentagon' if the sides are equal, otherwise 'Polygon with 5 vertices'. For more than five vertices, it returns 'Polygon with num vertices vertices'.

To summarise, the function classifies basic shapes (triangles, squares, rectangles, pentagons) based on the lengths of their sides and the number of vertices.

Lastly, the angle of rotation in relation to the x-axis of the figures was calculated (based on the side with the smallest y-coordinate).

To do this, it receives as parameters a list of vertices of the shape after rotation and a string that indicates the type of geometric shape. First, the vertices are sorted based on the y coordinate to find the two points with the smallest y coordinates. These two points are stored in p1 and p2. The function then calculates the angle of the line connecting p1 and p2 with respect to the x-axis using the np.arctan2 function 8, which calculates the arc tangent of the differences in y and x coordinates between the two points, resulting in an angle in radians.

This angle in radians is then converted to degrees with `np.degrees` 9. The angle in degrees is then adjusted according to the type of geometric shape using the `angles_dict` dictionary, which maps different shapes to their standard rotation angles (for example, 60 degrees for triangles, 90 degrees for squares, and so on). The % (modulus) operator is used to ensure that the angle is within the appropriate range for the specific shape. Finally, the adjusted rotation angle is returned by the function.

$$\theta = \arctan\left(\frac{y_2 - y_1}{x_2 - x_1}\right) \tag{8}$$

where (x_1, y_1) and (x_2, y_2) are the coordinates of p_1 and p_2, respectively.

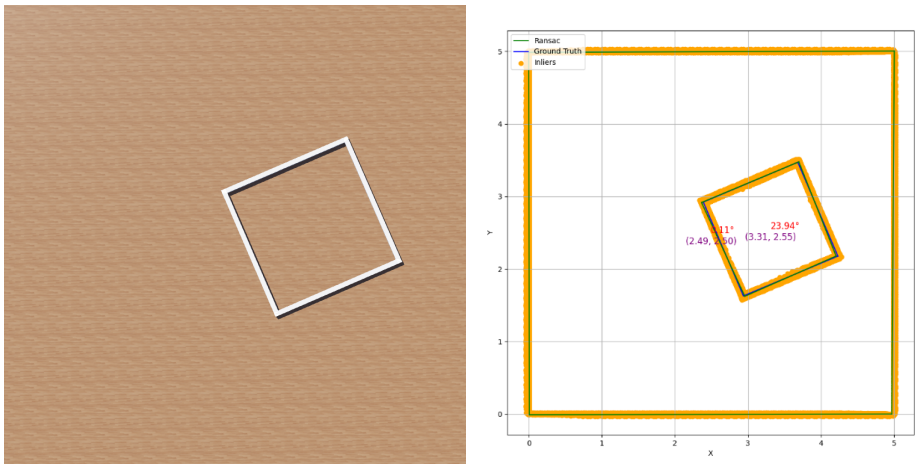
$$\text{angle\_degrees} = \theta \times \left(\frac{180}{\pi}\right) \tag{9}$$

After calculating the centers and angles and identifying the shapes, a comparison is made with the ground truth. The metrics used for evaluation are the correct classification of the shape, center positioning error, and angle error.

3.4 Results

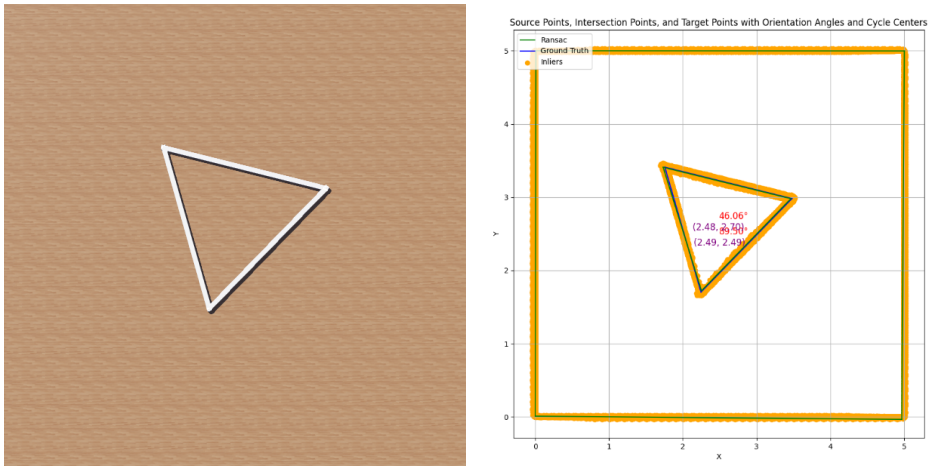
First Experiment In the first experiment, the model was tested with each primitive individually. For a single primitive from the default group (Figures 3b,4b,5b,6b and Tables 1,2,3,4), the results are highly satisfactory. The maximum error in center prediction is approximately 11.3 millimeters, and the angle error is around 3.9 degrees.

Experiment with "Special" Primitive - a Star RANSAC successfully identified the correct number of vertices but faced challenges in classifying the specific primitive shape since a star contains more than five vertices (it can only determine the primitive type if the shape has five or fewer vertices). Despite this limitation, the error remains minimal (see Table 5), indicating satisfactory performance in this scenario.



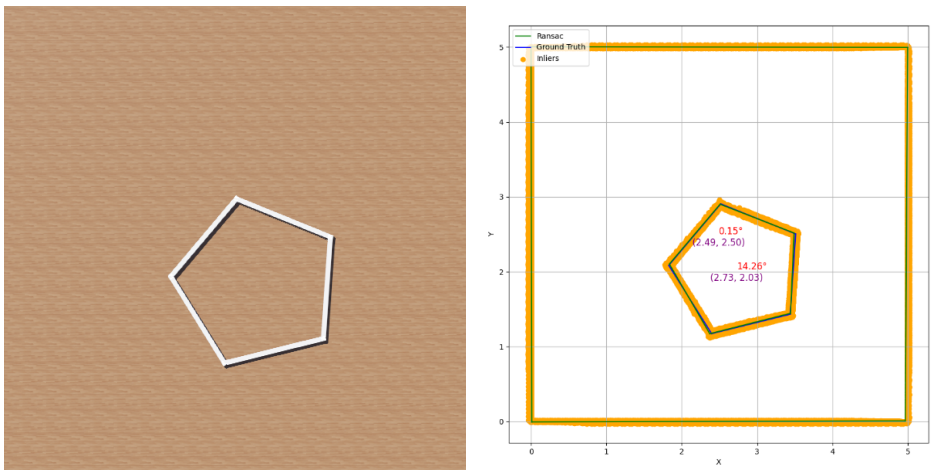
(a) Square Primitive Map (b) Square Primitive Prediction Vs Real

Fig. 3. Visual representation of Square Primitive Experiment



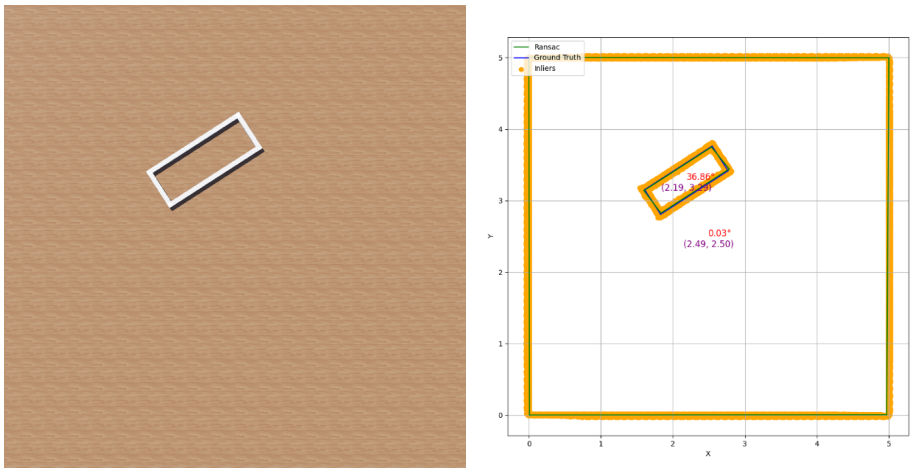
(a) Triangle Primitive Map (b) Triangle Primitive Prediction Vs Real

Fig. 4. Visual representation of Triangle Primitive Experiment



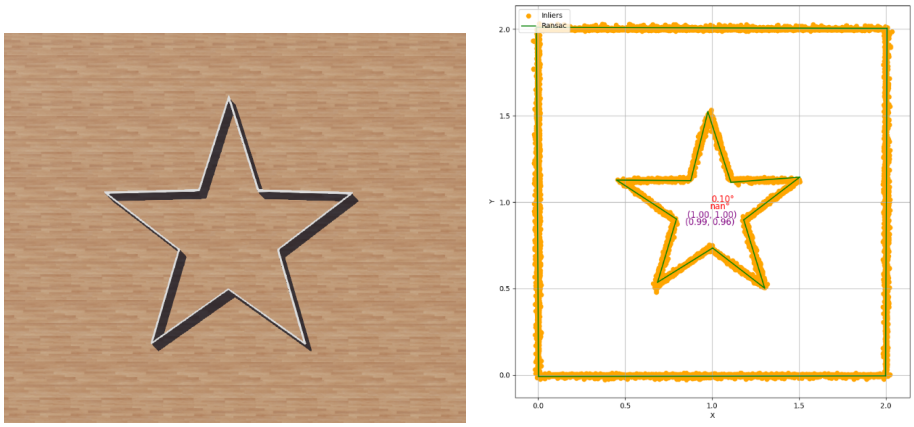
(a) Pentagon Primitive Map (b) Pentagon Primitive Prediction Vs Real

Fig. 5. Visual representation of Pentagon Primitive Experiment



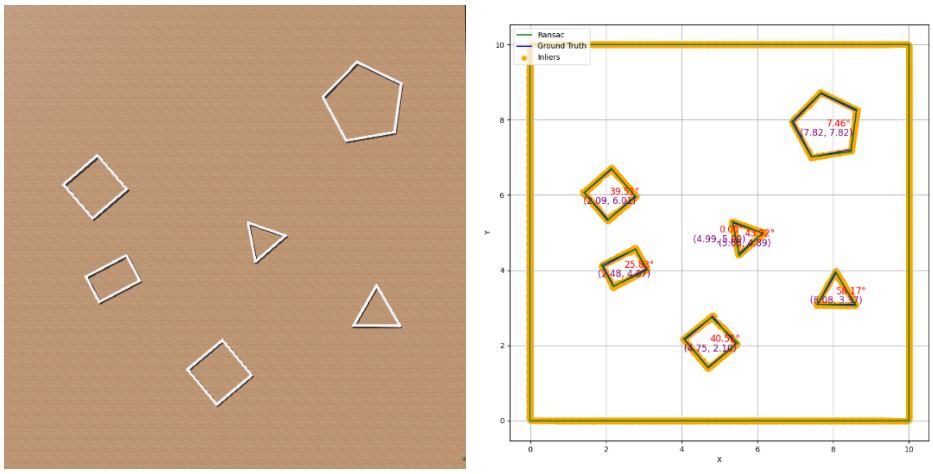
(a) Rectangle Primitive Map (b) Rectangle Primitive Prediction Vs Real

Fig. 6. Visual representation of Rectangle Primitive Experiment



(a) Star Primitive Map (b) Star Primitive Prediction Vs Real

Fig. 7. Visual representation of Star Primitive Experiment



(a) Test Map Example (b) Test Map Prediction Vs Real

Fig. 8. Visual representation of random map Experiment

Table 1. Detection Results Square Primitive

Ground Truth Shape	Detected Shape	Center Error (mm)	Angle Error (º)
Square	Square	4.536349	0.944905

Table 2. Detection Results Triangle Primitive

Ground Truth Shape	Detected Shape	Center Error (mm)	Angle Error (º)
Regular Triangle	Regular Triangle	10.527744	0.053722

Table 3. Detection Results Pentagon Primitive

Ground Truth Shape	Detected Shape	Center Error (mm)	Angle Error (º)
Pentagon	Pentagon	11.530275	0.264654

Table 4. Detection Results Rectangle Primitive

Ground Truth Shape	Detected Shape	Center Error (mm)	Angle Error (º)
Rectangle	Rectangle	2.989206	3.861961

Table 5. Detection Results Star Primitive

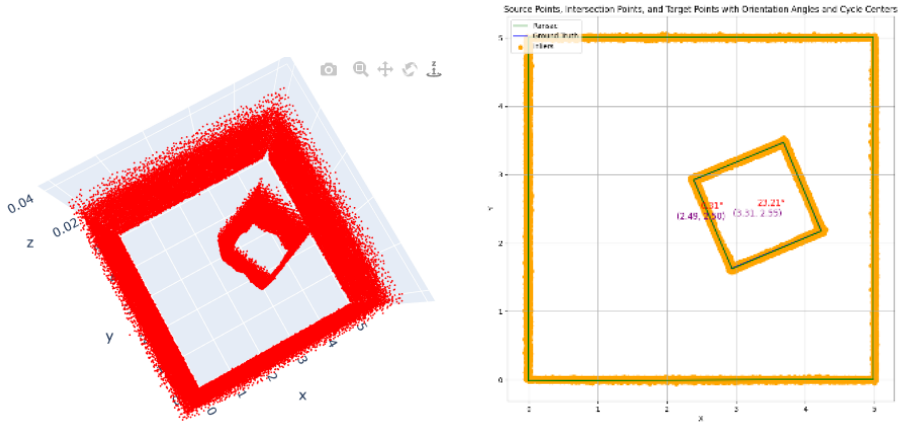
Ground Truth Shape	Detected Shape	Center Error (mm)	Angle Error (º)
Unknown	Polygon with 10 vertices	42.662973	0.325437

Table 6. Detection Results Map Test Example

Ground Truth Shape	Detected Shape	Center Error (mm)	Angle Error (º)
Square	Square	7.201172	1.493254
Square	Square	5.236529	0.505870
Triangle	Regular Triangle	6.541070	2.920182
Pentagon	Pentagon	9.345293	2.538148
Triangle	Regular Triangle	3.294407	58.171245
Rectangle	Rectangle	6.048286	2.174284

Experiment with multiple primitives For maps with multiple primitives (5-7) the results are also highly acceptable (Figure 8a and Table 6). The maximum error in center prediction is approximately 13.59mm and the maximum angle error was about 3.14 degrees. The only exception happened in a triangle whose rotation was 58.17 degrees (almost 60, the rotation that makes the regular triangle stay the same) and the algorithm's prediction was 0 degrees, resulting in the previously mentioned error.

Testing with noise and horizontal resolution Another experiment was to add noise to the LIDAR, starting with a small value of noise=0.01 to simulate real variations in the sensors 9. This procedure made it possible to assess the robustness of the processing algorithms in the face of small disturbances in the data.



(a) Point Cloud with LiDAR noise = 0.01 (b) Square with LiDAR noise = 0.01

Fig. 9. Visual representation of LiDAR Noise Experiment with noise = 0.01

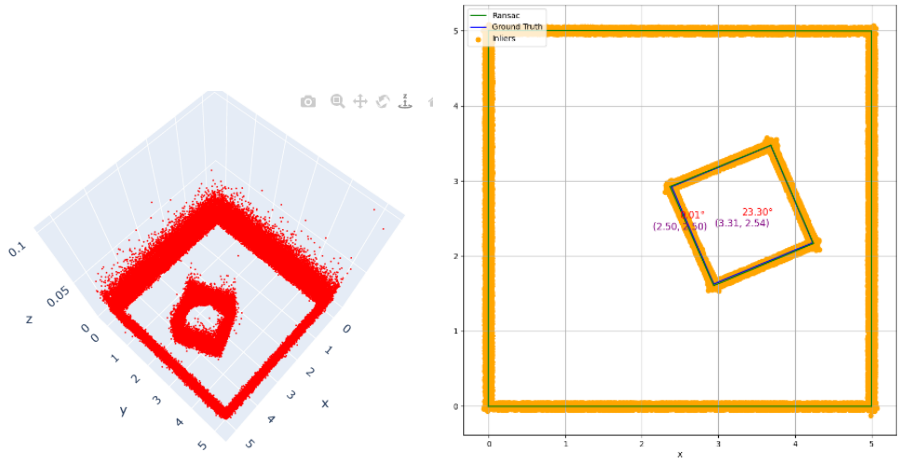
Table 7. Results LiDAR with noise = 0.01

Ground Truth Shape	Detected Shape	Center Error (mm)	Angle Error (º)
Square	Square	3.061519	0.210833

Table 8. Results LiDAR with noise = 0.025

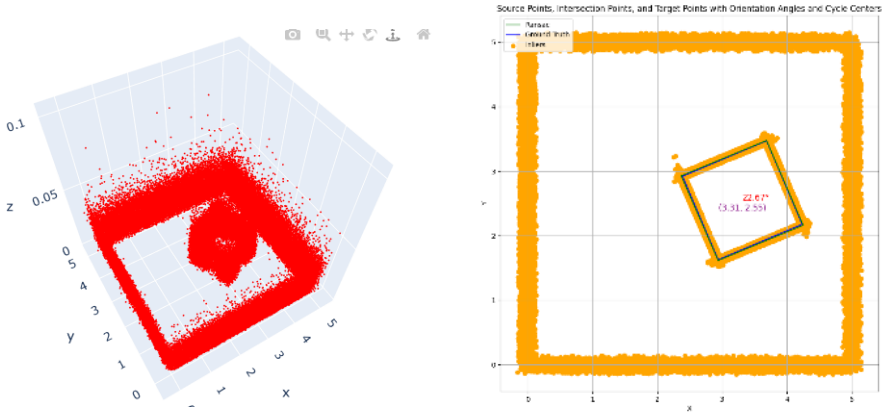
Ground Truth Shape	Detected Shape	Center Error (mm)	Angle Error (º)
Square	Square	7.484706	0.301659

The only error found when the noise was equal to 0.05 11 was that RANSAC detected 16 planes instead of the normal 8 (4 of the square and 4 of the arena).



(a) Point Cloud with LiDAR noise = 0.025 (b) Square with LiDAR noise = 0.025

Fig. 10. Visual representation of LiDAR Noise Experiment with noise = 0.025



(a) Point Cloud with LiDAR noise = 0.05 (b) Square with LiDAR noise = 0.05

Fig. 11. Visual representation of LiDAR Noise Experiment with noise = 0.05

Table 9. Results LiDAR with noise = 0.05

Ground Truth Shape	Detected Shape	Center Error (mm)	Angle Error (º)
Square	Square	4.665878	0.330059

However, when the noise level was increased to 0.06, no centre was found 12. As the centres are needed to calculate the metrics, no results could be obtained, indicating that the code works correctly up to a maximum noise limit of 0.05.

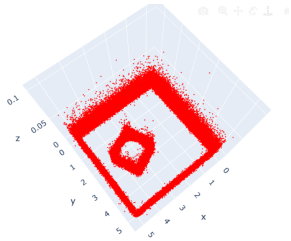


Fig. 12. Point Cloud with LiDAR noise = 0.06

Various parameters were also tested for the Horizontal resolution, which is set to 100 by default. Several variations were examined and it was observed that even with a horizontal resolution as low as 1, the code worked effectively. Therefore, variations in horizontal resolution do not significantly affect the code's performance.

4 Conclusions and Future Work

This paper presents a method for accurately detecting and identifying 3D primitives using a custom RANSAC algorithm applied to LiDAR point cloud data. The approach integrates simulated environments with a mobile robot equipped with a LiDAR sensor to showcase its robustness in detecting and characterizing diverse 3D primitives. The methodology underwent systematic development and validation through simulations, with the goal of enhancing reliability in determining the pose and parameters of 3D primitives. Experimental results consistently demonstrate high performance, showing excellent accuracy with minimal errors in the majority of cases.

While the proposed method demonstrates significant accuracy in detecting and identifying 3D primitives, several avenues for future work can enhance its robustness and applicability. Integrating additional sensors, such as stereo cameras or depth sensors, could improve detection accuracy and provide richer environmental context. Transitioning from a simulated environment to real-world

testing will evaluate the algorithm's performance in diverse and unpredictable conditions, identifying potential limitations and necessary adjustments.

Adapting the algorithm for dynamic environments, where obstacles may move or change over time, will require implementing real-time updating mechanisms for point cloud data and corresponding shape adjustments.

Extending the methodology to detect and identify more complex shapes and structures, including non-planar surfaces and irregular geometries, involves enhancing the algorithm to fit a wider variety of primitive models. These improvements will significantly increase the method's effectiveness and versatility in various real-world applications.

References

[1] Schnabel, R., Wahl, R., Klein, R.: Efficient ransac for point-cloud shape detection. In: Computer graphics forum. vol. 26, pp. 214–226. Wiley Online Library (2007)

[2] Se, S., Lowe, D.G., Little, J.J.: Vision-based global localization and mapping for mobile robots. IEEE Transactions on robotics 21(3), 364–375 (2005)

[3] Charrow, B., Kahn, G., Patil, S., Liu, S., Goldberg, K., Abbeel, P., Michael, N., Kumar, V.: Information-theoretic planning with trajectory optimization for dense 3d mapping. In: Robotics: Science and Systems. vol. 11, pp. 3–12 (2015)

[4] Adam, A., Chatzilari, E., Nikolopoulos, S., Kompatsiaris, I.: H-ransac: A hybrid point cloud segmentation combining 2d and 3d data. ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences 4, 1–8 (2018)

[5] Li, L., Sung, M., Dubrovina, A., Yi, L., Guibas, L.J.: Supervised fitting of geometric primitives to 3d point clouds. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 2652–2660 (2019)

[6] Cyberbotics: Wall.proto. https://webots.cloud/run?version=R2023b&url=https%3A%2F%2Fgithub.com%2Fcyberbotics%2Fwebots%2Fblob%2Freleased%2Fprojects%2Fobjects%2Fapartment_structure%2Fprotos%2FWall.proto