Position Papers of the 20th Conference on Computer Science and Intelligence Systems (FedCSIS) pp. 109–117 DOI: 10.15439/2025F7442 ISSN 2300-5963 ACSIS, Vol. 44

DBRow: A Density-Based algorithm for autonomous navigation within crop rows

Peder Ø. Bukaasen, Weria Khaksar
Norwegian University of Life Sciences, Ås, Norway
Email: peder.ormen.bukaasen@nmbu.no; weria.khaksar@nmbu.no

***Abstract*—This paper introduces DBRow, a density-based algorithm designed to improve autonomous navigation within crop rows, addressing the growing need for efficient agricultural robotics to boost productivity and tackle labour shortages. DBRow integrates Simultaneous Localisation and Mapping (SLAM) with Density-Based Spatial Clustering of Applications with Noise (DBSCAN), overcoming the limitations of previous navigation systems that relied solely on LIDAR data for NMBU's FRE participation. Experiments conducted in simulated and controlled indoor environments evaluated DBRow using A* path planning algorithm. The results show some weaknesses in the simulated environment, but it performs well in the controlled indoor environment. The paper calls for further testing for statistically significant results and suggests future enhancements, including LIDAR preprocessing improvements and machine learning integration, to optimise navigation accuracy and automate tasks like pesticide application.**

Keywords: Robotics, Navigation, Agriculture, Farming, crop, autonomous

I. Introduction

AGRICULTURAL robotics is a broad field that involves various robots performing tasks in agricultural environments, replacing or aiding humans. Such robots are often divided into self-propelled mobile robots and robotic sensors or actuators carried by a vehicle [24]. This paper focuses on the first one, self-propelled mobile robots.

Navigating crop environments seems like a straightforward task for humans: go in the middle of the row and do not destroy any plants. Enabling navigation for a mobile robot requires more work. First, the robot needs to have some representation of its environment so that it can plan when and where it should go. The representation of the environment in this paper is a map created by a SLAM algorithm. To navigate its environment, the robot needs a planner and a controller to move the robot; the Robot Operating System 2 (ROS2) Navigation stack solved this. To autonomously navigate, an algorithm was needed to set goal points. Here, Density-Based Spatial Clustering of Applications with Noise (DBSCAN) was used to extract rows and goal positions were extracted from these rows.

II. Methodology

A. DBSCAN

DBSCAN is a clustering algorithm that can extract clusters of varying sizes, assuming they have roughly the same density. This algorithm was first proposed in [7]. As the name implies, DBSCAN uses the densities of points to assign cluster labels. Density in DBSCAN is defined as the number of points within a specified radius eps. In the literature, this radius is also denoted by ϵ. Compared to other clustering algorithms, one advantage of this algorithm is that it does not require a number of clusters to find as input. Two other advantages are that it does not make assumptions about spherical clusters as k-means clustering does, and it does not partition the dataset into hierarchies that require some manual cut-off. The DBSCAN algorithm uses three-point labels: core, border, and noise points. These points are defined in this way:

- Core points have a minimum number of points ($MinPts$) within the radius eps.
- Border points fall within the eps of another core point but do not satisfy the $MinPts$ within the radius eps.
- Noise points neither satisfy the condition for border nor core points.

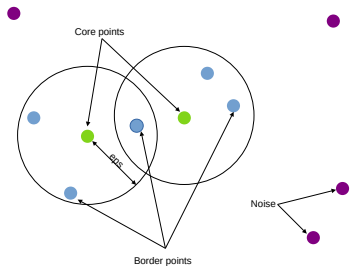


Fig. 1. Shows the different points and the eps variable used in the DBSCAN algorithm. All the purple points are noise, all the green points are core points, and all the blue points are border points.

Figure 1 shows how DBSCAN can label these points. The $MinPts$ in this example is three, and the circle shows the radius around the green points. One can see that the green points are labelled as core points since they contain three or more points within eps. The blue points are labelled border points since they fall within the radius of the core points. The purple points are labelled noise since they do not satisfy the conditions for core or border points. The algorithm can be simplified to:

1) Label all points into the three different point labels.
2) Create separate clusters for all core points or groups of core points. Two core points are considered to be in the

same cluster if they fall within the radius eps of each other.
3) Assign all border points to their respective core points.

Using these simple steps, DBSCAN can detect clusters of any shape or form as long as they are separated and have similar densities [19].

B. RANSAC

Random Sample Consensus (RANSAC) is an algorithm for fitting models to experimental data. Fisher and Bolles proposed the algorithm in 1981 [8]. The RANSAC algorithm can be seen as a trial-and-error approach to fitting data to a model where the dataset is contaminated with noise. RANSAC can be explained in four easy steps:

1) Sample the number of data points needed to fit the model.
2) Calculate the model parameters from the collected data points.
3) Score the model by the number of inlines with a predefined threshold.
4) Repeat the above steps until the best possible model is found.

Using these simple steps, RANSAC is able to fit models to the given data [5].

C. A*

The A* search algorithm was first introduced in 1968 in [10]. A* is a widely used pathfinding and graph traversal technique that utilised the strengths of both Dijkstra's algorithm and Greedy Best-first search. It is designed to efficiently compute the shortest path from a starting point to a goal node in a graph, making it particularly useful in robotics and game development. The algorithm is graph-based, and the conversion described in the path planning section is necessary for this algorithm. A* integrates the methodical search of the Dijkstra algorithm with the heuristic-driven guidance of the Greedy Best-First search. This guidance is implemented as two metrics:

- **g(n)**: The exact cost from the start node to the current node n.
- **h(n)**: The heuristic estimate of the cost from node n to the goal node.

A* evaluates paths by minimising this function:

$$f(n) = g(n) + h(n) \tag{1}$$

This function ensures a balance between the actual cost from the start and the estimated cost to reach the goal; this leads to efficient and optimal path planning [17]. The algorithm, when stripped down to its basics, is quite simple; it uses two sets: Open and Closed. The Open contains nodes that are candidates to explore. Initially, the Open set contains only the starting position. The Closed set contains nodes that have already been examined and begin empty. Each node contains a pointer to its parent to help create the optimal path at the end. The algorithm runs through a main loop that repeatedly selects the best node n from the Open set, which is the node with the lowest $f(n)$ score, and examines it. If n is the goal, the process ends; otherwise, n is moved from the Open set to the Closed set. Then, the neighbours of n that are already in the closed set are ignored, and the neighbours in the open set are scheduled to be examined. If a neighbour is not in the Open or the Closed set, it is added to the Open set with parent n [16].

D. Hardware

The robot platform used in the simulation is Peik, which Bård Tollef Pedersen and I built. In Figure 2(a), one can see an image of Peik without any sensors mounted. The specs of this robot platform are:

- **Weight:** 19.56 kg
- **Onboard computer:** Nvidia Jetson Nano ORIN
- **Operating system:** Jetpack 6.0
- **Steering type:** Skid-steer
- **Driven motors:** 4 x 350W motors, two on each side
- **Battery:** 36v 4.4Ah
- **Footprint (L x W x H):** 42 cm x 32 cm x 25 cm
- **Max speed:** 5.55 m/s
- **LIDAR:** Ouster OS1-64 (in the simulation)

For testing at the robotics lab, A Turtlebot3 Burger[22] was used. In Figure 2(b), one can see the Turtlebot3 Burger. The Turtlebot3 Burger has the following specs:

- **Weight:** 1 kg
- **Onboard computer:** Raspberry Pi 4
- **Operating system:** Ubuntu Server 22.04.5 LTS (64-bit)
- **Steering type:** Differential drive
- **Driven motors:** 2 x
- **Battery:** 11.1v 1800 mAh
- **Footprint (L x W x H):** 13.8 cm x 17.8 cm x 19.2 cm
- **Max speed:** 0.22 m/s
- **LIDAR:** LDS-02
- **IMU:** Gyroscope 3 Axis, Accelerometer 3 Axis

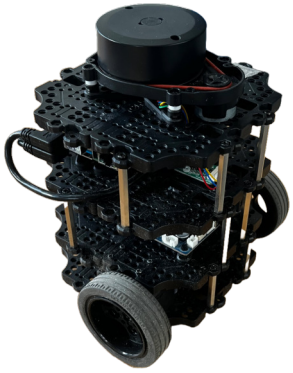
The simulations of Peik were run on a computer with a dedicated GPU. The PC used for these simulations had these specs:

- **Processor (CPU):** Intel Core I7-8700 6-Core 12-Thread 3.2/4.6 GHz
- **Graphics Processing Unit (GPU):** Nvidia GeForce GTX 1080
- **Memory:** 16 GB DDR4 2666 MHz
- **Storage:** 1000 GB M.2 SSD
- **Operating System:** Ubuntu 22.04 Jammy Jellyfish

Since the Turtlebot3 only has a Raspberry Pi 4, the code was run on a PC connected to the Turtlebot3. The computer had these specs:

- **Processor (CPU):** Intel Core Ultra 5 125H 14 Core 18 Thread 1.2/4.5 GHz
- **Memory:** 16 GB LPDDR5X
- **Storage:** 1000 GB M.2 SSD
- **Operating System:** Ubuntu 22.04 Jammy Jellyfish

(a) Peik



(b) Turtlebot3 Burger

Fig. 2. Shows the robots used in this paper, Peik (a) without any sensors mounted and the Turtlebot3 Burger (b) with the LDS-02 LIDAR.

E. Setting up the simulation environment

Setting up the simulation environment in Gazebo consists of a few steps: creating a world for the robot to move in, creating a robot model, and adding a control system and sensors to the robot. The world used for simulation in this project is generated with code from this GitHub repository [3]. This code has several options that change how the created field looks; these options can change how straight the rows are if there are holes in the crop rows and the size of the plants. These values can be specified in a YAML file. This paper uses three different simulated worlds with the main difference being the roughness of the terrain.

The robot used in this paper is a simulated version of Peik. Peik is the robot created for NMBU's participation in the Field Robot Event (FRE). A simulated version of this robot was created using Unified Robot Description Format (URDF) and Gazebo plugins for sensors and controlling the robot. The simulated robot was simplified to a box with four wheels. Utilising Xacro, the URDF files were further simplified with macros for values used several times, such as the wheel's mass or the offset of the different wheels. The base of the robot was created as a box with tags for the visual, collision and inertial. The wheels of the robot were connected using continuous joints, and the wheels also had tags for visual collision and inertia. Controlling the robot in Gazebo was done with this plugin *libgazebo_ros_diff_drive.so* [9]. This plugin leverages the wheel joints, wheel size, and wheel separation to facilitate robot control via the /cmd_vel topic. Additionally, it publishes odometry data to track the robot's movement. Simulating the Ouster OS1-64 was done by using the *libgazebo_ros_velodyne_laser.so* plugin [23] and changing the values for horizontal and vertical scans and ranges to match those of the Ouster OS1-64 [15]. This plugin simulates the LIDAR in Gazebo and publishes the point cloud to a topic. In Figure 3, one can see the simulated version of Peik in the virtual maize environment.

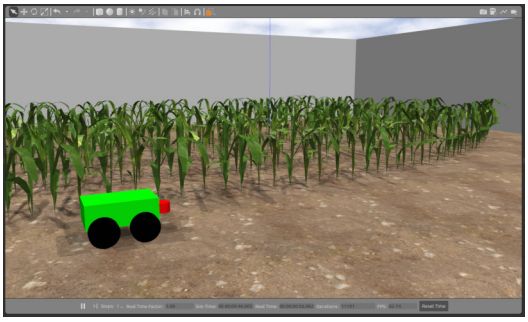
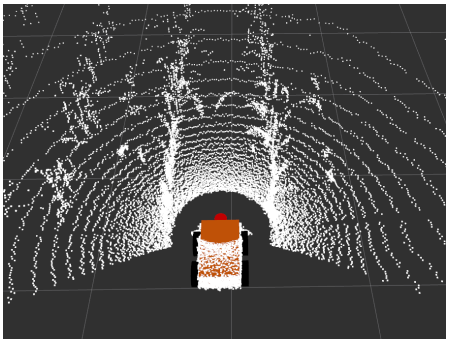


Fig. 3. Shows the robot with a LIDAR sensor in the simulated maize field environment. Here visualized in Gazebo.

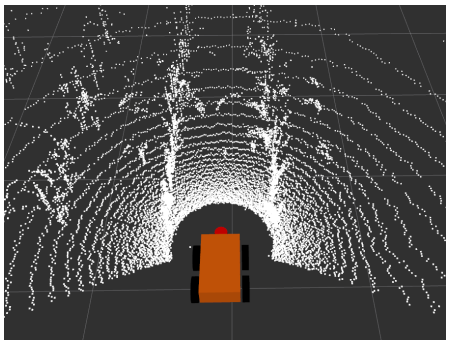
F. LIDAR preprocessing

This part is only used for the simulated robot since the Turtlebot3 had a 2D lidar and navigated in an indoor environment. The simulated lidar is a 3D sensor, and since Nav2 is mainly used for 2D data, this point cloud was projected into two dimensions. Before it was projected, some points had been removed. Firstly, the points that fell on the robot's chassis needed to be removed. This was accomplished by using a pcl::CropBox filter from the Point Cloud Library (PCL) [18]. This was a straightforward process of creating a box representing the robot, and the filter removed all points inside the box. The unprocessed point cloud can be seen in Figure 4(a), and the point cloud with the points from the robot filtered away can be seen in Figure 4(b).

Additionally, the ground plane needed to be filtered out since it was not used for navigation. The ground plane was filtered from the point cloud using RANSAC to fit the point cloud to a plane model, utilising the RANSAC filter from PCL. In this filter, restrictions were put such that the plane's normal must be within an angle threshold of the z-axis. This filter also had a maximum number of iterations to make sure that it did not run forever. Removing points was done using a threshold, and all points that fell within a threshold of the fitted plane were removed. Trial and error were used to find

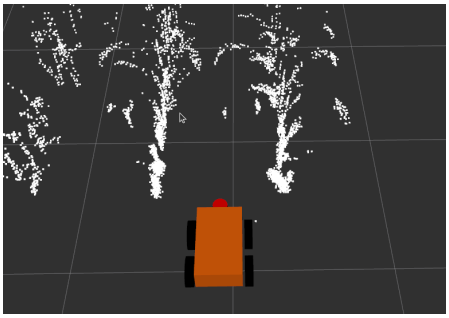


(a) Point cloud

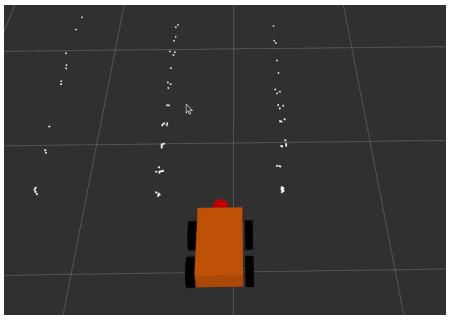


(b) Filtered point cloud

Fig. 4. (a) shows the point cloud from the simulated ouster and (b) shows the point cloud with the robot footprint filtered.



(a) Ground removed point cloud



(b) Laser scan

Fig. 5. (a) shows the point cloud where RANSAC has removed the ground, and (b) shows the projected laser scan.

the best parameters for this algorithm. The point cloud with the ground removed can be seen in Figure 5(a).

Finally, projecting the 3D point cloud into a laser scan was done utilising the pointcloud_to_laserscan package [20]. This package has several options for converting the point cloud to a laser scan; among these are min_height and max_height, which are the minimum and maximum heights to sample from the point cloud. These two parameters were tuned such that the points that remained in the laser scan mainly consisted of plant stem points. This laserscan can be seen in Figure 5(b).

1) Cartographer: Cartographer is a project developed by Google that provides a real-time solution for indoor 2D mapping using a sensor-equipped backpack [11]. The algorithm is also integrated with ROS with the cartographer ROS project. This implementation also offers real-time SLAM for 2D and 3D environments [4]. The SLAM algorithm used by Cartographers combines local and global optimisation strategies to maintain accurate mapping. Both of these approaches aim to optimise the pose of LIDAR scans [11]. The two different optimisation strategies are implemented as two related subsystems: the local and the global SLAM. Local SLAM constructs submaps that are locally consistent, accepting that they may drift over time. It handles immediate data from sensors to build submaps that are small enough to ensure local accuracy but large enough to be distinct for effective loop closure. Global SLAM runs in the background, focusing on loop closure by scan-matching scans against submaps and incorporating additional sensor data for the most consistent global map.

2) Nav2: Nav2 is a toolbox for ROS2 that allows autonomous navigation of mobile and surface robots. It is a successor to the ROS Navigation Stack and provides packages for perception, planning, control, localisation, and visualisation. Nav2 uses behaviour trees to enable autonomous navigation, which is achieved using several independent modular servers. A server can be used to localise the robot on the map or plan a path from point A to point B. These servers communicate with the behaviour tree using the different ROS2 interfaces: services, actions and topics [13]. The core of the navigation problem can be seen as planning and controlling a robot. Four of the servers in the Nav2 stack provide a robust solution for planning and control: Planner, Controller, Smoother and recovery servers [14].

G. Configuring Cartographer and Nav2

Configuring Cartographer is done by creating a *.lua file with all the parameters needed to launch the Cartographer package. This file was created by consulting the tuning guide [2], and the Lua configuration reference documentation [12].

Configuring Nav2 can be quite demanding due to its multiple components that require careful configuration and tuning. For the initial setup of the planners and controllers, the guide referenced in [21] was utilised, which outlines when to use different planners and controllers, as well as their suitability for various types of robots.

H. Navigation algorithm

The navigation algorithm can be divided into the in-row and switch-row algorithms. This subsection explains the two algorithms and the implementation of them. Both of these navigation algorithms use the information from the global costmap to set navigation goals. The data from the laser scan mainly contains plant stem values, which means that the data in the global costmap also contains plant stems. The in-row navigation is best described in some simple steps:

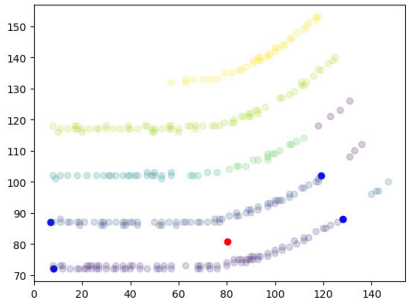
1) Get the robot's position and costmap data.
2) Cluster the costmap data using DBSCAN.
3) Find the two closest clusters to the robot, then for each of these clusters, find the furthest points from each other within its cluster.
4) From these four points, find the two closest pairs.
5) For these two pairs, calculate the mean, which should be two points in the middle of the crop rows, one in front of the robot and one behind the robot.
6) Transform the coordinates of these points into the coordinate system of the robot to easily calculate which goal is behind the robot.
7) Check if one of these goals is in front of the robot.
8) If one of the goals is in front of the robot, navigate to this goal. If not, the robot is at the end of the row.
9) When the goal is reached, repeat from step 1.

To obtain the map's position and the robot's pose, subscribers were created to track the global cost map from the global cost map topic and the robot's pose from the tracked pose topic. Extracting clusters from the cost map data involved several steps. First, lethal obstacles, defined as all values greater than 100, were extracted from the cost map. These values were then converted into a NumPy array.

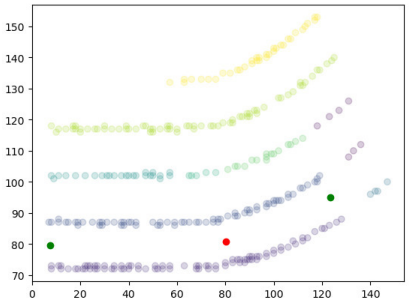
Using this array, the DBSCAN function from scikit-learn [6] was applied to cluster the data into crop rows. Next, the two closest clusters were identified by iterating through all clusters and calculating the distances between them, retaining only the two with the smallest distances. Simultaneously, the two furthest points within each cluster were determined by employing a nested loop to calculate the maximum distances. The two points that were furthest apart were saved, along with the corresponding distance for each cluster.

The goal was defined using the four points that represented the furthest distances from each other within the two closest clusters, visualised by the blue points in Figure 6(a). The four distances between these points were then compared to identify the two pairs closest to each other. From these pairs, two potential goal positions were calculated by finding the mean of the two closest pairs, represented by the green points in Figure 6(b).

Since the goals were referenced in the map's coordinate frame, they were transformed into the robot's coordinate system to determine which goals were in front of the robot. In this transformed frame, the goals behind the robot had negative values, while those in front had positive values. The goal with the largest x value was selected. If this goal was too close to the robot, it was also considered behind the robot. If both goals were determined to be behind the robot, the process would terminate, indicating that the robot had reached the end of the row. To navigate to these goals, Nav2's simple commander was used. The algorithm for switching between the rows has



(a) Four furthest points



(b) Two possible goals

Fig. 6. Shows the clustered points, the red points represent the robot's position, the blue points represent the further four points in the two closest clusters, and the green points show the two possible goals. The transparent points show the different clusters. (a) shows the points used to calculate the goals, and (b) shows the two possible goals.

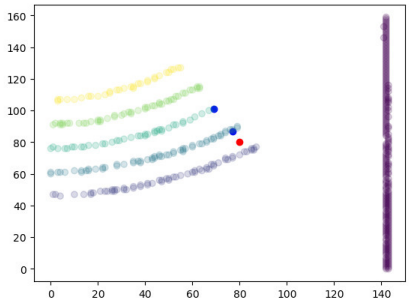
the same first two steps as the navigate row algorithm: getting the robot's position, map data, and clustering the rows. This algorithm can be described in these steps:

1) Get the robot's position and costmap data.
2) Cluster the costmap data using DBSCAN.
3) Find the minimum distance and the closest point in each of the clusters.
4) Transform the coordinates of these clusters into the robot's coordinate frame.
5) If turning left, keep all closest cluster points with a positive y value; if turning right, keep all closest cluster points with negative y values.
6) Select the two closest points from these clusters and find the mean of them, this mean is then the goal point.
7) Navigate to this goal.

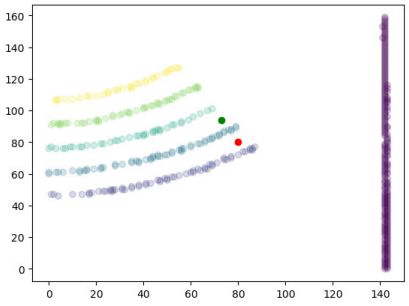
The first two steps of the switch row algorithm are the same as the navigate row algorithm. Therefore, they will not be explained further. This algorithm was implemented as an

action server in ROS2 Humble, and it also had a custom action.

The third step of the algorithm was completed by looping through all the clusters and calculating the closest distance from each cluster to the robot. The closest point was then saved together with the distance for each cluster. These points were then transformed into the coordinate frame of the robot. This transformation was done to easily separate the clusters to the left and right of the robot using the y-axis of the robot's coordinate frame. The cluster points with positive y values are to the left of the robot, and all cluster points with negative y values are to the right of the robot. Depending on the turning direction, a list of interesting clusters was created, containing only the cluster points and distances to the left or the right. From this list, the two closest clusters were selected, and the goal position was calculated as the mean of the closest points in these two clusters. These two points can be seen by the blue points in Figure 7(a), and the goal can be seen in Figure 7(b) by the green point. The heading of this goal was set to the inverted heading the robot had when standing at the end of the row, which was inverted by adding 180 degrees to it. The goal was then sent to the Nav2's simple commander. This implementation returned true if it was able to calculate and navigate to the goal; otherwise, it returned false. These



(a) Closest clusters points



(b) Goal in next row

Fig. 7. Shows the clustered points, the red points represent the robot's position, the two closest clusters closest points, and the green points show the goal position. The transparent points show the different clusters. Figure (a) shows the two closest, and Figure (b) shows the goal.

two action servers were used to navigate the entire field using two action clients implemented in one ROS2 node. For this to work, the number of rows to navigate and the first turning direction need to be specified. This node starts by initiating the action clients and the required variables for tracking the navigation, like row number and initial turning direction. Then, a goal is sent to the navigation row server. This node then waits for the node to finish while receiving and printing the feedback. When the navigate row server finishes, the switch row server is called. The robot then switches to the next row and the direction of the switch alternates between switching to the left and right. The node waited for the execution of this action server while receiving and monitoring the feedback. This node alternated between calling the navigate row action and calling the switch row action until the specified number of rows were navigated or the navigation failed, and the node shut down.

I. Experiment setup

The experiments conducted in this paper can be divided into two types: those conducted in the simulator with virtual maize plants and those conducted in the robotics lab with Turtlebot3 and thuja plants. Here, five runs in each were completed, following the design from here [1]. The simulated environment was created with five plant rows, with approximately 70 centimetres between them. In Figure 8, one can see the layout of the field used in the simulated run.

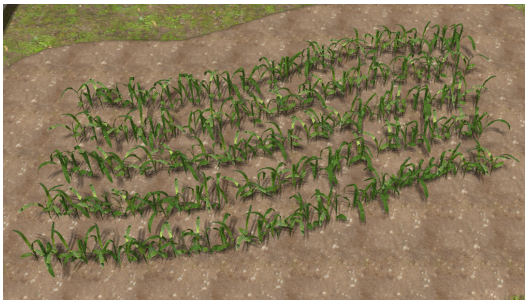


Fig. 8. Shows the layout for the simulated maize field.

In Figure 9, one can see the terrain from the simulated environment.

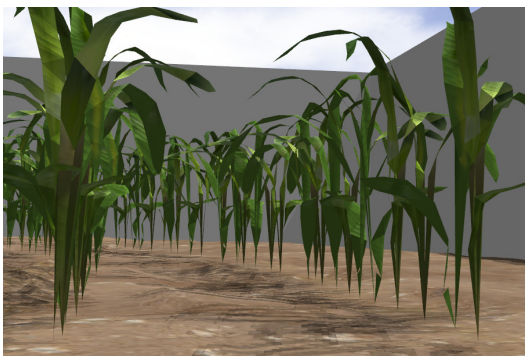


Fig. 9. Shows the terrain for the simulated maize environment

The testing environment in the robotics lab at NMBU was created using thuja plants. These plants were used because they were the only available plants in the robot lab. This environment was similar to the even simulated environment in the sense that they both have a very even ground. In the testing with the Turtlebot3, the LIDAR preprocessing steps were skipped since this robot has a 2D LIDAR. Four rows of plants were created, which meant three rows for the Turtlebot3 to navigate. These rows were approximately 3 meters long and had 0.7 meters between each row. In Figure 10, a square at the start of the row is visible. This square was used as a starting position for the robot to ensure similar conditions for all the runs.

Fig. 10. Shows the testing environment in the robotics lab at NMBU.

Measuring the performance of the algorithm was done using parameters similar to those used in the FRE competition. The performance was measured by the time used to navigate, the distance travelled, and how many plants were damaged. Calculating the distance and time was done by creating a node that subscribed to the position of the robot. This node was started by using a topic to publish, starting and stopping from the navigate field node. This node then used the positions over time to calculate the distance traversed by the robot. Due to an uncertainty in the position of the robot, a threshold was used to eliminate noise in the position data. A plant that the robot touched was not considered damaged; for a plant to be considered damaged, it had to be lying on the ground or visibly damaged. Detecting damaged plants was done by observing the simulation and the Turtlebot3 in the robotics lab.

III. Results and Discussion

A. Simulated environment

In this subsection, the results from the simulation terrain runs are presented. In Table I, the results are presented. Here, the average number of completed rows was 3.0, and the average number of damaged plants was 2.0. All of the plant damage occurred in run three. The average distance the robot managed to travel was 24.08 meters, and the average time was 234.0 seconds.

In Figure 11, the paths taken by the robot using A* are shown in the red line, and the green points show the plant's ground truth positions. In run three, ten plants were damaged, and where the plants were damaged can be seen by the overlap between the red line and the green plants. Runs two and five did not complete the field. Runs three and four had some assistance at the last row. The overlap of the red path and the green points in Figure 11. In runs two and five, one can see where the robot failed in the middle of row two. The robots in these runs were able to navigate 24.08 meters on average, which is good since the entire field is about 30 meters. This is promising for using DBSCAN to navigate crop rows. In the third run, one can see that the robot struggled a lot; this could be due to the rough terrain, causing the RANSAC not to be able to fit the plane and remove the ground points. This would add noise to the input data of the algorithm and could be the cause of the plant damage in this field, And also, what caused the robot to fail in the second and fifth runs. The robot damaged plants can be seen by the overlap of the red line and the green points in Figure 11. Another possible explanation for the poor performance in runs two, three and five could be that the rough terrain makes the point cloud laser scan pick up leaf points in the middle of the rows due to the robot being tilted.

TABLE I
Shows the results using A* in the simulated environment

Run Number	Plants damaged	Distance [m]	Time [s]
1	0.0	30.64	215.0
2	0.0	11.80	216.0
3	10.0	36.30	466.0
4	0.0	30.53	199.0
5	0.0	11.11	74.0
Average	2.0	24.08	234.0

B. Turtlebot3

This subsection presents the results from the testing in the robotics lab using the Turtlebot3 Burger. Table II shows the results for the A* planner with the Turtlebot3. Here, the Turtlebot3 managed to complete the three rows in all runs without damaging any plants. The average distance used was 9.29 meters with an average time of 99.2 seconds.

TABLE II
Shows the results for the runs with the Turtlebot

Run Number	Plants damaged	Distance [m]	Time [s]
1	0.0	9.23	95.0
2	0.0	8.72	96.0
3	0.0	9.17	94.0
4	0.0	9.51	95.0
5	0.0	9.82	116.0
Average	0.0	9.29	99.2

In Figure 12, the paths taken by the Turtlebot3 can be seen for the five runs using A*. The path in red is plotted here on the map generated by cartographer slam. Here, one can see that the robot mainly navigated to the middle of the rows and kept a distance when switching between the rows using A*. The first row here is the lowest row of plants in Figure 12, and the last row is the top row. At the beginning of the second row

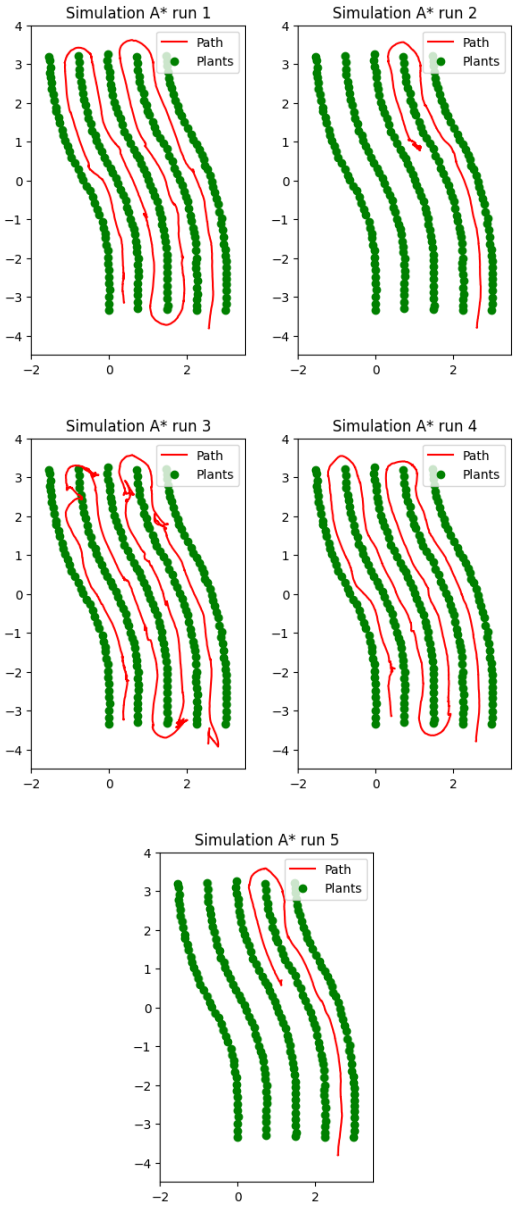


Fig. 11. Shows the five runs conducted in the rough terrain with A*. The green points visualise the ground truth position of the plants, and the red line visualises the path taken by the robot.

in run two, one can see that the robot navigated a bit closer to the plants.

The runs with the Turtlebot3 show good promise for this algorithm. One explanation of this performance could be that this environment is much simpler than the simulated one. The Turtlebot3 also did not need the preprocessing steps used in the simulated environment to remove the ground and extract stem points, since it used a 2D LIDAR and the ground in this environment was flat.

For both the Simulated and runs, one could not draw any definite conclusions since only five runs were conducted. These runs can only give an indication of the algorithm's performance.

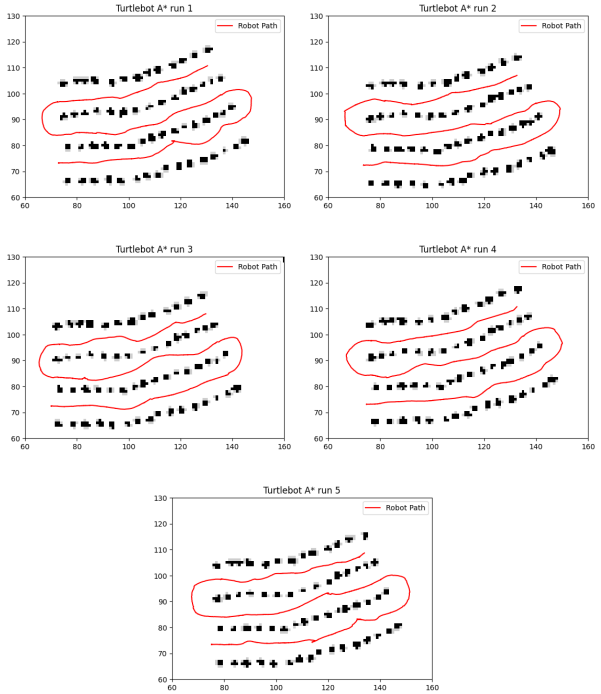


Fig. 12. Shows the five runs conducted with the Turtlebot. Here, the path is plotted in red on the map created by cartographer.

IV. Conclusion

To conclude, this paper introduces the DBRow navigation algorithm for autonomous navigation within crop rows. This algorithm addresses the limitation of the algorithm used in NMBU's last participation in FRE, which relied solely on LIDAR data. Through experiments conducted across different terrains and setups, this algorithm shows potential for being a more robust solution. This algorithm struggled a bit in the simulated terrain, but performed well in the robotics lab. A key weakness of these results is the limited number of experiments that restrict definitive conclusions. This limited number of experiments highlights the need for more expensive testing to achieve statistically significant results. The focus of further work should be on improving the lidar preprocessing and adding some object detection models for stem detection could also enhance the navigation algorithm. Conducting more extensive testing is crucial to validate the preliminary findings and refine the algorithm for practical deployment in actual agricultural environments. Additionally, adapting the navigation task to automate the manual task could enhance the algorithms' use case for agricultural operations.

Acknowledgment

This work is a part of the DLT-Farming project funded by the research council of Norway with the agreement number

344288.

REFERENCES

[1] Francisco Affonso et al. "CROW: A Self-Supervised Crop Row Navigation Algorithm for Agricultural Fields". en. In: *Journal of Intelligent & Robotic Systems* 111.1 (Feb. 2025), p. 28. ISSN: 1573-0409. DOI: 10.1007/s10846-025-02219-2. URL: https://link.springer.com/10.1007/s10846-025-02219-2 (visited on 03/10/2025).

[2] *Algorithm walkthrough for tuning — Cartographer ROS documentation*. URL: https://google-cartographer-ros.readthedocs.io/en/latest/algo_walkthrough.html (visited on 03/17/2025).

[3] Johannes Barthel et al. *Virtual Maize Field*. URL: https://github.com/FieldRobotEvent/virtual_maize_field.

[4] *Cartographer ROS Integration — Cartographer ROS documentation*. URL: https://google-cartographer-ros.readthedocs.io/en/latest/ (visited on 03/17/2025).

[5] Stanchniss Cyrill. *RANSAC – Random Sample Consensus*. Photogrammetry & Robotics Lab. University of Bonn. URL: https://www.ipb.uni-bonn.de/html/teaching/photo12-2021/2021-pho2-06-ransac.pptx.pdf (visited on 04/04/2025).

[6] *DBSCAN — scikit-learn 1.6.1 documentation*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html (visited on 04/13/2025).

[7] Martin Ester et al. "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise". en. In: (1996). (Visited on 02/17/2025).

[8] Martin A. Fischler and Robert C. Bolles. "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography". en. In: *Communications of the ACM* 24.6 (June 1981), pp. 381–395. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/358669.358692. URL: https://dl.acm.org/doi/10.1145/358669.358692 (visited on 03/24/2025).

[9] *gazebo_ros_pkgs/gazebo_plugins/worlds/gazebo_ros_skid_steer_drive_demo.world at ros2 · ros-simulation/gazebo_ros_pkgs*. URL: https://github.com/ros-simulation/gazebo_ros_pkgs/blob/ros2/gazebo_plugins/worlds/gazebo_ros_skid_steer_drive_demo.world (visited on 04/09/2025).

[10] Peter Hart, Nils Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. ISSN: 0536-1567. DOI: 10.1109/TSSC.1968.300136. URL: http://ieeexplore.ieee.org/document/4082128/ (visited on 03/31/2025).

[11] Wolfgang Hess et al. "Real-time loop closure in 2D LIDAR SLAM". en. In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. Stockholm, Sweden: IEEE, May 2016, pp. 1271–1278. ISBN: 978-1-4673-8026-3. DOI: 10.1109/ICRA.2016.7487258. URL: http://ieeexplore.ieee.org/document/7487258/ (visited on 03/17/2025).

[12] *Lua configuration reference documentation — Cartographer ROS documentation*. URL: https://google-cartographer-ros.readthedocs.io/en/latest/configuration.html (visited on 04/09/2025).

[13] *Nav2 — Nav2 1.0.0 documentation*. URL: https://docs.nav2.org/ (visited on 02/27/2025).

[14] *Navigation Concepts — Nav2 1.0.0 documentation*. URL: https://docs.nav2.org/concepts/index.html#concepts (visited on 02/27/2025).

[15] *OS1: High-Res Mid-Range Lidar Sensor for Automation & Security | Ouster*. URL: https://ouster.com/products/hardware/os1-lidar-sensor (visited on 04/09/2025).

[16] Patel. *Implementation notes*. URL: https://theory.stanford.edu/~amitp/GameProgramming/ImplementationNotes.html (visited on 04/01/2025).

[17] Patel. *Introduction to A**. URL: https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html (visited on 04/01/2025).

[18] *Point Cloud Library (PCL): pcl::CropBox< pcl::PCLPointCloud2 > Class Reference*. URL: https://pointclouds.org/documentation/classpcl_1_1_crop_box_3_01pcl_1_1_p_c_l_point_cloud2_01_4.html (visited on 04/09/2025).

[19] Sebastian Raschka and Vahid Mirjalili. "Locating regions of high density via DBSCAN". eng. In: *Python machine learning: machine learning and deep learning with Python, scikit-learn, and TensorFlow 2*. 3rd ed. Birmingham: Packt Publishing, 2020, pp. 376–383. ISBN: 978-1-78995-575-0 978-1-78995-829-4.

[20] *ros-perception/pointcloud_to_laserscan: Converts a 3D Point Cloud into a 2D laser scan.* URL: https://github.com/ros-perception/pointcloud_to_laserscan (visited on 04/09/2025).

[21] *Setting Up Navigation Plugins — Nav2 1.0.0 documentation*. URL: https://docs.nav2.org/setup_guides/algorithm/select_algorithm.html#select-algorithm (visited on 03/05/2025).

[22] *TurtleBot3*. URL: https://emanual.robotis.com/docs/en/platform/turtlebot3/features/#features (visited on 04/09/2025).

[23] *velodyne_simulator/velodyne_description/urdf/VLP-16.urdf.xacro at master · lmark1/velodyne_simulator*. URL: https://github.com/lmark1/velodyne_simulator/blob/master/velodyne_description/urdf/VLP-16.urdf.xacro (visited on 04/09/2025).

[24] Stavros G. Vougioukas. "Agricultural Robotics". en. In: *Annual Review of Control, Robotics, and Autonomous Systems* 2.1 (May 2019), pp. 365–392. ISSN: 2573-5144, 2573-5144. DOI: 10.1146/annurev-control-053018-023617. URL: https://www.annualreviews.org/doi/10.1146/annurev-control-053018-023617 (visited on 04/25/2025).