Algorithm for Autonomous Navigation in Vineyards

Author: **Veronica Carini**

Academic Year: 2021-22

Contents

1 | How to run the software

In this pdf, I will explain the functionality of each node that composes the navigation software. Given that it uses many parameters that have to be set by the user, I will report them with their explanation. The software can be used both in simulation or in a real vineyard, but different configurations are needed.
To run the algorithm in simulation, the following commands have to be run:

- roslaunch navigation nav_end_row.launch : this command loads a world in Gazebo and the Scout base. The world can be changed into the launch file. Also, a Rviz configuration should open;
- roslaunch navigation nav_sim.launch : this launch file includes all the nodes needed for navigation, the ActionServer, and also the robot tfs that are necessary to show the Scout base. Notice that the rqt_reconfigure node is invoked here.

Instead, if you want to use the robot in a real environment, you need to execute the following:

- roslaunch plant-phenotyper only_ouster_ugv.launch : it contains all the commands to use the needed sensors. In our case, only the use of LiDAR is mandatory;
- roslaunch navigation nav_real.launch : it contains the nodes used in the real scenario. It is different with respect to the *nav_sim.launch* launch file because the odometry calculations have to be made differently. Also, I kept the handler nodes separated.

2 | Parameters

In this section, an explanation of the parameters is reported. Notice that they are grouped based on the node that uses them.

1. **Controller node**: it is used to collect vineyard info and start the execution.

   Parameters:

   - start_cmd: makes the robot start. It has to be checked when all the parameters have been set;
   - path_from_file: if checked, the path that the robot will follow has to be taken from a .txt file located in the folder *paths* in the navigation package;
   - initial_turning_side: it can be used to set on which side the robot has to turn the first time it reaches the end of a corridor. It can also be used to modify at runtime the next turning side;
   - corridor_width: width of a corridor measured from pole to pole;
   - row_dim: width of a plant row;
   - row_length: minimum length of a corridor. It is better to round down this value by a couple of meters because after traveling this length the in-row navigation node will start checking for the end of the current corridor.

2. **Filter node**: it is in charge of filtering operations to perform over point clouds.

   Parameters:

   - points_height: maximum height beyond which the points will be filtered out. The zero altitude corresponds to the LiDAR sensor's altitude;
   - points_height_start: minimum height from which points have to be maintained;
   - lateral_dist: used during end-row navigation. Maintained points on the free

side[1] of the robot will be at this maximum distance;

- r: radius around the robot within which points are maintained;
- downsampling_cube: side of the cube used for downsampling;
- outliers_filtering: if checked, the outliers points will be filtered in the input point cloud;
- meanK: parameter used for the outliers filtering. It indicates how many points have to be considered to calculate the mean distance from a selected one;
- stddevMulThresh: parameter used for the outliers filtering. It is used to determine the threshold within which the filter will maintain points.

3. **In-row ActionServer**: it is invoked when the robot has to navigate into a corridor and reach its end.

   Parameters:

   - p_gain: proportional gain of the PID controller while navigating in-row;
   - i_gain: integral gain of the PID controller while navigating in-row;
   - d_gain: derivative gain of the PID controller while navigating in-row;
   - use_speed_control: if checked, the robot's speed is adjusted while navigating, taking care of obstacles and elements in the proximity of the sensor;
   - speed_low: minimum speed that the robot can have while navigating into a corridor;
   - speed_high: maximum speed that the robot can reach in a corridor;
   - end_line_meters_threshold: how many meters the robot has to exit from a corridor before terminating the current action;
   - todo_rows: how many corridors must be traveled before stopping;
   - slow_down_obstacle: if checked, the presence of something in front of the robot makes it slow down;
   - distance_obstacle: if at this distance an obstacle is present and *slow_down_obstacle* is enabled, the robot slows down;

[1]free side: when in the field portion that is at the end of all the corridors, the robot will have on one side all the plant rows and the corridors, and on the other one, it will have space that is not important for its purpose.

- ray_length: length of the cone side where free spaces are identified;
- max_lateral_rect_dist: the maximum distance that the robot can have from a plant wall when it is into a corridor. This value has to take care also of holes, so it is recommended to set it at $corridor\_width/2$ + 20% of $corridor\_width$;
- min_cone_width: minimum width of a free-space cone to be accepted;
- max_cone_width: maximum width of a free-space cone to be accepted;
- multi_cone_enabled: if checked, it allows to choose lateral cones when the central one is not accepted;
- obstacle_angle: given an object, it has to occupy a certain amplitude of angle to be considered as an obstacle;
- min_obs_density: given an object that has been identified as an obstacle, the points that constitute it need to have this density to confirm its classification as an obstacle;
- sensor_min_range: the minimum distance an object has to have from the sensor to be seen.

4. **Turn ActionServer**: it manages the rotate-in-place robot operations.

   Parameters:

   - use_personalized_angles: if checked, the *exit_ turning_ angle* and *entrance_turning_ angle* values are used to know how much to rotate. Otherwise, the rotation is 90°;
   - exit_turning_angle: angle to rotate when exiting a corridor;
   - entrance_turning_angle: angle to rotate to enter a new corridor;
   - fixed_speed: angular speed used to rotate in place.

5. **End-row ActionServer**: it is invoked when the robot has to change row and manages all the maneuvers to enter a new corridor.

   Parameters:

   - start: end-row navigation starts when this flag is checked;
   - p_gain: proportional gain of the PID controller while navigating at end-row;
   - i_gain: integral gain of the PID controller while navigating at end-row;

- d_gain: derivative gain of the PID controller while navigating at end-row;
- use_speed_control: if checked, the robot's speed is adjusted while navigating, taking care of obstacles and elements in the proximity of the sensor;
- speed_low: minimum speed that the robot can have while navigating perpendicular to corridors;
- speed_high: maximum speed that the robot can reach while it is perpendicular to corridors;
- correction_angle: if two subsequent segments have a very different slope, it is necessary to navigate straight for some centimeters before changing the orientation, to avoid oscillation problems. The *correction_angle* parameter indicates the dimension of the gap between the two segments' slope that creates this problem;
- corridors_to_skip: indicates how many corridors have to be skipped before entering a new one;
- correction_speed: this speed is used while reaching the middle point of the last segment. It is recommended to use a lower speed with respect to the *speed_high* one not to overshoot the new corridor entrance;
- pole_radius: the head of a plant row can be composed of a pole and some branches or fruits. The projection of these elements on the ground must be contained in a circle of radius *pole_radius*;
- use_barycenter_as_point: if checked, to calculate the segment that indicates to the robot the slope to assume, clusters' barycenters can be used as segment's ends.

6. **Clustering ActionServer**: when clustering operations are needed to recognize a row, the services of this ActionServer are requested.

   Parameters:

   - seg_radius: indicates the minimum distance between two different clusters. If two points have a distance that is less than *seg_radius*, they are associated with the same cluster;
   - point_neighborhood_check: if checked, an analysis of the neighborhood of points is performed to find the best candidate as a pole;
   - r: radius of the neighborhood;

- K: the first nearest K points neighborhood of the cluster are analyzed;
- min_near_points: a point is chosen as a candidate if it has at least *min_near_points* points in its neighborhood;
- use_RL_model: if checked, the plant rows will be approximated with a line using the linear regression method;
- use_RANSAC_model: if checked, the plant rows will be approximated with a line using the RANSAC method;
- threshold: useful when using RANSAC, a point is considered an inlier for the line if it is not farther than *threshold*;
- iterations: how many times the RANSAC algorithm can be repeated to obtain a line.

3 | Packages

Here the nodes are explained for every package. Also, a comment about file organization in each folder is reported. Notice that the important nodes for the software architecture are highlighted in bold.

3.0.1. odometry package

In the *src* folder, the following .cpp files can be found:

- **rpm_converter.cpp** : this node takes as input a *scout_status* message and uses the RPM data to calculate speeds of wheels on the left and right sides;
- **odometry.cpp** : this node calculates the current Odometry message for the robot. To do so, it takes in input the estimated parameters (alpha and minus_y_icr) and the left and right speeds of the robot's wheels;
- **odometry_sim.cpp** : Gazebo publishes information about speeds and positions. This node listens to the info published by Gazebo and converts them into an Odometry message. It has to be used in simulation instead of odometry.cpp node;
- param_estimator.cpp : this node has been used to estimate the parameters alpha and minus_y_icr used for odometry calculation;
- position.cpp : support library used by param_estimator.cpp node for estimation purposes;
- odom_comparator.cpp : node used for testing. It allows us to compare Odometry messages coming from different sources to evaluate the differences between them;
- gpose_to_pose.cpp : it converts the Pose2D message published by OptiTrack into a PoseStamped message that can be handled by other nodes. Used for parameter estimation only.

Notice that in the paper that explains our odometry model, the parameters were α and x_{ICR}. Due to different axes conventions, our equivalent of x_{ICR} is $-y_{ICR}$ (minus_y_icr).

In the package, there is a .txt file that explains the steps to follow to perform a good test for parameter estimation. Also, *param_calc.ods* shows some values we obtained during the estimation phase for the parameters.

Finally, notice that in the *launch* folder, an *odometry.launch* file is present other than two files used for tfs both in simulation and in reality (*tf_publisher.launch* and *tf_publisher_sim.launch*).

3.0.2. navigation package

In the navigation package, there is a folder named *paths*: it contains .txt files where a path for the robot to follow can be specified. When we choose the *path_from_file* option in the dynamic reconfigure window, the software will look for a file named *path1.txt* in this subdirectory.

Another relevant folder is the *world* one. Here the Gazebo worlds used for testing in the simulation are present.

Launch files have been already mentioned in Chapter 1, while the relevant nodes in *src* folder are:

- **controller.cpp** : this node is in charge of publishing parameters obtained through the dynamic reconfigure on topics so that different nodes can have access to them;
- robot.cpp : it is a class that represents the object robot with some parameters and useful methods to check its data;
- **filter_node.cpp** : this node listens to parameters given by the user and also to filtering policies defined by the handler and applies them to each PointCloud2 message received. To perform filtering operations, a library in the *pc_filters* package is used;
- **sim_handler.cpp** : this node is the handler when we are using the software in simulation. It is in charge of managing the requests to the different ActionServers and to do so it follows a scheme: when a new goal needs to be created it takes the correct ActionClient and calls on it a method to send a request. The handler is the one that keeps track of in which vineyard zone the robot is and it calls ActionServers accordingly. Knowing the robot's position, it also knows which filter policies have to be applied at point clouds, so it is in charge of publishing filtering instructions. It also updates the next turning side;
- **real_handler.cpp** : this node performs the same operations as the previous one, but it has to be used when we are working in a real environment.

3.0.3. pc_filters package

As mentioned before, this package contains a library used for filtering purposes. In particular, the file *filters.cpp* contains the methods invoked by the *filter_node.cpp* node to filter point clouds.

3.0.4. fre_row_navigation package

This package contains the nodes responsible for the in-row navigation. It is a clone of the GitHub repository made available after FRE21. Only some modifications have been performed.

In the *src* folder, you will find the following nodes:

- **client.cpp** : this node represents the ActionClient that can be instantiated by the handler. The *start_client()* method can be used to prepare a goal and send it to the server, then the client waits for a result;
- **crawl_row_node.cpp** : this node calculates linear and angular speeds that the robot has to assume to navigate in-row. The first thing that it does is to find a cone of free space in front (or on the sides) of the robot. Then it calculates the distances that the robot has from the plants on the sides using two rectangles, to do so it grows their sides until a certain number of points is into each rectangle. At this point, the check about the end of the row is performed. If the end row has been found and the robot has already traveled out enough, then it can stop. Otherwise, the node calculates the linear and angular speeds that have to be assumed to perform navigation. To do so, the PID object is used. Notice that each time a cone or a rectangle is calculated, it is also drawn as a marker;
- **cone.cpp** : this class is used to perform operations with cones. Its methods are used to find a cone in a given LaserScan message, to return a point that lies on the middle line of the cone, and to draw it;
- **rectangle.cpp** : this class is similar to the previous one because it is used to manage all the operations that have to be done with rectangles. The methods here allow to count the number of points inside a rectangle, grow a chosen rectangle side, and create a visual marker for that rectangle;
- **pid.cpp** : a class that implements a PID controller and allows to use it;
- pid_tester.cpp : this node has been used to tune the PID controller. In particular, it allows to register the PID's output in front of a step input function.

3.0.5. turn package

In the *src* folder we have two nodes:

- **client.cpp** : this node is the implementation of an ActionClient that is used by the handler to send requests to the turn ActionServer;
- **turn_action.cpp** : this is the just mentioned ActionServer. It calculates an angular velocity to make the robot turn on the spot. The rotation angle can be fixed or given by the user. To know how much the robot has already rotated, the Odometry message is used. Notice that a hypothesis has been made: a robot cannot rotate more than 90 degrees between two subsequent Odometry messages. This hypothesis allows us to eliminate the problem given by the fact that the robot oscillates a bit while rotating.

3.0.6. end_row_navigation package

In this package *src* folder there are two nodes:

- client.cpp : this node implements the ActionClient for the ActionServer that is in charge of elaborating the linear and angular speeds for end-row navigation. The *start_client*() method is called on a client object by the handler;
- parallel_line_navigation.cpp : this node is the ActionServer in charge of end-row navigation operations. The first thing it has to do when a new goal is received is to request the services of the clustering ActionServer. Then, this node operates using callbacks for messages coming from the clustering ActionServer in which the last pole of each row is specified. Given that the clustering ActionServer returns poles points or barycenters, the node has to manage the received output distinguishing between these two cases. To do so, the methods *clustersCb*(), *barycentersCb*(), and *store_points*() are used. At the same frequency with which point clouds are received, the node analyzes the content of the last received message (method *processClusters*()). In this method, the following operations are performed:
  1. the two nearest points are found: they represent the two poles that are one in front and one behind the robot (method *find_two_points*());
  2. the points are drawn and the poles are counted;
  3. the slope (angular coefficient) of the segment that connects the two chosen points is calculated;

4. linear and angular speeds are calculated accordingly to the segment's slope and the presence of obstacles in front of the robot;

5. if the new slope is very different w.r.t. the precedent one, then the robot has to move straight for a little space, thus avoiding problems with the reference points;

6. if we have counted the correct number of poles, it means the robot has to start aligning with the middle point of the last segment, so the middle point is calculated;

7. if the robot is aligning with the middle point, then we have to check if it has reached that point. If the sensor is in a neighborhood of the point with a radius equal to 10 centimeters, then the robot has reached the middle point and stops. The node notifies the clustering ActionServer to stop the clustering operations and terminates the goal.

Notice that a lot of support methods are present to perform different things, such as the choice of two points and the count of poles. Furthermore, given that while we elaborate on a response the robot moves, it is important to convert the pole points received in the odom frame, and then convert them in the sensor frame before using them. The same thing has to be done for the middle point.

3.0.7. pcl_assembler package

This package is in charge of clustering operations. It also has to choose, for each cluster, a point that represents the last element in the row (usually referred to as a pole). In the *src* folder, four nodes are present:

- pcl_normal_filter.cpp : an old version of the clustering algorithm that uses the DoN-based clustering. It is no more used;

- **client.cpp** : this ActionClient is used to send a goal to the clustering ActionServer. Its *get_clusters*() method is invoked by the end-row navigation ActionServer;

- **least_square.cpp**: a library to perform operations related to linear regression. Due to the structure of methods, some of them can be used also when a RANSAC model is preferred;

- **pcl_euclidean_dist.cpp**: this ActionServer is in charge of clustering operations. The clustering operations are performed at a frequency of 2 Hz, so the *find_clusters*() method is called only once every five received point clouds. Once the method is called, it applies Euclidean Cluster Extraction to a KdTree to find clusters. Then, it iterates over clusters, and for each one of them, it has to calculate the barycenter and choose a point that represents a pole, depending on the policy chosen by the user. If the neighborhood check has to be done, the (0, 0, 0) point, which represents the sensor, is assumed as the point to which the checked ones have to be close. The neighborhoods of the K nearest points to (0, 0, 0) are checked. If no one satisfies the condition, then the nearest point to (0, 0, 0) is returned. Then, if necessary, a linear regression model or a RANSAC one has to be calculated and the chosen pole has to be projected onto it. The projection point will be returned as a result. When all clusters have been analyzed, the *find_clusters*() method can terminate.

3.0.8. messages package

It contains all the user-defined messages used by the software.

3.0.9. speed_controller package

This package contains a single node that allows setting linear and angular speeds that the robot must assume. It is useful when you want to set a precise speed and register bags.

3.0.10. Other packages

Here are listed the packages that I have downloaded from the net and adjusted for my needs:

- ira_laser_tools package: this package contains useful nodes to convert a PointCloud2 message into a LaserScan one;
- lidar_simulator package: it contains the OS1 LiDAR simulated version;
- scout_gazebo, scout, scout_ros, ugv_sdk packages: useful packages to manage the Scout base both in simulation and in the reality. They are used by the already mentioned (Chapter 1) launch files.