AUTO DOCKING

PROJECT OVERVIEW

This project integrates pattern recognition, control logic and line extraction for a robotic system using the Robot Operating System (ROS2- Humble). It aims to enable a robot to autonomously navigate environments and perform docking operations by detecting patterns, making decisions based on these patterns, and accurately mapping surroundings using laser scan data.

OVERALL FLOW

**ROS 2 Launch System Setup (docking.py):**

* The auto-docking process is initiated through the ROS 2 launch system, defined in docking.py. Launch description includes setup for launching nodes related to laser line extraction, auto docking pattern detection and auto docking control.
* Generates a ‘LaunchDescription’ object (ld) with configured nodes and launch arguments.

**Laser Line Extraction (laser\_line\_extraction\_node):**

* Purpose: Extracts lines from laser scan data to identify environmental features.
* Implementation: Implemented in ‘laser\_line\_extraction\_node’ from ‘laser\_line\_extraction’ package. Parameters configured include frequency, frame ID (‘base\_scan’), scan topic (‘scan;), and various line extraction thresholds.
* Input Information: Receives laser scan data (‘sensor\_msgs/LaserScan messages’) from the robot's sensors.
* Output Information: Publishes line segments (‘laser\_line\_msgs/LineSegment messages’) and optionally visualization markers.

**Auto Docking Pattern Detection (‘pattern\_node’):**

* Purpose: Detects specific patterns related to auto-docking, leveraging the extracted lines.
* Implementation: Implemented in ‘pattern\_node’ from ‘auto\_dock’ package. Parameters include angle and distance tolerances for pattern detection.
* Input Information: Receives line segments (‘laser\_line\_msgs/LineSegment’ messages) from ‘laser\_line\_extraction\_node’.
* Output Information: Outputs detected patterns or relevant information about potential docking locations.

**Auto Docking Controller (‘controller\_node’):**

* Purpose: Executes control logic to guide the robot to dock at the detected pattern.
* Implementation: Implemented in ‘controller\_node’ from ‘auto\_dock’ package. Controls robot's velocity and orientation based on relative positions and orientations to the docking pattern.
* Input Information: Receives information about the detected pattern (position and orientation) from ‘pattern\_node’.
* Output Information: Sends velocity commands (‘geometry\_msgs/Twist messages’) to drive the robot towards the docking location. Publishes status updates and logs regarding the docking process.

Data Flow:

* Laser scan data is processed by ‘laser\_line\_extraction\_node’ to extract line segments.
* These segments are analyzed by ‘pattern\_node’ to identify suitable docking patterns.

‘controller\_node’ then uses this information to guide the robot towards the docking location.

COMPONENTS

**Pattern Recognition Module:**

* Detects specific patterns in sensor data, such as objects, landmarks, or conditions.
* Provides pattern indices or information to the control logic module.

**Control Logic Module:**

* Receives input from the pattern recognition module.
* Executes decision-making algorithms to control the robotic system's behavior based on detected patterns.
* Manages navigation, task execution, and responses to environmental changes.

**Line Extraction Module:**

* Processes laser scan data to detect and extract lines representing obstacles or features in the environment.
* Provides accurate geometric representations of detected lines for navigation and mapping purposes.

**Integration:**

* Utilizes ROS2 nodes for communication between modules.
* Publishes and subscribes to topics for exchanging sensor data, control commands, and line extraction results.
* Facilitates visualization and monitoring through ROS2 messages and visualization tools.

**Docking.py**

1. import os
2. from pathlib import Path
3. from ament\_index\_python.packages import get\_package\_share\_directory
4. from launch import LaunchDescription
5. from launch.actions import DeclareLaunchArgument, IncludeLaunchDescription, ExecuteProcess
6. from launch.conditions import IfCondition
7. from launch.launch\_description\_sources import PythonLaunchDescriptionSource
8. from launch.substitutions import LaunchConfiguration
9. from launch\_ros.actions import Node
10. from launch.actions import OpaqueFunction
11. def generate\_launch\_description():
12. # Declare launch arguments
13. declare\_open\_rviz\_arg = DeclareLaunchArgument
14. ( 'open\_rviz',
15. default\_value='true',
16. description='Open RViz automatically’ )
17. open\_rviz = LaunchConfiguration('open\_rviz')
18. def launch\_setup(context, \*args, \*\*kwargs):
19. # Conditional RViz node launch
20. rviz\_node = Node
21. ( package='rviz2'
    1. executable='rviz2',
    2. name='rviz',
    3. arguments=['-d', os.path.join(get\_package\_share\_directory('auto\_dock'), 'rviz', 'rviz')],
    4. output='screen')
22. if open\_rviz.perform(context) == 'true' else None
    1. return [
    2. # Uncomment this block if you need to use static\_transform\_publisher
    3. # ExecuteProcess(
    4. # cmd=['ros2', 'run', 'tf2\_ros', 'static\_transform\_publisher', '0', '0', '0', '0', '0', '0', 'map'
    5. 'laser'],
    6. # output='screen'
    7. # )
23. # Node for laser line extraction
24. Node (
    1. package='laser\_line\_extraction',
    2. executable='laser\_line\_extraction\_node',
    3. name='laser\_line\_extraction',
    4. output='screen',
    5. parameters= [{
    6. "frequency": 50.0,
    7. "frame\_id": "base\_scan",
    8. "scan\_topic": "scan",
    9. "publish\_markers": True,
    10. "bearing\_std\_dev": 0.0015,
    11. "range\_std\_dev": 0.01,
    12. "least\_sq\_angle\_thresh": 0.0001,
        1. "least\_sq\_radius\_thresh": 0.0001,
        2. "max\_line\_gap": 0.5,
        3. "min\_line\_length": 0.03,
        4. "min\_range": 0.2,
        5. "min\_split\_dist": 0.05,
        6. "outlier\_dist": 0.05,
        7. "min\_line\_points": 5,
    13. }]
    14. ),
25. # Node for auto docking pattern detection
    1. Node(
    2. package='auto\_dock',
    3. executable='pattern',
    4. name='pattern\_node',
    5. output='screen',
    6. parameters=[{
       1. "detect\_angle\_tolerance": 0.2,
       2. "group\_dist\_tolerance": 0.15,
       3. "laser\_frame\_id": "laser\_frame",
    7. }]
    8. ),
26. #Node for auto docking controller

Node (

* + 1. package='auto\_dock',
    2. executable='controller',
    3. name='controller\_node',
    4. output='screen',
    5. parameters=[{
    6. "dist\_to\_dock": 0.22,
    7. "dist\_to\_center": 0.03,
    8. }]

),

1. # Node for TurtleBot3 navigation # Conditional RViz node

rviz\_node

1. ]
2. # Launch description
3. ld = LaunchDescription([
4. declare\_open\_rviz\_arg,
5. OpaqueFunction(function=launch\_setup)
6. ])
7. return ld

**EXPLANATION**

* **(1-10)Imports**: Imports necessary modules and functions for handling launch configurations and ROS nodes.
* **(11)generate\_launch\_description() Function**:

**Launch Setup Function (launch\_setup())**: This function is used to configure and return a list of nodes and processes to be launched based on the context provided.

**Conditional RViz Node**: Conditionally launches an RViz node (rviz\_node) if open\_rviz is set to 'true'. This node loads an RViz configuration file located in the auto\_dock package directory.

* **Nodes**:

**(24)Laser Line Extraction Node**: Launches a node (laser\_line\_extraction\_node) from the laser\_line\_extraction package, which extracts lines from laser scan data. It is configured with various parameters related to line extraction.

**(25) Docking Pattern Detection Node**: Launches a node (pattern\_node) from the auto\_dock package, responsible for detecting patterns related to auto docking. It is configured with parameters such as detect\_angle\_tolerance, group\_dist\_tolerance, and laser\_frame\_id.

**(26)Auto Docking Controller Node**: Launches a node (controller\_node) from the auto\_dock package, which controls the auto docking process. It is configured with parameters like dist\_to\_dock and dist\_to\_center.

* **(30)Launch Description (ld)**: Constructs a LaunchDescription object containing The declare\_open\_rviz\_arg launch argument declaration. An Opaque Function that executes launch\_setup() to gather all configured nodes and processes.

### **Output :** This script outputs a LaunchDescription object (ld) that defines the entire launch configuration for launching nodes related to laser line extraction, auto docking pattern detection, and auto docking control. Launches nodes for laser line extraction (laser\_line\_extraction\_node), auto docking pattern detection (pattern\_node), and auto docking controller (controller\_node).

**line\_extraction\_node.cpp**

It is the main entry point and class definition for a ROS 2 node that performs line extraction from laser scan data.

**Line\_extraction\_node.h**

1. #ifndef LINE\_EXTRACTION\_ROS\_H
2. #define LINE\_EXTRACTION\_ROS\_H
3. #include <vector>
4. #include <string>
5. #include "rclcpp/rclcpp.hpp"
6. #include "sensor\_msgs/msg/laser\_scan.hpp"
7. #include "visualization\_msgs/msg/marker.hpp"
8. #include "geometry\_msgs/msg/point.hpp"
9. #include "laser\_line\_msgs/msg/line\_segment\_list.hpp"
10. #include "laser\_line\_msgs/msg/line\_segment.hpp"
11. #include "line.h"
12. #include "line\_extraction.h"
13. namespace line\_extraction
14. {
15. class LineExtractionROS : public rclcpp::Node
16. {
17. public:
    1. LineExtractionROS(): Node("line\_extraction")
    2. {
    3. // Parameters used by this node
    4. this->declare\_parameter<bool>("data\_cached", false);
    5. this->get\_parameter("data\_cached", data\_cached\_);
    6. this->declare\_parameter<std::string>("frame\_id", "base\_scan");
    7. this->get\_parameter("frame\_id", frame\_id\_);
    8. RCLCPP\_INFO(this->get\_logger(), "[%s] frame\_id = %s", \_\_func\_\_, frame\_id\_.c\_str());
    9. this->declare\_parameter<std::string>("scan\_topic", "scan");
    10. this->get\_parameter("scan\_topic", scan\_topic\_);
    11. RCLCPP\_INFO(this->get\_logger(), "[%s] scan\_topic = %s", \_\_func\_\_, scan\_topic\_.c\_str());
    12. this->declare\_parameter<bool>("publish\_markers", true);
    13. this->get\_parameter("publish\_markers", pub\_markers\_);
    14. RCLCPP\_INFO(this->get\_logger(), "[%s] publish\_markers = %d", \_\_func\_\_, pub\_markers\_);
    15. // this->declare\_parameter<float>("frequency", 1.0);
    16. // this->get\_parameter("frequency", frequency\_);
    17. // RCLCPP\_INFO(this->get\_logger(), "[%s] frequency = %f", \_\_func\_\_, frequency\_);
    18. // Parameters used by the line extraction algorithm
    19. this->declare\_parameter<double>("bearing\_std\_dev", 1e-3);
    20. this->get\_parameter("bearing\_std\_dev", bearing\_std\_dev);
    21. RCLCPP\_INFO(this->get\_logger(), "[%s] bearing\_std\_dev = %f", \_\_func\_\_, bearing\_std\_dev);
    22. this->declare\_parameter<double>("range\_std\_dev", 0.02);
    23. this->get\_parameter("range\_std\_dev", range\_std\_dev);
    24. RCLCPP\_INFO(this->get\_logger(), "[%s] range\_std\_dev = %f", \_\_func\_\_, range\_std\_dev);
    25. this->declare\_parameter<double>("least\_sq\_angle\_thresh", 1e-4);
    26. this->get\_parameter("least\_sq\_angle\_thresh", least\_sq\_angle\_thresh);
    27. RCLCPP\_INFO(this->get\_logger(), "[%s] least\_sq\_angle\_thresh = %f", \_\_func\_\_, least\_sq\_angle\_thresh);
    28. this->declare\_parameter<double>("least\_sq\_radius\_thresh", 1e-4);
    29. this->get\_parameter("least\_sq\_radius\_thresh", least\_sq\_radius\_thresh);
    30. RCLCPP\_INFO(this->get\_logger(), "[%s] least\_sq\_radius\_thresh = %f", \_\_func\_\_, least\_sq\_radius\_thresh);
    31. this->declare\_parameter<double>("max\_line\_gap", 0.4);
    32. this->get\_parameter("max\_line\_gap", max\_line\_gap);
    33. RCLCPP\_INFO(this->get\_logger(), "[%s] max\_line\_gap = %f", \_\_func\_\_, max\_line\_gap);
    34. this->declare\_parameter<double>("min\_line\_length", 0.03);
    35. this->get\_parameter("min\_line\_length", min\_line\_length);
    36. RCLCPP\_INFO(this->get\_logger(), "[%s] min\_line\_length = %f", \_\_func\_\_, min\_line\_length);
    37. this->declare\_parameter<double>("min\_range", 0.4);
    38. this->get\_parameter("min\_range", min\_range);
    39. RCLCPP\_INFO(this->get\_logger(), "[%s] min\_range = %f", \_\_func\_\_, min\_range);
    40. this->declare\_parameter<double>("min\_split\_dist", 0.05);
    41. this->get\_parameter("min\_split\_dist", min\_split\_dist);
    42. RCLCPP\_INFO(this->get\_logger(), "[%s] min\_split\_dist = %f", \_\_func\_\_, min\_split\_dist);
    43. this->declare\_parameter<double>("outlier\_dist", 0.05);
    44. this->get\_parameter("outlier\_dist", outlier\_dist);
    45. RCLCPP\_INFO(this->get\_logger(), "[%s] outlier\_dist = %f", \_\_func\_\_, outlier\_dist);
    46. this->declare\_parameter<int>("min\_line\_points", 5);
    47. this->get\_parameter("min\_line\_points", min\_line\_points);
    48. RCLCPP\_INFO(this->get\_logger(), "[%s] min\_line\_points = %d", \_\_func\_\_, min\_line\_points);
    49. scan\_subscriber\_ = this->create\_subscription<sensor\_msgs::msg::LaserScan>(scan\_topic\_, rclcpp::QoS(rclcpp::KeepLast(10)).best\_effort().durability\_volatile(), std::bind(&LineExtractionROS::scan\_cb, this, std::placeholders::\_1));
    50. line\_publisher\_ = this->create\_publisher<laser\_line\_msgs::msg::LineSegmentList>("line\_segments", 1);
    51. marker\_publisher\_ = this->create\_publisher<visualization\_msgs::msg::Marker>("line\_markers", 1)
18. }
19. void run();
20. private:
    1. LineExtraction line\_extraction\_;
    2. bool data\_cached\_, pub\_markers\_;
    3. std::string frame\_id\_;
    4. std::string scan\_topic\_;
    5. // float frequency\_;
    6. double bearing\_std\_dev, range\_std\_dev, least\_sq\_angle\_thresh, least\_sq\_radius\_thresh, max\_line\_gap, min\_line\_length, min\_range, min\_split\_dist, outlier\_dist;
    7. int min\_line\_points;
    8. void populateLineSegListMsg(const std::vector<Line>&, laser\_line\_msgs::msg::LineSegmentList&);
    9. void populateMarkerMsg(const std::vector<Line>&, visualization\_msgs::msg::Marker&);
    10. void scan\_cb(const sensor\_msgs::msg::LaserScan::SharedPtr scan);
    11. void CachedData(const sensor\_msgs::msg::LaserScan::SharedPtr scan);
    12. rclcpp::Subscription<sensor\_msgs::msg::LaserScan>::SharedPtr scan\_subscriber\_;
    13. rclcpp::Publisher<laser\_line\_msgs::msg::LineSegmentList>::SharedPtr line\_publisher\_;
    14. rclcpp::Publisher<visualization\_msgs::msg::Marker>::SharedPtr marker\_publisher\_;
21. };
22. }
23. #endif

**EXPLANATION**

* (3-12)Includes necessary headers: Includes standard library headers and ROS 2 message headers.
* (13)Namespace line\_extraction: Defines a namespace to avoid name conflicts.
* (15)Class definition: Defines the LineExtractionROS class

**LineExtractionROS Class** :

* + Constructor: Declares and retrieves parameters, sets up subscriptions and publishers.
  + Parameters: Includes various parameters for the line extraction algorithm and node configuration.
  + Subscriptions: Subscribes to the scan topic to receive sensor\_msgs::msg::LaserScan messages.
  + Publishers: Publishes extracted line segments and markers.

**Member Functions:**

* + run(): Main loop for running the line extraction process.
  + populateLineSegListMsg(): Populates a LineSegmentList message with extracted lines.
  + populateMarkerMsg(): Populates a Marker message for visualization.
  + scan\_cb(): Callback for processing incoming laser scan messages.
  + CachedData(): Caches incoming laser scan data if required.
  + Private members: Defines private member variables and methods for internal use.

**line\_extraction\_node.cpp**

1. #include "laser\_line\_extraction/line\_extraction\_node.h"
2. int main(int argc, char\* argv[])
3. {
4. std::shared\_ptr<line\_extraction::LineExtractionROS> line\_extraction\_node;
5. rclcpp::init(argc, argv);
6. line\_extraction\_node = std::make\_shared<line\_extraction::LineExtractionROS>();
7. RCLCPP\_INFO(line\_extraction\_node->get\_logger(), "[%s] line\_extraction\_node has been started", \_\_func\_\_);
8. rclcpp::WallRate rate(50.0);
9. while (rclcpp::ok())
10. {
11. line\_extraction\_node->run();
12. rclcpp::spin\_some(line\_extraction\_node);
13. rate.sleep();
14. }
15. return 0; }

**FLOW**

* Initialization: The node is initialized, parameters are declared and retrieved, and subscriptions and publishers are set up.
* Main Loop: The node enters a loop where it continuously processes incoming laser scan data to extract lines and publish the results.

**EXPLANATION**

Main Function: Initializes the ROS 2 environment and the LineExtractionROS node.Starts the node and processes callbacks in a loop. Entry point of the program.

* (1)Includes the header file: line\_extraction\_node.h is included to use LineExtractionROS class.
* (5)ROS 2 initialization: Initializes the ROS 2 client library.
* (6)Create node instance: Creates an instance of the LineExtractionROS class.
* (7)Log start message: Logs that the node has started.
* (8)Loop rate: Sets the rate at which the loop runs (50 Hz).
* (9)Main loop: Continuously calls the run method, processes incoming messages, and sleeps to maintain the loop rate**.**

**Input**

Laser scan data: Subscribed from a ROS 2 topic (e.g., scan\_topic\_).

**Output**

Line segments: Published to a ROS 2 topic (e.g., line\_segments).

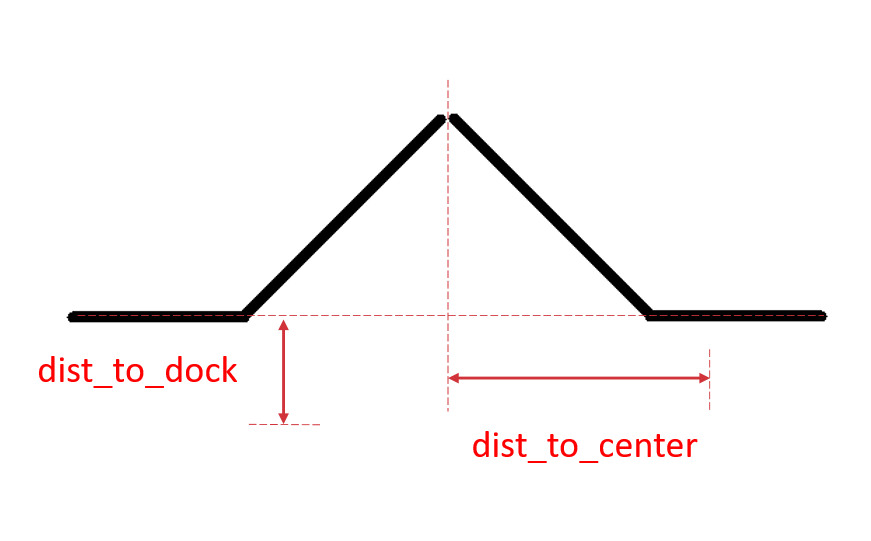
Markers: Published to a ROS 2 topic (e.g., line\_markers).

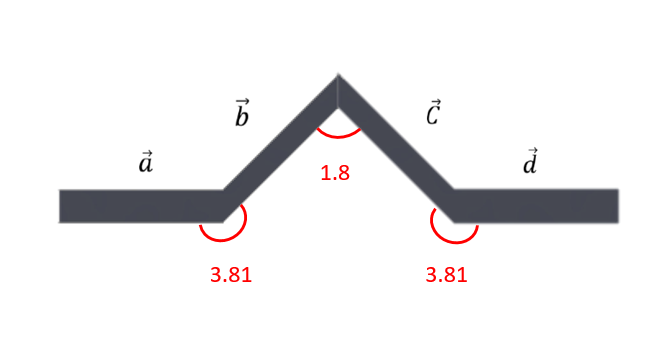
This structure allows the node to process incoming laser scan data, extract line segments, and publish the results for other nodes to us

Pattern Angle Definition

### 

Pattern Parameters Definition





**pattern\_angle1**

Theta 1 as shown in pattern angle definition. Note that the unit is radian.

**pattern\_angle2**

Theta 2 as shown in pattern angle definition. Note that the unit is radian.

**pattern\_angle3**

Theta 3 as shown in pattern angle definition. Note that the unit is radian.

**detect\_angle\_tolerance**

The maximum difference between detected angle and pattern angle(pattern\_angle1, pattern\_angle2, pattern\_angle3)

**group\_dist\_tolerance**

The maximum distance between two line to be recognize as a part of pattern. Such as the distance between the end point of detected vector b and the start point of detected vector a should smaller than group\_dist\_tolerance to be consider as a part of pattern.

**laser\_frame\_id**

The laser frame of the robot.

**base\_frame\_id**

The base frame of the robot.

**min\_v**

Minimum linear velocity of the robot.

**min\_w**

Minimum angular velocity of the robot.

**max\_v**

Maximum linear velocity of the robot.

**max\_w**

Maximum angular velocity of the robot.

**threshold\_v**

The threshold of changing between minimum and maximum linear velocity.

**threshold\_w**

The threshold of changing between minimum and maximum angular velocity.

**dist\_to\_dock**

Distance to the dock frame.

**dist\_to\_center**

Distance to the center.

**CHISQUARED DISTANCE**

* Statistical measure used to assess the dissimilarities between two probability distribution
* Computed based on observed and expected frequency
* Used to show dissimilarities between lines
* It is crucial in deciding whether the lines should merge or not based on similarities and dissimilarities
* Below threshold - merge , Above threshold – doesn’t merge

**EUCLIDEAN DISTANCE**

* Straight line distance between 2 points space
* Represent length of shortest path connecting them
* Used to find the separation between points

**line\_extraction.cpp**

The LineExtraction class is responsible for extracting lines from laser scan data using various filtering, splitting, merging, and fitting techniques. **line\_extraction.h**: Defines the LineExtraction class with methods for setting parameters and extracting lines from laser scan data. Includes necessary libraries and dependencies. **line\_extraction.cpp**: Implements methods to set parameters and perform line extraction algorithms.

**line\_extraction.h**

1. #ifndef LINE\_EXTRACTION\_H
2. #define LINE\_EXTRACTION\_H
3. #include <cmath>
4. #include <vector>
5. #include <boost/array.hpp>
6. #include <eigen3/Eigen/Dense>
7. #include "utilities.h"
8. #include "line.h"
9. namespace line\_extraction
10. {
11. class LineExtraction
12. {
13. public:
14. // Constructor / destructor
15. LineExtraction();
16. ~LineExtraction();
17. // Run
18. void extractLines(std::vector<Line>&);
19. // Data setting
20. void setCachedData(const std::vector<double>&, const std::vector<double>&,
    * 1. const std::vector<double>&, const std::vector<unsigned int>&);
21. void setRangeData(const std::vector<double>&);
22. // Parameter setting
23. void setBearingVariance(double);
24. void setRangeVariance(double);
25. void setLeastSqAngleThresh(double);
26. void setLeastSqRadiusThresh(double);
27. void setMaxLineGap(double);
28. void setMinLineLength(double);
29. void setMinLinePoints(unsigned int);
30. void setMinRange(double);
31. void setMinSplitDist(double);
32. void setOutlierDist(double);
33. private:
34. // Data structures
35. CachedData c\_data\_;
36. RangeData r\_data\_;
37. Params params\_;
38. // Indices after filtering
39. std::vector<unsigned int> filtered\_indices\_;
40. // Line data
41. std::vector<Line> lines\_;
42. // Methods
43. double chiSquared(const Eigen::Vector2d&, const Eigen::Matrix2d&,
    * 1. const Eigen::Matrix2d&);
44. double distBetweenPoints(unsigned int index\_1, unsigned int index\_2);
45. void filterClosePoints();
46. void filterOutlierPoints();
47. void filterLines();
48. void mergeLines();
49. void split(const std::vector<unsigned int>&);
50. };
51. } // namespace line\_extraction
52. #endif

**Explanation**

* (15-16)Constructor and Destructor: Initialize and clean up resources.
* (18)extractLines: Main method to extract lines from data.
* (20-21)Data Setting Methods: Set cached and range data required for processing.
* (23-32)Parameter Setting Methods: Set parameters controlling the line extraction process.
* (33-41)Private Members: Store cached data, range data, parameters, filtered indices, and extracted lines.
* (43-49)Private Members: Internal methods for calculations, filtering, and line processing.

**Input and Output Information**

Input:

* Cached data and range data from sensors.
* Various parameters set by the user or configuration.

Output:

* Extracted lines stored in a vector of Line objects.

This header file defines the interface and private members of the LineExtraction class, which is responsible for extracting lines from sensor data, a critical step in the auto-docking process.

**auto\_dock.h**

1. #include <boost/array.hpp>
2. #include <iostream>
3. #include <cmath>
4. //distance function for middle point array
5. double dist( auto vector\_i , auto vector\_j){
6. float dist = sqrt(pow(fabs(vector\_i[0] - vector\_j[0]),2)+pow(fabs(vector\_i[1] - vector\_j[1]),2));
7. return dist;
8. }
9. //translate line vector to middle point array
10. auto mid\_point(auto vector){
11. boost::array<double, 2> mid ;
12. mid[0] = (vector.start[0]+vector.end[0])/2 ;
13. mid[1] = (vector.start[1]+vector.end[1])/2 ;
14. return mid;
15. }
16. //middle point between two middle point array
17. auto mid\_two\_point( auto array\_a , auto array\_b){
18. boost::array<double, 2> mid ;
19. mid[0] = (array\_a[0] + array\_b[0])/2 ;
20. mid[1] = (array\_a[1] + array\_b[1])/2 ;
21. return mid;
22. }
23. //distance function for two line vector
24. double mid\_dist( auto vector\_i , auto vector\_j){
25. boost::array<double, 2> mid\_i = mid\_point(vector\_i);
26. boost::array<double, 2> mid\_j = mid\_point(vector\_j);
27. double dist = sqrt(pow(fabs(mid\_i[0] - mid\_j[0]),2)+pow(fabs(mid\_i[1] - mid\_j[1]),2));
28. return dist;
29. }

**Explanation**

* (1-3)Includes: Include necessary headers for arrays, input/output operations, and mathematical functions.
* (5-7)Distance Function for Middle Point Array : Calculate the Euclidean distance between two 2D points.**Inputs**: auto vector\_i, auto vector\_j 2D points represented as arrays.**Outputs**: Distance between the points.
* (10-14)Translate Line Vector to Middle Point Array: Calculate the midpoint of a line segment. **Inputs**: auto vector: A line segment with start and end points. **Outputs**: boost::array<double, 2>: Midpoint of the line segment.
* (17-21)Middle Point Between Two Middle Point Arrays: Calculate the midpoint between two midpoints. **Inputs**: auto array\_a, auto array\_b: 2D midpoints represented as arrays.**Outputs**: boost::array<double, 2>: Midpoint between the two given midpoints
* (24-28)Distance Function for Two Line Vectors: Calculate the distance between the midpoints of two line segments. **Inputs**: auto vector\_i, auto vector\_j: Two line segments.

**Outputs**: Distance between the midpoints of the two line segments.

**Usage of auto\_dock.h in line\_extraction.cpp**

* The auto\_dock.h utilities provide additional tools for calculating distances and midpoints, which can enhance the line merging and splitting processes in the LineExtraction class.
* They can be integrated into the mergeLines and split methods to refine the criteria used for line segmentation and merging, leading to more accurate line extraction and better performance in an auto-docking system.

**line\_extraction.cpp**

1. #include "laser\_line\_extraction/line\_extraction.h"
2. #include <algorithm>
3. #include <eigen3/Eigen/Dense>
4. #include <iostream>
5. namespace line\_extraction
6. {
7. ///////////////////////////////////////////////////////////////////////////////
8. // Constructor / destructor
9. ///////////////////////////////////////////////////////////////////////////////
10. LineExtraction::LineExtraction()
11. {
12. }
13. LineExtraction::~LineExtraction()
14. {
15. }
16. ///////////////////////////////////////////////////////////////////////////////
17. // Main run function
18. ///////////////////////////////////////////////////////////////////////////////
19. void LineExtraction::extractLines(std::vector<Line>& lines)
20. {
21. // Resets
22. filtered\_indices\_ = c\_data\_.indices;
23. lines\_.clear();
24. // Filter indices
25. filterClosePoints();
26. filterOutlierPoints();
27. // Return no lines if not enough points left
28. // std::cout << params\_.min\_line\_points << std::endl;
29. if (filtered\_indices\_.size() <= std::max(params\_.min\_line\_points, static\_cast<unsigned int>(3)))
30. {
31. return;
32. }
33. // Split indices into lines and filter out short and sparse lines
34. split(filtered\_indices\_);
35. filterLines();
36. // Fit each line using least squares and merge colinear lines
37. for (std::vector<Line>::iterator it = lines\_.begin(); it != lines\_.end(); ++it)
38. {
39. it->leastSqFit();
40. }
41. // If there is more than one line, check if lines should be merged based on the merging criteria
42. if (lines\_.size() > 1)
43. {
44. mergeLines();
45. }
46. lines = lines\_;
47. // std::cout << lines.size() << std::endl;
48. }
49. ///////////////////////////////////////////////////////////////////////////////
50. // Data setting
51. ///////////////////////////////////////////////////////////////////////////////
52. void LineExtraction::setCachedData(const std::vector<double>& bearings,
    * + 1. const std::vector<double>& cos\_bearings,
        2. const std::vector<double>& sin\_bearings,
        3. const std::vector<unsigned int>& indices)
53. {
54. c\_data\_.bearings = bearings;
55. c\_data\_.cos\_bearings = cos\_bearings;
56. c\_data\_.sin\_bearings = sin\_bearings;
57. c\_data\_.indices = indices;
58. }
59. void LineExtraction::setRangeData(const std::vector<double>& ranges)
60. {
61. r\_data\_.ranges = ranges;
62. r\_data\_.xs.clear();
63. r\_data\_.ys.clear();
64. for (std::vector<unsigned int>::const\_iterator cit = c\_data\_.indices.begin();
65. cit != c\_data\_.indices.end(); ++cit)
66. {
67. r\_data\_.xs.push\_back(c\_data\_.cos\_bearings[\*cit] \* ranges[\*cit]);
68. r\_data\_.ys.push\_back(c\_data\_.sin\_bearings[\*cit] \* ranges[\*cit]);
69. }
70. }
71. ///////////////////////////////////////////////////////////////////////////////
72. // Parameter setting
73. ///////////////////////////////////////////////////////////////////////////////
74. void LineExtraction::setBearingVariance(double value)
75. {
76. params\_.bearing\_var = value;
77. }
78. void LineExtraction::setRangeVariance(double value)
79. {
80. params\_.range\_var = value;
81. }
82. void LineExtraction::setLeastSqAngleThresh(double value)
83. {
84. params\_.least\_sq\_angle\_thresh = value;
85. }
86. void LineExtraction::setLeastSqRadiusThresh(double value)
87. {
88. params\_.least\_sq\_radius\_thresh = value;
89. }
90. void LineExtraction::setMaxLineGap(double value)
91. {
92. params\_.max\_line\_gap = value;
93. }
94. void LineExtraction::setMinLineLength(double value)
95. {
96. params\_.min\_line\_length = value;
97. }
98. void LineExtraction::setMinLinePoints(unsigned int value)
99. {
100. params\_.min\_line\_points = value;
101. }
102. void LineExtraction::setMinRange(double value)
103. {
104. params\_.min\_range = value;
105. }
106. void LineExtraction::setMinSplitDist(double value)
107. {
108. params\_.min\_split\_dist = value;
109. }
110. void LineExtraction::setOutlierDist(double value)
111. {
112. params\_.outlier\_dist = value;
113. }
114. ///////////////////////////////////////////////////////////////////////////////
115. // Utility methods
116. ///////////////////////////////////////////////////////////////////////////////
117. double LineExtraction::chiSquared(const Eigen::Vector2d &dL, const Eigen::Matrix2d &P\_1,const Eigen::Matrix2d &P\_2)
118. {
119. return dL.transpose() \* (P\_1 + P\_2).inverse() \* dL;
120. }
121. double LineExtraction::distBetweenPoints(unsigned int index\_1, unsigned int index\_2)
122. {
123. return sqrt(pow(r\_data\_.xs[index\_1] - r\_data\_.xs[index\_2], 2) +

pow(r\_data\_.ys[index\_1] - r\_data\_.ys[index\_2], 2));

1. }
2. ///////////////////////////////////////////////////////////////////////////////
3. // Filtering points
4. ///////////////////////////////////////////////////////////////////////////////
5. void LineExtraction::filterClosePoints()
6. {
7. std::vector<unsigned int> output;
8. for (std::vector<unsigned int>::const\_iterator cit = filtered\_indices\_.begin();
9. cit != filtered\_indices\_.end(); ++cit)
10. {
11. if (r\_data\_.ranges[\*cit] >= params\_.min\_range)
12. {
13. output.push\_back(\*cit);
14. }
15. }
16. filtered\_indices\_ = output;
17. }
18. void LineExtraction::filterOutlierPoints()
19. {
20. if (filtered\_indices\_.size() < 3)
21. {
22. return;
23. }
24. std::vector<unsigned int> output;
25. unsigned int p\_i, p\_j, p\_k;
26. for (std::size\_t i = 0; i < filtered\_indices\_.size(); ++i)
27. {
28. // Get two closest neighbours
29. p\_i = filtered\_indices\_[i];
30. if (i == 0) // first point
31. {
32. p\_j = filtered\_indices\_[i + 1];
33. p\_k = filtered\_indices\_[i + 2];
34. }
35. else if (i == filtered\_indices\_.size() - 1) // last point
36. {
37. p\_j = filtered\_indices\_[i - 1];
38. p\_k = filtered\_indices\_[i - 2];
39. }
40. else // middle points
41. {
42. p\_j = filtered\_indices\_[i - 1];
43. p\_k = filtered\_indices\_[i + 1];
44. }
45. // Check if point is an outlier
46. if (fabs(r\_data\_.ranges[p\_i] - r\_data\_.ranges[p\_j]) > params\_.outlier\_dist &&
47. fabs(r\_data\_.ranges[p\_i] - r\_data\_.ranges[p\_k]) > params\_.outlier\_dist)
48. {
49. // Check if it is close to line connecting its neighbours
50. std::vector<unsigned int> line\_indices;
51. line\_indices.push\_back(p\_j);
52. line\_indices.push\_back(p\_k);
53. Line line(c\_data\_, r\_data\_, params\_, line\_indices);
54. line.endpointFit();
55. if (line.distToPoint(p\_i) > params\_.min\_split\_dist)
56. {
57. continue; // point is an outlier
58. }
59. }
60. output.push\_back(p\_i);
61. }
62. filtered\_indices\_ = output;
63. }
64. ///////////////////////////////////////////////////////////////////////////////
65. // Filtering and merging lines
66. ///////////////////////////////////////////////////////////////////////////////
67. void LineExtraction::filterLines()
68. {
69. std::vector<Line> output;
70. for (std::vector<Line>::const\_iterator cit = lines\_.begin(); cit != lines\_.end(); ++cit)
71. {
72. if (cit->length() >= params\_.min\_line\_length && cit->numPoints() >= params\_.min\_line\_points)
73. {
74. output.push\_back(\*cit);
75. }
76. }
77. lines\_ = output;
78. }
79. void LineExtraction::mergeLines()
80. {
81. std::vector<Line> merged\_lines;
82. for (std::size\_t i = 1; i < lines\_.size(); ++i)
83. {
84. // Get L, P\_1, P\_2 of consecutive lines
85. Eigen::Vector2d L\_1(lines\_[i-1].getRadius(), lines\_[i-1].getAngle());
86. Eigen::Vector2d L\_2(lines\_[i].getRadius(), lines\_[i].getAngle());
87. Eigen::Matrix2d P\_1;
88. P\_1 << lines\_[i-1].getCovariance()[0], lines\_[i-1].getCovariance()[1],
89. lines\_[i-1].getCovariance()[2], lines\_[i-1].getCovariance()[3];
90. Eigen::Matrix2d P\_2;
91. P\_2 << lines\_[i].getCovariance()[0], lines\_[i].getCovariance()[1],
92. lines\_[i].getCovariance()[2], lines\_[i].getCovariance()[3];
93. // Merge lines if chi-squared distance is less than 3
94. if (chiSquared(L\_1 - L\_2, P\_1, P\_2) < 3)
95. {
96. // Get merged angle, radius, and covariance
97. Eigen::Matrix2d P\_m = (P\_1.inverse() + P\_2.inverse()).inverse();
98. Eigen::Vector2d L\_m = P\_m \* (P\_1.inverse() \* L\_1 + P\_2.inverse() \* L\_2);
99. // Populate new line with these merged parameters
100. boost::array<double, 4> cov;
101. cov[0] = P\_m(0,0);
102. cov[1] = P\_m(0,1);
103. cov[2] = P\_m(1,0);
104. cov[3] = P\_m(1,1);
105. std::vector<unsigned int> indices;
106. const std::vector<unsigned int> &ind\_1 = lines\_[i-1].getIndices();
107. const std::vector<unsigned int> &ind\_2 = lines\_[i].getIndices();
108. indices.resize(ind\_1.size() + ind\_2.size());
109. indices.insert(indices.end(), ind\_1.begin(), ind\_1.end());
110. indices.insert(indices.end(), ind\_2.begin(), ind\_2.end());
111. Line merged\_line(L\_m[1], L\_m[0], cov, lines\_[i-1].getStart(), lines\_[i].getEnd(), indices);
112. // Project the new endpoints
113. merged\_line.projectEndpoints();
114. lines\_[i] = merged\_line;
115. }
116. else
117. {
118. merged\_lines.push\_back(lines\_[i-1]);
119. }
120. if (i == lines\_.size() - 1)
121. {
122. merged\_lines.push\_back(lines\_[i]);
123. }
124. }
125. lines\_ = merged\_lines;
126. }
127. ///////////////////////////////////////////////////////////////////////////////
128. // Splitting points into lines
129. ///////////////////////////////////////////////////////////////////////////////
130. void LineExtraction::split(const std::vector<unsigned int>& indices)
131. {
132. // Don't split if only a single point (only occurs when orphaned by gap)
133. if (indices.size() <= 1)
134. {
135. return;
136. }
137. Line line(c\_data\_, r\_data\_, params\_, indices);
138. line.endpointFit();
139. double dist\_max = 0;
140. double gap\_max = 0;
141. double dist, gap;
142. int i\_max, i\_gap;
143. // Find the farthest point and largest gap
144. for (std::size\_t i = 1; i < indices.size() - 1; ++i)
145. {
146. dist = line.distToPoint(indices[i]);
147. if (dist > dist\_max)
148. {
149. dist\_max = dist;
150. i\_max = i;
151. }
152. gap = distBetweenPoints(indices[i], indices[i+1]);
153. if (gap > gap\_max)
154. {
155. gap\_max = gap;
156. i\_gap = i;
157. } }
158. // Check for gaps at endpoints
159. double gap\_start = distBetweenPoints(indices[0], indices[1]);
160. if (gap\_start > gap\_max)
161. {
162. gap\_max = gap\_start;
163. i\_gap = 1;
164. }
165. double gap\_end = distBetweenPoints(indices.rbegin()[1], indices.rbegin()[0]);
166. if (gap\_end > gap\_max)
167. {
168. gap\_max = gap\_end;
169. i\_gap = indices.size() - 1;
170. }
171. // Check if line meets requirements or should be split
172. if (dist\_max < params\_.min\_split\_dist && gap\_max < params\_.max\_line\_gap)
173. {
174. lines\_.push\_back(line);
175. } else
176. {
177. int i\_split = dist\_max >= params\_.min\_split\_dist ? i\_max : i\_gap;
178. std::vector<unsigned int> first\_split(&indices[0], &indices[i\_split - 1]);
179. std::vector<unsigned int> second\_split(&indices[i\_split], &indices.back());
180. split(first\_split);
181. split(second\_split);
182. } } } // namespace line\_extraction

**EXPLANATION**

* (1-4)Includes and Namespace: Include necessary headers and start the line\_extraction namespace.
* (10-15)Constructor / Destructor: Define the constructor and destructor for the LineExtraction class.
* (19)Main Run Function:

(22-23)**Resets**: Clears previously filtered indices and lines.

(25-26)**Filters**: Filters points based on proximity (filterClosePoints) and outliers (filterOutlierPoints).

(29-31)**Early Return**: If there are not enough points left (filtered\_indices\_.size()), returns no lines.

(34**)Splitting**: Splits indices into line segments (split).

(35)**Filtering Lines**: Removes lines that are too short or sparse (filterLines).

(37-39)**Fitting**: Fits each line using a least squares method.

(42-44)**Merging**: Optionally merges colinear lines (mergeLines).

(46)**Outputs**: Sets the lines vector with detected lines.

**Purpose**: Extract lines from the provided data, filter and process them, and store the results in the provided lines vector. **Inputs**: std::vector<Line>& lines: Reference to a vector where the extracted lines will be stored. **Outputs**: Extracted lines are stored in the lines vector.

* (52-68)Data Setting Method **:** **setCachedData**: Sets bearings, cosines, sines, and indices for cached data. **setRangeData**: Sets laser scan ranges and calculates corresponding x, y coordinates.

**Purpose**: Set cached and range data required for line extraction. **Inputs**:

‘const std::vector<double>& bearings’,‘const std::vector<double>& cos\_bearings’, ‘const std::vector<double>& sin\_bearings’, ‘const std::vector<unsigned int>& indices’ :- Cached data for line extraction.

* (74-112)Parameter Setting Method: Set various parameters that control the line extraction process (setBearingVariance, setRangeVariance, etc).
* (115)Utility Methods : **Purpose**: Define utility methods for chi-squared distance calculation and distance between points. **Inputs**: 1. const Eigen::Vector2d &dL, const Eigen::Matrix2d &P\_1, const Eigen::Matrix2d &P\_2: Inputs for chi-squared calculation. 2. unsigned int index\_1, unsigned int index\_2: Indices of points for distance calculation. **Outputs**: Computed chi-squared distance and distance between points.
* (128)Filtering Points: **Purpose**: Define methods for filtering out close points and outlier points. **Inputs**: Uses internal data structures filtered\_indices\_, r\_data\_, and params\_.**Outputs**: Updates filtered\_indices\_ to exclude close points and outliers.
* (190)Filtering and Merging Lines : **Purpose**: Define method for splitting points into lines based on distance and gap criteria. **Inputs**: const std::vector<unsigned int>& indices: Indices of points to be split into lines. **Outputs**: Updates lines\_ with newly split lines if the criteria are met.

**SUMMARY**

**Input**: Sensor data (bearings, ranges), parameters controlling the extraction process.

**Output**: Extracted lines stored in the provided vector.

**Main Methods**:

* + extractLines: Main function to extract lines.
  + setCachedData, setRangeData: Set sensor data.
  + set: Set various parameters.
  + filterClosePoints, filterOutlierPoints: Filters out lines that are too short or sparse, Merges colinear lines that are close enough based on chi-squared distance.
  + split: Split points into lines based on distance criteria (dist\_max, gap\_max). If criteria are not met, recursively splits the segment.

This file implements the LineExtraction class, which processes sensor data to extract and refine lines for applications like auto-docking.

**Line.cpp**

**Line.h**

1. #ifndef LINE\_EXTRACTION\_LINE\_H
2. #define LINE\_EXTRACTION\_LINE\_H
3. #include <vector>
4. #include <boost/array.hpp>
5. #include "utilities.h"
6. namespace line\_extraction
7. {
8. class Line
9. {
10. public:
    1. // Constructor / destructor
    2. Line(const CachedData&, const RangeData&, const Params&, std::vector<unsigned int>);
    3. Line(double angle, double radius, const boost::array<double, 4> &covariance,
    4. const boost::array<double, 2> &start, const boost::array<double, 2> &end,
    5. const std::vector<unsigned int> &indices);
    6. ~Line();
    7. // Get methods for the line parameters
    8. double getAngle() const;
    9. const boost::array<double, 4>& getCovariance() const;
    10. const boost::array<double, 2>& getEnd() const;
    11. const std::vector<unsigned int>& getIndices() const;
    12. double getRadius() const;
    13. const boost::array<double, 2>& getStart() const;
    14. // Methods for line fitting
    15. double distToPoint(unsigned int);
    16. void endpointFit();
    17. void leastSqFit();
    18. double length() const;
    19. unsigned int numPoints() const;
    20. void projectEndpoints();
11. private:
    1. std::vector<unsigned int> indices\_;
    2. // Data structures
    3. CachedData c\_data\_;
    4. RangeData r\_data\_;
    5. Params params\_;
    6. PointParams p\_params\_;
    7. // Point variances used for least squares
    8. std::vector<double> point\_scalar\_vars\_;
    9. std::vector<boost::array<double, 4> > point\_covs\_;
    10. double p\_rr\_;
    11. // Line parameters
    12. double angle\_;
    13. double radius\_;
    14. boost::array<double, 2> start\_;
    15. boost::array<double, 2> end\_;
    16. boost::array<double, 4> covariance\_;
    17. // Methods
    18. void angleFromEndpoints();
    19. void angleFromLeastSq();
    20. double angleIncrement();
    21. void calcCovariance();
    22. void calcPointCovariances();
    23. void calcPointParameters();
    24. void calcPointScalarCovariances();
    25. void radiusFromEndpoints();
    26. void radiusFromLeastSq();
12. }; // class Line
13. } // namespace line\_extraction
14. #endif

**EXPLANATION**

**(6)Namespace line\_extraction**: Encloses the Line class within the namespace to prevent naming conflicts.

**(8)Class Line**: Represents a line extracted from laser scan data.

**Constructor and Destructor**:

* **Line (const CachedData&, const RangeData&, const Params&, std::vector<unsigned int>)**: Initializes the line with cached data, range data, parameters, and indices.
* **Line (double angle, double radius, const boost::array<double, 4>& covariance, const std::vector<unsigned int>& indices)**: Initializes the line with specific parameters.
* **~Line()**: Destructor, performs cleanup if necessary.

**Get Methods**:Provide access to various parameters (angle\_, covariance\_, start\_, end\_, indices\_, radius\_) of the line.

**Methods for Line Fitting**:

* **distToPoint (unsigned int)**: Calculates the perpendicular distance from the line to a point.
* **endpointFit()**: Fits the line using endpoints.
* **leastSqFit()**: Fits the line using least squares method.
* **length()**: Calculates the length of the line.
* **numPoints()**: Returns the number of points that form the line.
* **projectEndpoints()**: Projects the endpoints of the line.

**Private Members**:

* **indices\_**: Indices of points that belong to the line.
* **c\_data\_, r\_data\_, params\_, p\_params\_**: Cached data, range data, parameters, and point parameters.
* **point\_scalar\_vars\_, point\_covs\_, p\_rr\_**: Point variances, point covariances, and a scalar variance used for least squares fitting.
* **angle\_, radius\_, start\_, end\_, covariance\_**: Parameters defining the line (angle, radius, start and end points, covariance).

**Private Methods**:

* **Utility methods (**angleFromEndpoints(), angleFromLeastSq(), angleIncrement(), calcCovariance(), calcPointCovariances(), calcPointParameters(), calcPointScalarCovariances(), radiusFromEndpoints(), radiusFromLeastSq()**)**: Calculate line parameters, angles, variances, and perform calculations for least squares fitting.

**Line.cpp**

1. // Constructor / destructor
2. ///////////////////////////////////////////////////////////////////////////////
3. Line::Line(const CachedData &c\_data, const RangeData &r\_data, const Params &params,

std::vector<unsigned int> indices):

1. c\_data\_(c\_data), #include "laser\_line\_extraction/line.h"
2. namespace line\_extraction
3. {
4. ///////////////////////////////////////////////////////////////////////////////
5. r\_data\_(r\_data),
6. params\_(params),
7. indices\_(indices)
8. {
9. }
10. Line::Line(double angle, double radius, const boost::array<double, 4> &covariance,
11. const boost::array<double, 2> &start, const boost::array<double, 2> &end,
12. const std::vector<unsigned int> &indices):
13. angle\_(angle),
14. radius\_(radius),
15. covariance\_(covariance),
16. start\_(start),
17. end\_(end),
18. indices\_(indices)
19. {
20. }
21. Line::~Line()
22. {
23. }
24. ///////////////////////////////////////////////////////////////////////////////
25. // Get methods for line parameters
26. ///////////////////////////////////////////////////////////////////////////////
27. double Line::getAngle() const
28. {
29. return angle\_;
30. }
31. const boost::array<double, 4>& Line::getCovariance() const
32. {
33. return covariance\_;
34. }
35. const boost::array<double, 2>& Line::getEnd() const
36. {
37. return end\_;
38. }
39. const std::vector<unsigned int>& Line::getIndices() const
40. {
41. return indices\_;
42. }
43. double Line::getRadius() const
44. {
45. return radius\_;
46. }
47. const boost::array<double, 2>& Line::getStart() const
48. {
49. return start\_;
50. }
51. ///////////////////////////////////////////////////////////////////////////////
52. // Utility methods
53. ///////////////////////////////////////////////////////////////////////////////
54. double Line::distToPoint(unsigned int index)
55. {
56. double p\_rad = sqrt(pow(r\_data\_.xs[index], 2) + pow(r\_data\_.ys[index], 2));
57. double p\_ang = atan2(r\_data\_.ys[index], r\_data\_.xs[index]);
58. return fabs(p\_rad \* cos(p\_ang - angle\_) - radius\_);
59. }
60. double Line::length() const
61. {
62. return sqrt(pow(start\_[0] - end\_[0], 2) + pow(start\_[1] - end\_[1], 2));
63. }
64. unsigned int Line::numPoints() const
65. {
66. return indices\_.size();
67. }
68. void Line::projectEndpoints()
69. {
70. double s = -1.0 / tan(angle\_);
71. double b = radius\_ / sin(angle\_);
72. double x = start\_[0];
73. double y = start\_[1];
74. start\_[0] = (s \* y + x - s \* b) / (pow(s, 2) + 1);
75. start\_[1] = (pow(s, 2) \* y + s \* x + b) / (pow(s, 2) + 1);
76. x = end\_[0];
77. y = end\_[1];
78. end\_[0] = (s \* y + x - s \* b) / (pow(s, 2) + 1);
79. end\_[1] = (pow(s, 2) \* y + s \* x + b) / (pow(s, 2) + 1);
80. }
81. ///////////////////////////////////////////////////////////////////////////////
82. // Methods for endpoint line fitting
83. ///////////////////////////////////////////////////////////////////////////////
84. void Line::endpointFit()
85. {
86. start\_[0] = r\_data\_.xs[indices\_[0]];
87. start\_[1] = r\_data\_.ys[indices\_[0]];
88. end\_[0] = r\_data\_.xs[indices\_.back()];
89. end\_[1] = r\_data\_.ys[indices\_.back()];
90. angleFromEndpoints();
91. radiusFromEndpoints();
92. }
93. void Line::angleFromEndpoints()
94. {
95. double slope;
96. if (fabs(end\_[0] - start\_[0]) > 1e-9)
97. {
98. slope = (end\_[1] - start\_[1]) / (end\_[0] - start\_[0]);
99. angle\_ = pi\_to\_pi(atan(slope) + M\_PI/2);
100. }
101. else
102. {
103. angle\_ = 0.0;
104. }
105. }
106. void Line::radiusFromEndpoints()
107. {
108. radius\_ = start\_[0] \* cos(angle\_) + start\_[1] \* sin(angle\_);
109. if (radius\_ < 0)
110. {
111. radius\_ = -radius\_;
112. angle\_ = pi\_to\_pi(angle\_ + M\_PI);
113. }
114. }
115. ///////////////////////////////////////////////////////////////////////////////
116. // Methods for least squares line fitting
117. ///////////////////////////////////////////////////////////////////////////////
118. void Line::leastSqFit()
119. {
120. calcPointCovariances();
121. double prev\_radius = 0.0;
122. double prev\_angle = 0.0;
123. while (fabs(radius\_ - prev\_radius) > params\_.least\_sq\_radius\_thresh ||
124. fabs(angle\_ - prev\_angle) > params\_.least\_sq\_angle\_thresh)
125. {
126. prev\_radius = radius\_;
127. prev\_angle = angle\_;
128. calcPointScalarCovariances();
129. radiusFromLeastSq();
130. angleFromLeastSq();
131. }
132. calcCovariance();
133. projectEndpoints();
134. }
135. void Line::angleFromLeastSq()
136. {
137. calcPointParameters();
138. angle\_ += angleIncrement();
139. }
140. double Line::angleIncrement()
141. {
142. const std::vector<double> &a = p\_params\_.a;
143. const std::vector<double> &ap = p\_params\_.ap;
144. const std::vector<double> &app = p\_params\_.app;
145. const std::vector<double> &b = p\_params\_.b;
146. const std::vector<double> &bp = p\_params\_.bp;
147. const std::vector<double> &bpp = p\_params\_.bpp;
148. const std::vector<double> &c = p\_params\_.c;
149. const std::vector<double> &s = p\_params\_.s;
150. double numerator = 0;
151. double denominator = 0;
152. for (std::size\_t i = 0; i < a.size(); ++i)
153. {
154. numerator += (b[i] \* ap[i] - a[i] \* bp[i]) / pow(b[i], 2);
155. denominator += ((app[i] \* b[i] - a[i] \* bpp[i]) \* b[i] -
     * 1. 2 \* (ap[i] \* b[i] - a[i] \* bp[i]) \* bp[i]) / pow(b[i], 3);
156. }
157. return -(numerator/denominator);
158. }
159. void Line::calcCovariance()
160. {
161. covariance\_[0] = p\_rr\_;
162. const std::vector<double> &a = p\_params\_.a;
163. const std::vector<double> &ap = p\_params\_.ap;
164. const std::vector<double> &app = p\_params\_.app;
165. const std::vector<double> &b = p\_params\_.b;
166. const std::vector<double> &bp = p\_params\_.bp;
167. const std::vector<double> &bpp = p\_params\_.bpp;
168. const std::vector<double> &c = p\_params\_.c;
169. const std::vector<double> &s = p\_params\_.s;
170. double G = 0;
171. double A = 0;
172. double B = 0;
173. double r, phi;
174. for (std::size\_t i = 0; i < a.size(); ++i)
175. {
176. r = r\_data\_.ranges[indices\_[i]]; // range
177. phi = c\_data\_.bearings[indices\_[i]]; // bearing
178. G += ((app[i] \* b[i] - a[i] \* bpp[i]) \* b[i] - 2 \* (ap[i] \* b[i] - a[i] \* bp[i]) \* bp[i]) / pow(b[i], 3);
179. A += 2 \* r \* sin(angle\_ - phi) / b[i];
180. B += 4 \* pow(r, 2) \* pow(sin(angle\_ - phi), 2) / b[i];
181. }
182. covariance\_[1] = p\_rr\_ \* A / G;
183. covariance\_[2] = covariance\_[1];
184. covariance\_[3] = pow(1.0 / G, 2) \* B;
185. }
186. void Line::calcPointCovariances()
187. {
188. point\_covs\_.clear();
189. double r, phi, var\_r, var\_phi;
190. for (std::vector<unsigned int>::const\_iterator cit = indices\_.begin(); cit != indices\_.end(); ++cit)
191. {
192. r = r\_data\_.ranges[\*cit]; // range
193. phi = c\_data\_.bearings[\*cit]; // bearing
194. var\_r = params\_.range\_var; // range variance
195. var\_phi = params\_.bearing\_var; // bearing variance
196. boost::array<double, 4> Q;
197. Q[0] = pow(r, 2) \* var\_phi \* pow(sin(phi), 2) + var\_r \* pow(cos(phi), 2);
198. Q[1] = -pow(r, 2) \* var\_phi \* sin(2 \* phi) / 2.0 + var\_r \* sin(2 \* phi) / 2.0;
199. Q[2] = Q[1];
200. Q[3] = pow(r, 2) \* var\_phi \* pow(cos(phi), 2) + var\_r \* pow(sin(phi), 2);
201. point\_covs\_.push\_back(Q);
202. }
203. }
204. void Line::calcPointParameters()
205. {
206. p\_params\_.a.clear();
207. p\_params\_.ap.clear();
208. p\_params\_.app.clear();
209. p\_params\_.b.clear();
210. p\_params\_.bp.clear();
211. p\_params\_.bpp.clear();
212. p\_params\_.c.clear();
213. p\_params\_.s.clear();
214. double r, phi, var\_r, var\_phi;
215. double a, ap, app, b, bp, bpp, c, s;
216. for (std::vector<unsigned int>::const\_iterator cit = indices\_.begin(); cit != indices\_.end(); ++cit)
217. {
218. r = r\_data\_.ranges[\*cit]; // range
219. phi = c\_data\_.bearings[\*cit]; // bearing
220. var\_r = params\_.range\_var; // range variance
221. var\_phi = params\_.bearing\_var; // bearing variance
222. c = cos(angle\_ - phi);
223. s = sin(angle\_ - phi);
224. a = pow(r \* c - radius\_, 2);
225. ap = -2 \* r \* s \* (r \* c - radius\_);
226. app = 2 \* pow(r, 2) \* pow(s, 2) - 2 \* r \* c \* (r \* c - radius\_);
227. b = var\_r \* pow(c, 2) + var\_phi \* pow(r, 2) \* pow(s, 2);
228. bp = 2 \* (pow(r, 2) \* var\_phi - var\_r) \* c \* s;
229. bpp = 2 \* (pow(r, 2) \* var\_phi - var\_r) \* (pow(c, 2) - pow(s, 2));
230. p\_params\_.a.push\_back(a);
231. p\_params\_.ap.push\_back(ap);
232. p\_params\_.app.push\_back(app);
233. p\_params\_.b.push\_back(b);
234. p\_params\_.bp.push\_back(bp);
235. p\_params\_.bpp.push\_back(bpp);
236. p\_params\_.c.push\_back(c);
237. p\_params\_.s.push\_back(s);
238. }
239. }
240. void Line::calcPointScalarCovariances()
241. {
242. point\_scalar\_vars\_.clear();
243. double P;
244. double inverse\_P\_sum = 0;
245. for (std::vector<boost::array<double, 4> >::const\_iterator cit = point\_covs\_.begin();
246. cit != point\_covs\_.end(); ++cit)
247. {
248. P = (\*cit)[0] \* pow(cos(angle\_), 2) + 2 \* (\*cit)[1] \* sin(angle\_) \* cos(angle\_) +
249. (\*cit)[3] \* pow(sin(angle\_), 2);
250. inverse\_P\_sum += 1.0 / P;
251. point\_scalar\_vars\_.push\_back(P);
252. }
253. p\_rr\_ = 1.0 / inverse\_P\_sum;
254. }
255. void Line::radiusFromLeastSq()
256. {
257. radius\_ = 0;
258. double r, phi;
259. for (std::vector<unsigned int>::const\_iterator cit = indices\_.begin(); cit != indices\_.end(); ++cit)
260. {
261. r = r\_data\_.ranges[\*cit]; // range
262. phi = c\_data\_.bearings[\*cit]; // bearing
263. radius\_ += r \* cos(angle\_ - phi) / point\_scalar\_vars\_[cit - indices\_.begin()]; // cit to index
264. }
265. radius\_ \*= p\_rr\_;
266. }
267. } // namespace line\_extraction

**EXPLANATION**

* **Includes**:
  + Includes the line.h header file which defines the Line class.
* **Namespace line\_extraction**:
  + Implements the Line class within the same namespace as declared in line.h.
* **Constructor Definitions**:
  + Implementations for both constructors (Line::Line(const CachedData&, const RangeData&, const Params&, std::vector<unsigned int>) and Line::Line(double angle, double radius, const boost::array<double, 4>& covariance, const boost::array<double, 2>& start, const boost::array<double, 2>& end, const std::vector<unsigned int>& indices)).
* **Destructor Definition**:
  + Empty implementation for the destructor (Line::~Line()).
* **Getter Method Implementations**:
  + Implementations for the getter methods defined in line.h, retrieving various line parameters.
* **Utility Methods Implementation**:
  + **distToPoint(unsigned int index)**: Calculates the distance from the line to a specified point index using laser scan data.
  + **length()**: Calculates the Euclidean distance between start\_ and end\_.
  + **numPoints()**: Returns the number of points (indices\_) that define the line.
  + **projectEndpoints()**: Projects the endpoints (start\_ and end\_) of the line based on angle and radius.
* **Endpoint Line Fitting Methods Implementation**:
  + **endpointFit()**: Fits a line using endpoints defined by indices in indices\_.
  + **angleFromEndpoints()**: Calculates the angle of the line using the endpoints.
  + **radiusFromEndpoints()**: Calculates the radius of the line using the endpoints.
* **Least Squares Line Fitting Methods Implementation**:
  + **leastSqFit()**: Refines the line parameters using a least squares approach, iterating until convergence based on thresholds.
  + **angleFromLeastSq()**: Adjusts the angle of the line during least squares fitting.
  + **angleIncrement()**: Computes the angle adjustment increment during least squares fitting.
  + **calcCovariance()**: Calculates the covariance matrix of the line parameters.
  + **calcPointCovariances()**: Calculates the covariance matrices for each point (from indices\_).
  + **calcPointParameters()**: Calculates parameters used in least squares fitting for each point.
  + **calcPointScalarCovariances()**: Calculates scalar variances for each point used in least squares fitting.
  + **radiusFromLeastSq()**: Adjusts the radius of the line during least squares fitting.

### **Input and Output:**

**Input**: The class takes input data (CachedData, RangeData, Params, and std::vector<unsigned int> indices) through its constructors and various methods. This data includes cached sensor data (c\_data\_ and r\_data\_), fitting parameters (params\_), and indices of points defining the line.

**Output**: The class provides methods (getAngle(), getRadius(), getStart(), getEnd(), etc.) to retrieve line parameters (angle\_, radius\_, start\_, end\_, covariance\_) after fitting procedures (endpointFit(), leastSqFit()) have been performed. These parameters represent the geometric characteristics and uncertainties of the detected line segment based on the input laser scan data.

**Line\_extraction\_ros.cpp**

**Line\_extraction\_ros.h**

1. #ifndef LINE\_EXTRACTION\_ROS\_HPP
2. #define LINE\_EXTRACTION\_ROS\_HPP
3. #include <vector>
4. #include <string>
5. #include "rclcpp/rclcpp.hpp"
6. #include "sensor\_msgs/msg/laser\_scan.hpp"
7. #include "visualization\_msgs/msg/marker.hpp"
8. #include "geometry\_msgs/msg/point.hpp"
9. #include "laser\_line\_extraction/LineSegment.hpp"
10. #include "laser\_line\_extraction/LineSegmentList.hpp"
11. #include "laser\_line\_extraction/line\_extraction.hpp"
12. #include "laser\_line\_extraction/line.hpp"
13. namespace line\_extraction
14. {
15. class LineExtractionROS : public rclcpp::Node
16. {
17. public:
18. // Constructor / destructor
19. LineExtractionROS(const std::string& node\_name, const std::string& node\_namespace);
20. ~LineExtractionROS();
21. // Running
22. void run();
23. private:
24. // ROS
25. rclcpp::Subscription<sensor\_msgs::msg::LaserScan>::SharedPtr scan\_subscription\_;
26. rclcpp::Publisher<laser\_line\_extraction::msg::LineSegmentList>::SharedPtr line\_publisher\_;
27. rclcpp::Publisher<visualization\_msgs::msg::Marker>::SharedPtr marker\_publisher\_;
28. // Parameters
29. std::string frame\_id\_;
30. std::string scan\_topic\_;
31. bool pub\_markers\_;
32. // Line extraction
33. LineExtraction line\_extraction\_;
34. bool data\_cached\_; // true after first scan used to cache data
35. // Members
36. void loadParameters();
37. void populateLineSegListMsg(const std::vector<Line>& lines, laser\_line\_extraction::msg::LineSegmentList& msg);
38. void populateMarkerMsg(const std::vector<Line>& lines, visualization\_msgs::msg::Marker& marker\_msg);
39. void cacheData(const sensor\_msgs::msg::LaserScan::SharedPtr scan\_msg);
40. void laserScanCallback(const sensor\_msgs::msg::LaserScan::SharedPtr scan\_msg);
41. };
42. } // namespace line\_extraction
43. #endif // LINE\_EXTRACTION\_ROS\_HPP

**EXPLANATION**

* **Header Guards (#ifndef, #define, #endif)**:

These guards prevent the header file from being included more than once in the same compilation unit, ensuring that the declarations are processed only once.

* **Namespace line\_extraction**:

Encloses the LineExtractionROS class within a namespace to avoid naming conflicts.

* **Class LineExtractionROS**:

Inherits from rclcpp::Node, making it a ROS 2 node capable of handling communication and lifecycle events.

* **Public**:

**Constructor and Destructor**: Initializes and cleans up the node.

**run()**: Main function to execute the line extraction and publishing process.

* **Private Members**:

**ROS Communication**:

* + scan\_subscription\_: Subscription to laser scan messages.
  + line\_publisher\_: Publisher for LineSegmentList messages.
  + marker\_publisher\_: Publisher for visualization markers.

**Parameters**:

* + frame\_id\_: Frame ID used for message headers.
  + scan\_topic\_: Topic name for subscribing to laser scan messages.
  + pub\_markers\_: Flag indicating whether to publish visualization markers.

**Line Extraction**:

* + line\_extraction\_: Instance of LineExtraction class responsible for line extraction algorithms.
  + data\_cached\_: Flag indicating if laser scan data has been cached.

**Private Methods**:

**loadParameters()**: Loads parameters from ROS parameters server.

**populateLineSegListMsg()**: Converts Line objects to LineSegmentList messages.

**populateMarkerMsg()**: Converts Line objects to visualization marker messages.

**cacheData()**: Caches laser scan data for processing.

**laserScanCallback()**: Callback function triggered upon receiving laser scan messages.

**Line\_extraction\_ros.cpp**

1. #include "laser\_line\_extraction/line\_extraction\_node.h"
2. void line\_extraction::LineExtractionROS::run()
3. {
4. // Extract the lines
5. std::vector<Line> lines;
6. line\_extraction\_.setBearingVariance(bearing\_std\_dev);
7. line\_extraction\_.setRangeVariance(range\_std\_dev);
8. line\_extraction\_.setLeastSqAngleThresh(least\_sq\_angle\_thresh);
9. line\_extraction\_.setLeastSqRadiusThresh(least\_sq\_radius\_thresh);
10. line\_extraction\_.setMaxLineGap(max\_line\_gap);
11. line\_extraction\_.setMinLineLength(min\_line\_length);
12. line\_extraction\_.setMinLinePoints(min\_line\_points);
13. line\_extraction\_.setMinRange(min\_range);
14. line\_extraction\_.setMinSplitDist(min\_split\_dist);
15. line\_extraction\_.setOutlierDist(outlier\_dist);
16. line\_extraction\_.extractLines(lines);
17. // Populate message
18. laser\_line\_msgs::msg::LineSegmentList msg;
19. populateLineSegListMsg(lines, msg);
20. // publish the line
21. line\_publisher\_->publish(msg);
22. if (pub\_markers\_)
23. {
24. visualization\_msgs::msg::Marker marker\_msg;
25. populateMarkerMsg(lines, marker\_msg);
26. marker\_publisher\_->publish(marker\_msg);
27. }
28. }
29. ///////////////////////////////////////////////////////////////////////////////
30. // Cache data on first LaserScan message received
31. ///////////////////////////////////////////////////////////////////////////////
32. void line\_extraction::LineExtractionROS::CachedData(const sensor\_msgs::msg::LaserScan::SharedPtr scan)
33. {
34. std::vector<double> bearings, cos\_bearings, sin\_bearings;
35. std::vector<unsigned int> indices;
36. // calculated number of laser scan of each ray
37. const std::size\_t num\_measurements = std::ceil((scan->angle\_max - scan->angle\_min) / scan->angle\_increment);
38. for (std::size\_t i = 0; i < num\_measurements; ++i)
39. {
40. const double b = scan->angle\_min + i \* scan->angle\_increment; // calculated each laser point angle
41. bearings.push\_back(b); // store each scan angle in bearings
42. cos\_bearings.push\_back(cos(b)); // calculated each cos(angle)
43. sin\_bearings.push\_back(sin(b)); // calculated each sin(angle)
44. indices.push\_back(i);
45. }
46. line\_extraction\_.setCachedData(bearings, cos\_bearings, sin\_bearings, indices);
47. std::cout << "Data has been cached." << std::endl;
48. }
49. void line\_extraction::LineExtractionROS::scan\_cb(const sensor\_msgs::msg::LaserScan::SharedPtr scan)
50. {
51. // std::cout << "data\_cached\_\tis\t" << data\_cached\_ << std::endl;
52. if (!data\_cached\_)
53. {
54. CachedData(scan);
55. data\_cached\_ = true;
56. }
57. // std::cout << "data\_cached\_\tis\t" << data\_cached\_ << std::endl;
58. std::vector<double> scan\_ranges\_doubles(scan->ranges.begin(), scan->ranges.end());
59. line\_extraction\_.setRangeData(scan\_ranges\_doubles);
60. }
61. void line\_extraction::LineExtractionROS::populateLineSegListMsg(const std::vector<Line>& lines, laser\_line\_msgs::msg::LineSegmentList& line\_list\_msg)
62. {
63. // std::cout << "populateLineSegListMsg start" << std::endl;
64. // std::cout << "number of line is\t" << lines.size() << std::endl;
65. // start vector (x, y)
66. std::vector<float> line\_start(2);
67. //end vector (x,y)
68. std::vector<float> line\_end(2);
69. // line angel for each line
70. std::vector<float> line\_angle(lines.size());
71. double diff\_x, diff\_y, angelofline;
72. for (std::vector<Line>::const\_iterator cit = lines.begin(); cit != lines.end(); ++cit)
73. {
74. laser\_line\_msgs::msg::LineSegment line\_msg;
75. line\_msg.angle = cit->getAngle();
76. line\_msg.radius = cit->getRadius();
77. // get covariance
78. line\_msg.covariance[0] = cit->getCovariance()[0];
79. line\_msg.covariance[1] = cit->getCovariance()[1];
80. line\_msg.covariance[2] = cit->getCovariance()[2];
81. line\_msg.covariance[3] = cit->getCovariance()[3];
82. // get line start point (x, y)
83. line\_msg.start[0] = cit->getStart()[0];
84. line\_msg.start[1] = cit->getStart()[1];
85. // get line end point (x, y)
86. line\_msg.end[0] = cit->getEnd()[0];
87. line\_msg.end[1] = cit->getEnd()[1];
88. line\_list\_msg.line\_segments.push\_back(line\_msg);
89. }
90. line\_list\_msg.header.frame\_id = frame\_id\_;
91. line\_list\_msg.header.stamp = rclcpp::Node::now();
92. }
93. void line\_extraction::LineExtractionROS::populateMarkerMsg(const std::vector<Line>& lines, visualization\_msgs::msg::Marker& marker\_msg)
94. {
95. marker\_msg.ns = "line\_extraction";
96. marker\_msg.id = 0;
97. marker\_msg.type = visualization\_msgs::msg::Marker::LINE\_LIST;
98. marker\_msg.scale.x = 0.1;
99. marker\_msg.color.r = 0.0;
100. marker\_msg.color.g = 1.0;
101. marker\_msg.color.b = 0.0;
102. marker\_msg.color.a = 1.0;
103. for (std::vector<Line>::const\_iterator cit = lines.begin(); cit != lines.end(); ++cit)
104. {
105. geometry\_msgs::msg::Point p\_start;
106. p\_start.x = cit->getStart()[0];
107. p\_start.y = cit->getStart()[1];
108. p\_start.z = 0;
109. marker\_msg.points.push\_back(p\_start);
110. geometry\_msgs::msg::Point p\_end;
111. p\_end.x = cit->getEnd()[0];
112. p\_end.y = cit->getEnd()[1];
113. p\_end.z = 0;
114. marker\_msg.points.push\_back(p\_end);
115. }
116. marker\_msg.header.frame\_id = frame\_id\_;
117. marker\_msg.header.stamp = rclcpp::Node::now();
118. }

**EXPLANATION**

* **Implementation of run()**:
  + Executes the main processing loop:
    - Configures line\_extraction\_ parameters (bearing\_std\_dev, range\_std\_dev, etc.).
    - Calls line\_extraction\_.extractLines() to extract lines from laser scan data.
    - Populates ROS messages (msg and marker\_msg) using populateLineSegListMsg() and publishes them (line\_publisher\_

->publish() and marker\_publisher\_->publish()).

* **cacheData()**:
  + Caches laser scan data (bearings, cos\_bearings, sin\_bearings, indices) into line\_extraction\_ using setCachedData().
* **laserScanCallback()**:
  + Entry point for incoming laser scan messages.
  + Caches data if not already cached (!data\_cached\_), then sets range data in line\_extraction\_.
* **populateLineSegListMsg()**:
  + Converts Line objects to LineSegmentList messages (line\_list\_msg).
* **populateMarkerMsg()**:
  + Converts Line objects to visualization marker messages (marker\_msg).

### **FLOW:**

* **Initialization**: Upon instantiation, parameters are loaded and publishers/subscribers are set up.
* **Callback Handling**: laserScanCallback() caches data and triggers line extraction when new scan data arrives.
* **Line Extraction**: run() manages the line extraction process and publishes results as ROS messages.
* **Message Population**: populateLineSegListMsg() and populateMarkerMsg() convert extracted lines into appropriate message formats for publishing.

### **INPUT AND OUTPUT:**

* **Input**: Laser scan data (sensor\_msgs::msg::LaserScan) received via callbacks (laserScanCallback()).
* **Output**: Published messages (laser\_line\_extraction::msg::LineSegmentList and visualization\_msgs::msg::Marker) containing detected line segments and visualization markers.

**LineSegementList.msg**

The LineSegmentList.msg file defines a ROS message type for publishing a list of line segments detected by the line extraction process. This message includes a header for timestamping and frame information, and an array of LineSegment messages that describe the individual line segments.

Here’s the content of LineSegmentList.msg:

1. std\_msgs/Header header
2. laser\_line\_msgs/LineSegment[] line\_segments

**LineSegment.msg**

**Code**

1. float32 angle
2. float32 radius
3. float32 length
4. geometry\_msgs/Point start
5. geometry\_msgs/Point end
6. float32[4] covariance

* f**loat32 angle**: Angle of the line segment relative to some reference (e.g., x-axis).
* **float32 radius**: Distance of the line segment from the origin.
* **Float32 length**: Length of the line segment.
* **geometry\_msgs/Point start**: Starting point of the line segment.
* **geometry\_msgs/Point end**: Ending point of the line segment.
* **Float32[4] covariance**: Covariance matrix of the line segment parameters.

The LineSegmentList message would come into play towards the end of the line extraction process when the extracted lines are published for further use, such as by the docking controller or other higher-level components in the system. A LineSegmentList message is populated with the header and the array of LineSegment messages. This message is then published on a ROS topic for consumption by other nodes.

**Pattern.cpp**

**Pattern.h**

#include <string>

#include <rclcpp/rclcpp.hpp>

#include <sensor\_msgs/msg/laser\_scan.hpp>

#include <geometry\_msgs/msg/point.hpp>

#include <tf2\_ros/transform\_broadcaster.h>

#include <geometry\_msgs/msg/twist.hpp>

#include <tf2\_geometry\_msgs/tf2\_geometry\_msgs.h>

#include <chrono>

#include <tf2/LinearMath/Quaternion.h>

#include <auto\_dock.h>

#include "laser\_line\_msgs/msg/line\_segment\_list.hpp"

#include "laser\_line\_msgs/msg/line\_segment.hpp"

#include "laser\_line\_extraction/line\_extraction.h"

#include <vector>

namespace automatic\_parking {

class autodock\_pattern : public rclcpp::Node{

public:

autodock\_pattern(): Node("autodock\_pattern"){

this->declare\_parameter<double>("pattern\_angle1",3.81);

this->get\_parameter("pattern\_angle1", pattern\_angle1);

this->declare\_parameter<double>("pattern\_angle2",1.8);

this->get\_parameter("pattern\_angle2" ,pattern\_angle2);

this->declare\_parameter<double>("pattern\_angle3",3.81);

this->get\_parameter("pattern\_angle3" , pattern\_angle3);

this->declare\_parameter<double>("detect\_angle\_tolerance",0.23);

this->get\_parameter("detect\_angle\_tolerance" ,detect\_angle\_tolerance);

this->declare\_parameter<double>("group\_dist\_tolerance",0.15);

this->get\_parameter("group\_dist\_tolerance" , group\_dist\_tolerance);

this->declare\_parameter<std::string>("laser\_frame\_id","laser\_frame");

this->get\_parameter("laser\_frame\_id", laser\_frame\_id);

//this->declare\_parameter<std::string>("base\_frame\_id","odom");

this->declare\_parameter<std::string>("base\_frame\_id","base\_link");

this->get\_parameter("base\_frame\_id" , base\_frame\_id);

line\_sub\_=this>create\_subscription<laser\_line\_msgs::msg::LineSegmentList>("line\_segments",rclcpp::QoS(rclcpp::KeepLast(10)).best\_effort().durability\_volatile(),

std::bind(&autodock\_pattern::patternCallback,this, std::placeholders::\_1));

tf\_broadcaster\_ = std::make\_unique<tf2\_ros::TransformBroadcaster>(this);

}

private:

struct Point\_set{

boost::array<double, 2> vector\_a;

boost::array<double, 2> vector\_b;

boost::array<double, 2> vector\_c;

boost::array<double, 2> vector\_d;

} ;

void patternCallback(const laser\_line\_msgs::msg::LineSegmentList::SharedPtr msg );

void temp\_vector(int& , int& ,int& , std::vector<laser\_line\_msgs::msg::LineSegment\_<std::allocator<void>>>&);

bool check\_center(const std::vector<int>& dock\_vector,const std::vector<laser\_line\_msgs::msg::LineSegment\_<std::allocator<void>>>&);

void updateVectors();

void populateTF(double x, double y, double theta );

bool calAngle(double a, double b, double angle\_ab);

rclcpp::Subscription<laser\_line\_msgs::msg::LineSegmentList>::SharedPtr line\_sub\_;

std::unique\_ptr<tf2\_ros::TransformBroadcaster> tf\_broadcaster\_;

std::vector<int> dock\_vector = {0 ,0 ,0 ,0};

//std::vector<int> dock\_vector = {0, 1, 2, 3};

Point\_set point\_set,point\_temp;

boost::array<double, 2> temp\_point\_1 , temp\_point\_2;

bool check\_angle = false;

std::string frame\_name = "dock\_frame";

double pattern\_angle1 , pattern\_angle2 ,pattern\_angle3 ,detect\_angle\_tolerance ,group\_dist\_tolerance;

std::string laser\_frame\_id ,base\_frame\_id;

boost::array<double, 2> center\_point;

};

**EXPLANATION**

* **Header Guards (#ifndef, #define, #endif)**:

These prevent multiple inclusions of the header file in the same translation unit, ensuring definitions are processed only once.

* **Namespace automatic\_parking**:

Encloses the autodock\_pattern class within a namespace to avoid naming conflicts.

* **Class autodock\_pattern**:

Inherits from rclcpp::Node, making it a ROS 2 node capable of communication and lifecycle management.

* **Public and Private Members**:

**Public Constructor**: Initializes parameters and sets up the ROS subscription for line segment messages. **Private Struct Point\_set**: Holds arrays for vector points (vector\_a, vector\_b, vector\_c, vector\_d).

**Private**:

* + patternCallback(): Callback function for handling incoming line segment messages.
  + temp\_vector(): Helper function to manage temporary vectors during processing.
  + check\_center(): Checks if the center condition between specified vectors is met.
  + updateVectors(): Updates the point\_set with point\_temp.
  + populateTF(): Publishes a TF message for the dock frame.
  + calAngle(): Calculates the angle between two lines.

**Private**:

* + line\_sub\_: ROS subscription to laser\_line\_msgs::msg::LineSegmentList messages.
  + tf\_broadcaster\_: Manages broadcasting of TF messages.
  + Various variables (dock\_vector, point\_set, point\_temp, etc.) for storing and managing data during processing.
  + Parameters (pattern\_angle1, pattern\_angle2, etc.) and frame IDs (laser\_frame\_id, base\_frame\_id) configured via ROS parameters.

**Pattern.cpp**

1. #include "pattern.h"
2. #include <chrono>
3. #include <iomanip>
4. #include <iostream>
5. #include <vector>
6. using namespace automatic\_parking;
7. bool autodock\_pattern::calAngle(double a, double b, double angle\_ab){
8. double angle;
9. angle = fabs(a-b);
10. if (angle < M\_PI and angle > 1.7 ){
11. angle = angle - M\_PI\_2 ;
12. }
13. else if (angle> (M\_PI\_2 + M\_PI)){
14. angle = angle - M\_PI - M\_PI\_2;
15. }
16. else if (angle> M\_PI and angle < (M\_PI\_2 + M\_PI)){
17. angle = angle - M\_PI ;
18. }
19. RCLCPP\_INFO(get\_logger(),"angle:%f , ab:%f \n",angle,angle\_ab);
20. if (fabs(angle\_ab-angle)<=detect\_angle\_tolerance){
21. return true;}
22. else return false;
23. }
24. void autodock\_pattern::populateTF(double x, double y, double theta){
25. // publish dock\_frame
26. auto now = get\_clock()->now();
27. tf2::Quaternion q;
28. q.setRPY(0.0, 0.0, theta);
29. geometry\_msgs::msg::TransformStamped tf\_dock;
30. tf\_dock.header.stamp = now;
31. tf\_dock.header.frame\_id = base\_frame\_id;
32. tf\_dock.child\_frame\_id = frame\_name ;
33. tf\_dock.transform.translation.x = x;
34. tf\_dock.transform.translation.y = y;
35. tf\_dock.transform.translation.z = 0;
36. tf\_dock.transform.rotation.x = q.x();
37. tf\_dock.transform.rotation.y = q.y();
38. tf\_dock.transform.rotation.z = q.z();
39. tf\_dock.transform.rotation.w = q.w();
40. tf\_broadcaster\_->sendTransform(tf\_dock);
41. RCLCPP\_INFO(get\_logger(), "Published dock\_frame: frame\_id=%s, child\_frame\_id=%s, x=%f, y=%f, theta=%f", tf\_dock.header.frame\_id.c\_str(), tf\_dock.child\_frame\_id.c\_str(), x, y, theta);
42. }
43. void autodock\_pattern::updateVectors(){
44. point\_set = point\_temp;
45. //printf("Upate vectors!\n");
46. RCLCPP\_INFO(get\_logger(),"Upate vectors!");
47. }
48. bool autodock\_pattern::check\_center(const std::vector<int> &dock\_vector, const std::vector<laser\_line\_msgs::msg::LineSegment\_<std::allocator<void>>> &vectors){
49. for(int i=0; i<3; i++){
50. for(int j(i+1); j<=3 ; j++){
    1. if (calAngle(vectors[dock\_vector[i]].angle,vectors[dock\_vector[j]].angle, 3.14-pattern\_angle2)){
    2. return true;
    3. }
51. }
52. }
53. return false;
54. }
55. void autodock\_pattern::temp\_vector(int &i , int &j ,int &angle\_count, std::vector<laser\_line\_msgs::msg::LineSegment\_<std::allocator<void>>> &vectors ){
56. if (dock\_vector.size() < 4){
57. dock\_vector.resize(4);}
58. if (angle\_count == 1){
59. point\_temp.vector\_a = mid\_point(vectors[i]);
60. point\_temp.vector\_b = mid\_point(vectors[j]);
61. dock\_vector[0] = i;
62. dock\_vector[1] = j;
63. }
64. else{
65. dock\_vector[2] = i;
66. dock\_vector[3] = j;
67. point\_temp.vector\_d = mid\_point(vectors[i]);
68. point\_temp.vector\_c = mid\_point(vectors[j]);
69. temp\_point\_1 = mid\_two\_point(point\_temp.vector\_a , point\_temp.vector\_b);
70. temp\_point\_2 = mid\_two\_point(point\_temp.vector\_c , point\_temp.vector\_d);
71. if (dist(temp\_point\_1,temp\_point\_2) <= 0.3 and check\_center( dock\_vector, vectors)){
    1. //printf("%s\n", check\_center(dock\_vector , vectors ) ? "true" : "false");
    2. updateVectors();
    3. check\_angle = true;
72. }
73. }
74. }
75. void autodock\_pattern::patternCallback(const laser\_line\_msgs::msg::LineSegmentList::SharedPtr msg){
76. std::vector<laser\_line\_msgs::msg::LineSegment\_<std::allocator<void>>> vectors = msg->line\_segments;
77. // Number of the line
78. int lineNum = vectors.size();
79. int angle\_count = 0;
80. bool check\_vec\_size = true;
81. check\_angle = false;
82. RCLCPP\_INFO(get\_logger(),"lineNum = %d\n",lineNum);
83. // Check whether topic line\_segments is publishing
84. if (lineNum < 4){
85. RCLCPP\_ERROR(get\_logger(),"There isn't enough line in the laser field!");
86. check\_vec\_size = false;
87. }
88. if (check\_vec\_size){
89. for(int i=0; i<lineNum; i++){
    1. for(int j(i+1); j<=lineNum ; j++){
    2. if (mid\_dist(vectors[i] , vectors[j]) <= group\_dist\_tolerance) {
       1. if (calAngle(vectors[i].angle,vectors[j].angle, pattern\_angle1-3.14)){
       2. angle\_count+=1;
       3. temp\_vector(i , j ,angle\_count , vectors );
       4. // RCLCPP\_INFO(get\_logger(),"angle\_count = %d\n",angle\_count);}
    3. }
    4. }
90. }
91. }
92. if (check\_angle){
93. //Set origin of frame
94. boost::array<double, 2> theta\_point\_1 = mid\_two\_point(point\_set.vector\_a , point\_set.vector\_b);
95. boost::array<double, 2> theta\_point\_2 = mid\_two\_point(point\_set.vector\_c , point\_set.vector\_d);
96. boost::array<double, 2> goal\_point = mid\_two\_point(theta\_point\_1 , theta\_point\_2);
97. double theta = atan2((theta\_point\_1[1]-theta\_point\_2[1]),(theta\_point\_1[0]-theta\_point\_2[0]));
98. RCLCPP\_INFO(get\_logger(),"x:%f , y:%f , theta:%f",goal\_point[0],goal\_point[1],theta);
99. // populate dock\_frame
100. populateTF(goal\_point[0],goal\_point[1],theta );
101. }
102. }
103. int main(int argc, char\*\* argv){
104. rclcpp::init(argc, argv);
105. auto autodock\_pattern\_node = std::make\_shared<automatic\_parking::autodock\_pattern>();
106. rclcpp::WallRate rate(20.0);
107. while (rclcpp::ok()){
108. rclcpp::spin\_some(autodock\_pattern\_node);
109. rate.sleep();
110. }
111. return 0;
112. }

**EXPLANATION**

* **Includes**: Include necessary headers for ROS 2 messages (rclcpp), TF broadcasting (tf2\_ros), message types (sensor\_msgs, geometry\_msgs, laser\_line\_msgs), and others (chrono).
* **Namespace**: Implements within the automatic\_parking namespace to encapsulate class and avoid naming conflicts.
* **Constructor (autodock\_pattern::autodock\_pattern())**:
  + Initializes ROS parameters (pattern\_angle1, pattern\_angle2, etc.) from the parameter server.
  + Creates a subscription to line\_segments topic, setting up patternCallback() as the callback function.
* **Callback (autodock\_pattern::patternCallback())**:
  + Handles incoming line\_segments messages (laser\_line\_msgs::msg::LineSegmentList::SharedPtr).
  + Processes line segments to find pairs (i, j) that meet distance and angle criteria (mid\_dist() and calAngle()).
  + Calls temp\_vector() to handle temporary vector updates and checks (check\_center()).
  + If conditions (check\_angle) are met, calculates a goal point and angle, then broadcasts a TF message (populateTF()).
* **Other Functions**:
  + temp\_vector(): Manages temporary vectors and updates when conditions are met.
  + check\_center(): Checks if the center condition among specified vectors is satisfied.
  + updateVectors(): Updates point\_set with point\_temp.
  + populateTF(): Constructs and publishes TF messages for broadcasting.
  + calAngle(): Calculates the angle between two given angles, adjusting if necessary based on conditions.

### **INPUT AND OUTPUT:**

* **Input**:
  + Subscribes to line\_segments topic (ROS message type laser\_line\_msgs::msg::LineSegmentList) containing line segments detected from laser scans.
  + Receives parameters (pattern\_angle1, pattern\_angle2, etc.) from the ROS parameter server.
* **Output**:
  + Publishes TF messages (geometry\_msgs::msg::TransformStamped) for the dock\_frame using tf\_broadcaster\_->sendTransform(tf\_dock).
  + Logs messages (RCLCPP\_INFO, RCLCPP\_ERROR) to ROS console for debugging and information purposes.

**Controller.cpp**

**Controller.h**

1. #include <tf2\_ros/transform\_listener.h>
2. #include <tf2\_ros/buffer.h>
3. #include <geometry\_msgs/msg/twist.hpp>
4. #include <tf2\_geometry\_msgs/tf2\_geometry\_msgs.h>
5. #include <chrono>
6. #include <tf2/exceptions.h>
7. #include <tf2/utils.h>
8. #include <tf2/LinearMath/Quaternion.h>
9. #include <visualization\_msgs/msg/marker.hpp>
10. #include <auto\_dock.h>
11. #include "laser\_line\_extraction/line\_extraction.h"
12. #include <string>
13. #include <rclcpp/rclcpp.hpp>
14. #include <sensor\_msgs/msg/laser\_scan.hpp>
15. #include <geometry\_msgs/msg/point.hpp>
16. #include <vector>
17. #define \_USE\_MATH\_DEFINES
18. namespace automatic\_parking {
19. class autodock\_controller : public rclcpp::Node{
20. public:
    1. autodock\_controller(): Node("autodock\_controller"){
    2. this->declare\_parameter<double>("min\_v",0.1);
    3. this->get\_parameter("min\_v", min\_v);
    4. this->declare\_parameter<double>("min\_w",0.1);
    5. this->get\_parameter("min\_w" ,min\_w);
    6. this->declare\_parameter<double>("max\_v",0.3);
    7. this->get\_parameter("max\_v" , max\_v);
    8. this->declare\_parameter<double>("max\_w",0.3);
    9. this->get\_parameter("max\_w" , max\_w);
    10. this->declare\_parameter<double>("dist\_to\_dock",0.22);
    11. this->get\_parameter("dist\_to\_dock" ,dist\_to\_dock);
    12. this->declare\_parameter<double>("dist\_to\_center",0.03);
    13. this->get\_parameter("dist\_to\_center" , dist\_to\_center);
    14. this->declare\_parameter<double>("threshold\_v",0.3);
    15. this->get\_parameter("threshold\_v" , threshold\_v);
    16. this->declare\_parameter<double>("threshold\_w",0.4);
    17. this->get\_parameter("threshold\_w" , threshold\_w);
    18. this->declare\_parameter<std::string>("base\_frame\_id","base\_link");
    19. this->get\_parameter("base\_frame\_id" , base\_frame\_id);
    20. vel\_pub = this->create\_publisher<geometry\_msgs::msg::Twist>("cmd\_vel",1);
    21. buffer\_dock = std::make\_unique<tf2\_ros::Buffer>(this->get\_clock());
    22. buffer\_odom = std::make\_unique<tf2\_ros::Buffer>(this->get\_clock());
    23. tf\_listener\_dock = std::make\_unique<tf2\_ros::TransformListener>(\*buffer\_dock);
    24. tf\_listener\_odom = std::make\_unique<tf2\_ros::TransformListener>(\*buffer\_odom);
    25. }
    26. void run();
21. private:
    1. void receive\_tf();
    2. void setVel(float x, float y, float yaw, boost::array<float,2> robot\_point);
    3. rclcpp::Publisher<geometry\_msgs::msg::Twist>::SharedPtr vel\_pub;
    4. std::unique\_ptr<tf2\_ros::Buffer> buffer\_dock;
    5. std::unique\_ptr<tf2\_ros::Buffer> buffer\_odom;
    6. std::unique\_ptr<tf2\_ros::TransformListener> tf\_listener\_dock ,tf\_listener\_odom ;
    7. double min\_v, min\_w, max\_v, max\_w , dist\_to\_dock, dist\_to\_center,threshold\_v, threshold\_w;
    8. std::string base\_frame\_id;
    9. int step = 0;
    10. int step1\_count = 0 , step2\_count = 0;
    11. int split\_num = 2;
    12. float tune\_distense = 0.05;
    13. float tune\_threshold = 0.5;
    14. float x\_origin;
    15. boost::array<float, 2> odom , robot\_point\_temp;
    16. geometry\_msgs::msg::TransformStamped tf\_dock , tf\_odom;
22. };
23. }

**EXPLANATION**

* **Header Includes**:
  + Includes necessary headers for TF2, ROS messages (geometry\_msgs, sensor\_msgs), ROS 2 (rclcpp), and other libraries (chrono, boost).
* **Namespace automatic\_parking**:
  + Encloses the autodock\_controller class within a namespace to avoid naming conflicts.
* **Class autodock\_controller**:
  + Inherits from rclcpp::Node, making it a ROS 2 node capable of communication and lifecycle management.
* **Public and Private Members**:
  + **Constructor (autodock\_controller())**: Initializes parameters and sets up ROS 2 publishers (vel\_pub) and TF2 buffers/listeners (buffer\_dock, buffer\_odom, tf\_listener\_dock, tf\_listener\_odom).
  + **Public Method run()**: Entry point for running the controller logic.
* **Private Methods**:
  + receive\_tf(): Retrieves TF transformations (tf\_dock, tf\_odom) from the robot's base frame to the dock frame (dock\_frame) and odometry frame (odom).
  + setVel(float x, float y, float yaw, boost::array<float,2> robot\_point): Computes and sets linear and angular velocities based on docking logic and current robot state.
* **ROS 2 Components**:
  + **Publishers (vel\_pub)**: Publishes velocity commands (geometry\_msgs::msg::Twist) to control the robot's movement.
  + **TF2 Buffers (buffer\_dock, buffer\_odom)**: Stores TF transformations over time for dock\_frame and odom.
  + **TF2 Listeners (tf\_listener\_dock, tf\_listener\_odom)**: Listens for TF transformations and updates the buffers accordingly.
* **Parameters and Variables**:
  + Various parameters (min\_v, min\_w, etc.) and variables (step, step1\_count, etc.) used for controlling docking behavior and state management.

**Controller.cpp**

1. #include "controller.h"
2. using namespace automatic\_parking;
3. // Setting linear and angular velocity
4. void autodock\_controller::setVel(float x, float y, float yaw, boost::array<float,2> robot\_point){
5. static geometry\_msgs::msg::Twist vel\_msg;
6. double ideal\_theta\_1 = 0;
7. double ideal\_theta\_2 = -1.57; //angle
8. double angle\_threshold = 0.05; //maximum error of angle
9. if (step == 0){
10. vel\_msg.linear.x = 0;
11. RCLCPP\_INFO(get\_logger(),"docking......, step\_1\_spin\_%d",step1\_count);
12. if (((-(ideal\_theta\_1)-angle\_threshold)<=yaw) && (yaw<=(-(ideal\_theta\_1)+angle\_threshold))){
    1. vel\_msg.angular.z = 0;
    2. vel\_pub->publish(vel\_msg);
    3. step = 1;
    4. if (step1\_count==0){
    5. x\_origin = x;
    6. robot\_point\_temp = robot\_point ;
    7. }
13. }
14. else if (((-(ideal\_theta\_1)+angle\_threshold)<=yaw) && (yaw<(-(ideal\_theta\_1)+threshold\_w))){
    1. vel\_msg.angular.z = min\_w;
15. }
16. else if (((-(ideal\_theta\_1)+threshold\_w)<=yaw) && (yaw< 3.14)){
    1. vel\_msg.angular.z = max\_w;
17. }
18. else if (((-(ideal\_theta\_1)-threshold\_w)<=yaw) && (yaw<(-(ideal\_theta\_1)-angle\_threshold))){
    1. vel\_msg.angular.z = -min\_w;
19. }
20. else {
    1. vel\_msg.angular.z = -max\_w;
21. }
22. }
23. else if (step == 1){
24. RCLCPP\_INFO(get\_logger(),"docking......, step\_1\_move\_%d",step1\_count);
25. //printf("%f\n",dist(robot\_point\_temp,robot\_point));
26. if ((fabs(x) <= dist\_to\_center) or (dist(robot\_point\_temp,robot\_point) >= fabs(x\_origin/split\_num))){
    1. vel\_msg.linear.x = 0;
    2. vel\_pub->publish(vel\_msg);
    3. step1\_count += 1;
    4. if (fabs(x) <= dist\_to\_center){
    5. step = 2;
    6. }
    7. else{
    8. robot\_point\_temp = robot\_point;
    9. step = 0;
    10. }
27. }
28. else if (x<0){
    1. if ((dist\_to\_center<fabs(x)) && (fabs(x)<(dist\_to\_center+threshold\_v))){
    2. vel\_msg.linear.x = -min\_v;
    3. }
    4. else {
    5. vel\_msg.linear.x = -max\_v;
    6. }
29. }
30. else if (x>0){
    1. if ((dist\_to\_center<fabs(x)) && (fabs(x)<(dist\_to\_center+threshold\_v))){
    2. vel\_msg.linear.x = min\_v;
    3. }
    4. else {
    5. vel\_msg.linear.x = max\_v;
    6. }
31. }
32. }
33. else if (step == 2){
34. RCLCPP\_INFO(get\_logger(),"docking......, step\_2\_spin\_%d",step2\_count);
35. if (((-(ideal\_theta\_2)-angle\_threshold)<=yaw) && (yaw<=(-(ideal\_theta\_2)+angle\_threshold))){
    1. vel\_msg.angular.z = 0;
    2. vel\_pub->publish(vel\_msg);
    3. step = 3;
    4. if (step2\_count==0){
    5. x\_origin = x;
    6. robot\_point\_temp = robot\_point ;
    7. }
36. }
37. else if (((-(ideal\_theta\_2)+angle\_threshold)<=yaw) && (yaw<(-(ideal\_theta\_2)+threshold\_w))){
    1. vel\_msg.angular.z = min\_w;
38. }
39. else if (((-(ideal\_theta\_2)+threshold\_w)<=yaw) && (yaw<(3.14))){
    1. vel\_msg.angular.z = max\_w;
40. }
41. else if (((-(ideal\_theta\_2)-threshold\_w)<=yaw) && (yaw<(-(ideal\_theta\_2)-angle\_threshold))){
    1. vel\_msg.angular.z = -min\_w;
42. }
43. else {
    1. vel\_msg.angular.z = -max\_w;
44. }
45. }
46. else if (step == 3){
47. RCLCPP\_INFO(get\_logger(),"docking......, step\_2\_move\_%d",step2\_count);
48. if ((fabs(x)<dist\_to\_dock) or (dist(robot\_point\_temp,robot\_point) >= fabs(x\_origin/split\_num)) or ((fabs(y) > tune\_distense) and (fabs(x) <= tune\_threshold) )){
    1. vel\_msg.linear.x = 0;
    2. vel\_pub->publish(vel\_msg);
    3. step2\_count += 1;
    4. if (fabs(x)<dist\_to\_dock){
    5. step = 4;
    6. }
    7. else if ((fabs(y) > dist\_to\_center) and (fabs(x) <= 0.6)){
    8. step = 0;
    9. robot\_point\_temp = robot\_point;
    10. }
    11. else{
    12. robot\_point\_temp = robot\_point;
    13. step = 2;
    14. }
49. }
50. else if (fabs(x)<=(dist\_to\_dock+threshold\_v)){
    1. vel\_msg.linear.x = -min\_v;
51. }
52. else {
    1. vel\_msg.linear.x = -max\_v;
53. }
54. }
55. else if (step == 4){
56. vel\_msg.linear.x = 0;
57. vel\_msg.angular.z = 0;
58. vel\_pub->publish(vel\_msg);
59. RCLCPP\_INFO(get\_logger(),"Finish Docking!");
60. }
61. vel\_pub->publish(vel\_msg);
62. }
63. void autodock\_controller::receive\_tf(){
64. while(rclcpp::ok()){
65. try{
66. //autodock\_controller::tf\_listener\_.waitForTransform("base\_link","dock\_frame",rclcpp::Time(0),rclcpp::Duration(3.0));
    1. tf\_dock = buffer\_dock->lookupTransform("base\_link","dock\_frame",tf2::TimePoint(std::chrono::seconds(0)),tf2::Duration(std::chrono::seconds(3)));
    2. //tf\_odom = buffer\_odom->lookupTransform("base\_link","dock\_frame",tf2::TimePoint(std::chrono::milliseconds(0)),tf2::Duration(std::chrono::seconds(3)));
    3. //tf\_dock = buffer\_dock->lookupTransform("base\_link","dock\_frame",tf2::TimePoint(std::chrono::milliseconds(3000)),tf2::Duration(3));
    4. break;
67. }
68. catch (tf2::TransformException &ex){
    1. RCLCPP\_ERROR(get\_logger(),"%s",ex.what());
    2. RCLCPP\_ERROR(get\_logger(),"Did not find the pattern!");
    3. rclcpp::sleep\_for(std::chrono::seconds(3));
69. }
70. }
71. //tf\_listener\_.waitForTransform("odom","base\_link",ros::Time(0),ros::Duration(3.0));
72. tf\_odom = buffer\_odom->lookupTransform("odom","base\_link",tf2::TimePoint(std::chrono::milliseconds(0)),tf2::Duration(std::chrono::seconds(3)));
73. //tf\_odom = buffer\_odom->lookupTransform("odom","base\_link",tf2::Duration(3));
74. }
75. void autodock\_controller::run(){
76. receive\_tf();
77. // and yaw
78. float dock\_x = tf\_dock.transform.translation.x;
79. float dock\_y = tf\_dock.transform.translation.y;
80. float dock\_yaw = tf2::getYaw(tf\_dock.transform.rotation);
81. odom[0] = tf\_odom.transform.translation.x;
82. odom[1] = tf\_odom.transform.translation.y;
83. setVel(dock\_x, dock\_y, dock\_yaw, odom);
84. }
85. int main(int argc, char\*\* argv){
86. rclcpp::init(argc, argv);
87. auto autodock\_controller\_node = std::make\_shared<automatic\_parking::autodock\_controller>();
88. rclcpp::WallRate rate(20.0);
89. while(rclcpp::ok()){
90. autodock\_controller\_node->run();
91. rclcpp::spin\_some(autodock\_controller\_node);
92. rate.sleep();
93. }
94. }

**EXPLANATION**

**Include and Namespace**:

* + Includes the controller.h header file that defines the autodock\_controller class within the automatic\_parking namespace.

**Main Function (main())**:

* + Initializes ROS 2 using rclcpp::init(argc, argv).
  + Creates an instance of autodock\_controller node.
  + Sets up a control loop that calls run() method of autodock\_controller to execute the docking logic continuously.
  + Spins ROS 2 callbacks using rclcpp::spin\_some() and maintains a fixed loop rate using rclcpp::WallRate.

**Class Method Implementations**:

* + **Constructor (autodock\_controller::autodock\_controller())**: Initializes ROS 2 parameters (min\_v, min\_w, etc.), creates a publisher for velocity commands (vel\_pub), and initializes TF2 buffers and listeners (buffer\_dock, buffer\_odom, tf\_listener\_dock, tf\_listener\_odom).
  + **setVel(float x, float y, float yaw, boost::array<float,2> robot\_point)**: Implements the docking control logic in several steps (step).
    - Adjusts angular velocity (vel\_msg.angular.z) based on the current yaw angle to align the robot with the dock.
    - Adjusts linear velocity (vel\_msg.linear.x) to move the robot towards the dock, handling various conditions (step1, step2) until docking is completed (step4).
  + **receive\_tf()**: Retrieves TF transformations (tf\_dock, tf\_odom) from the robot's base frame to the dock frame (dock\_frame) and odometry frame (odom).
    - Uses buffer\_dock and buffer\_odom to lookup transformations.
    - Handles exceptions (tf2::TransformException) if the transformation lookup fails, logging errors and retrying after a delay.
  + **run()**: Entry point for running the docking controller logic.
    - Calls receive\_tf() to update TF transformations.
    - Extracts position (dock\_x, dock\_y) and yaw angle (dock\_yaw) from TF transformations.
    - Calls setVel() to compute and publish velocity commands based on the current robot state.

### **INPUT AND OUTPUT:**

**Input**:

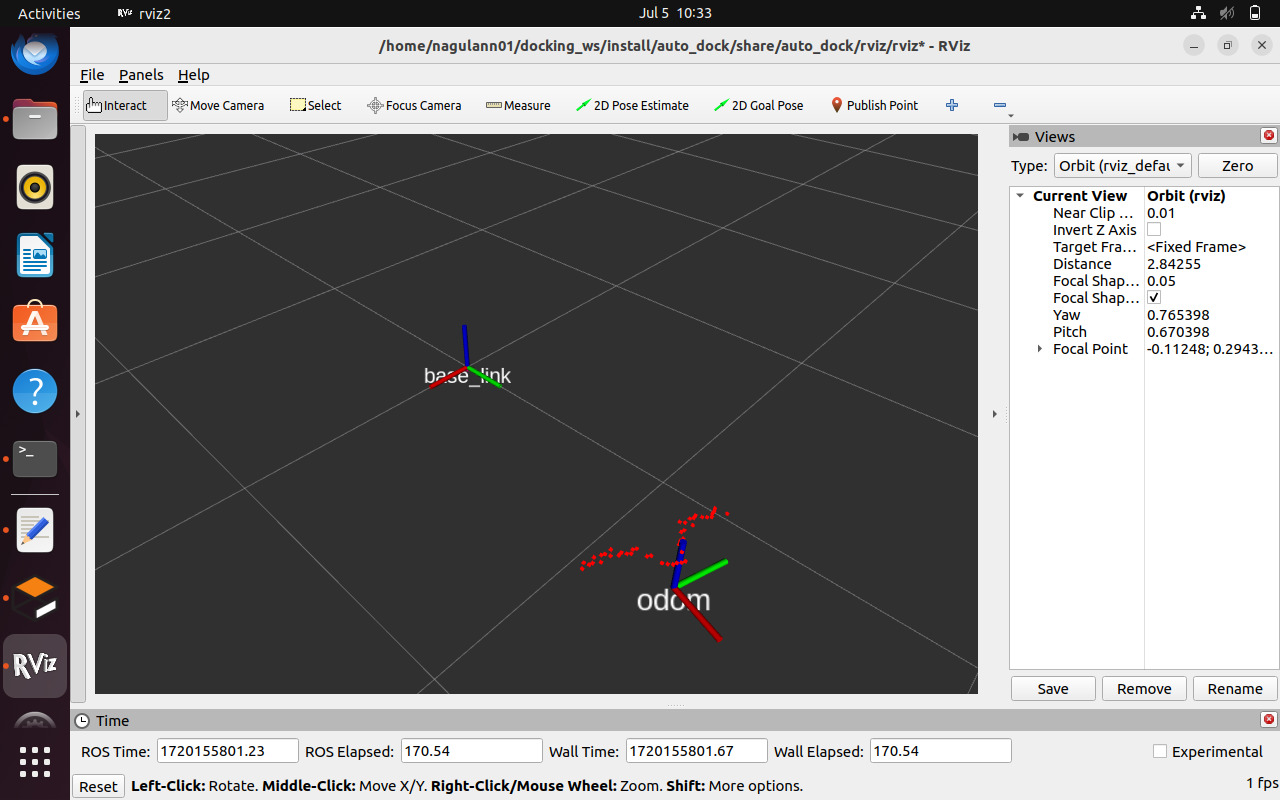
* + Parameters (min\_v, min\_w, max\_v, etc.) are initialized from the ROS parameter server within the constructor (autodock\_controller::autodock\_controller()).
  + TF transformations (tf\_dock, tf\_odom) are received and updated in the receive\_tf() method based on data from external sources (TF broadcasts).

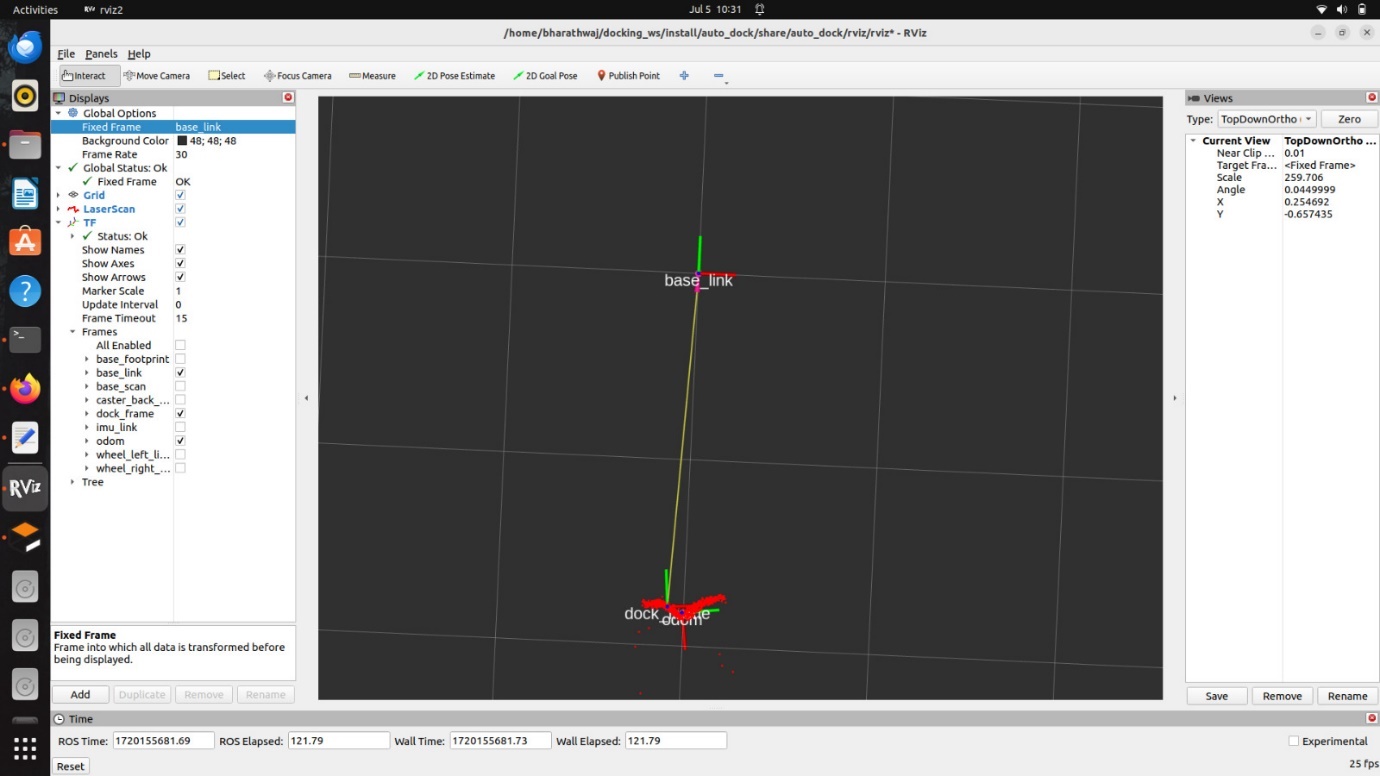
**Output**:

* + Velocity commands (geometry\_msgs::msg::Twist) are published to the cmd\_vel topic (vel\_pub->publish(vel\_msg)).
  + Logging messages (RCLCPP\_INFO, RCLCPP\_ERROR) provide status updates and error handling information to the ROS console.

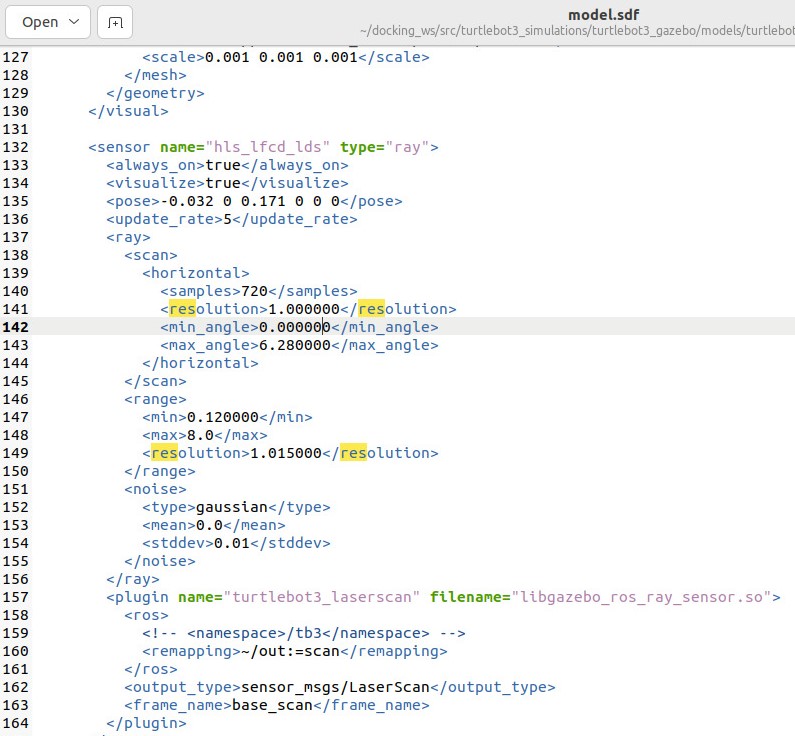
**ERROR solved:**

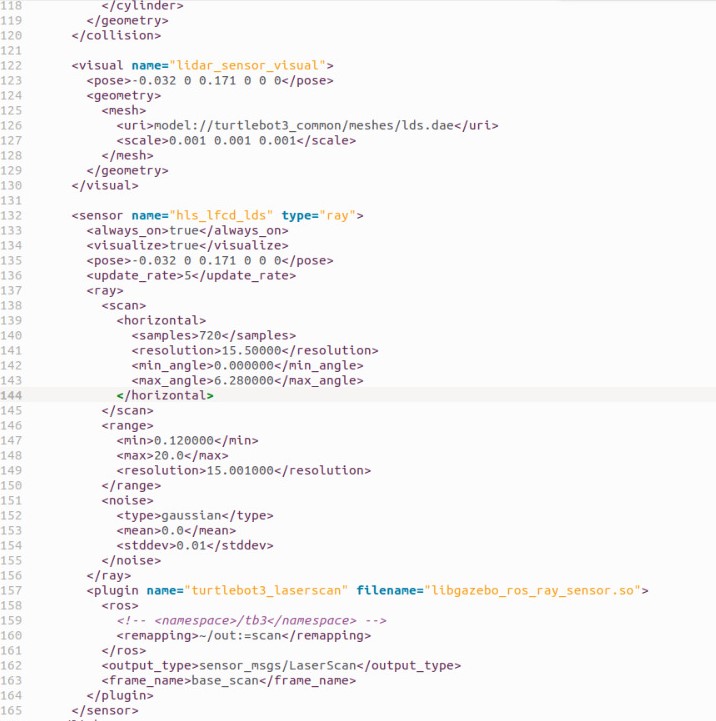
* If the dock does not happen correctly in RViz, consider adjusting parameters such as 'distance to dock' or 'distance to center' to resolve the issue.



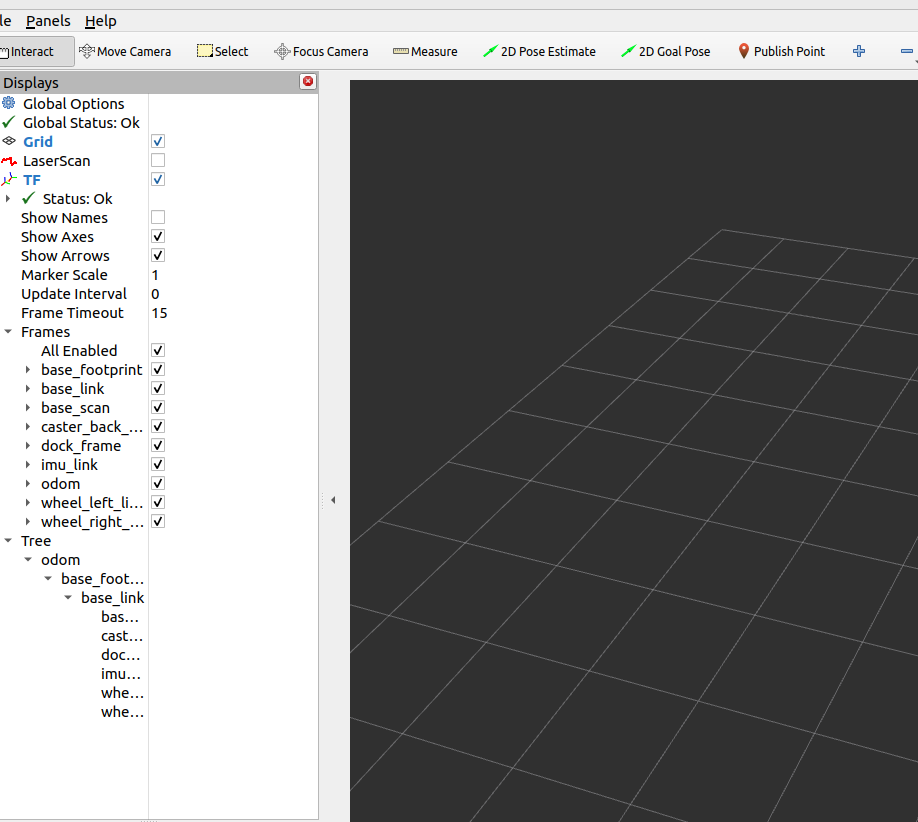


* The dock frame was not visible in RViz, indicating an error. By adjusting and increasing the resolution, the dock frame became visible.





* We initially set the odom frame as the parent and the dock frame as the child, but the correct configuration is to have the base\_link as the parent and the dock frame as the child.



* Keep the max\_angle above 17 to perfectly dock.

**FOLDER :**

Docking\_ws

* src
  + automatic\_parking
    - auto\_dock
      * include
        + auto\_dock.h
        + controller.h
        + pattern.h
      * launch
        + auto\_humble.launch.py
      * rviz
        + rviz.rviz
      * src
        + controller.cpp
        + pattern.cpp
      * CMakeLists.txt
      * Package.xml
    - Model
      * + dock1

model.config

model.sdf

* + - * + dock2

model.config

model.sdf

* + - readme\_resource
    - README.md
  + DynamixelSDK
    - * dynamixel\_sdk

include

src

CHANGELOG.rst

CMakeLists.txt

Package.xml

* + - * dynamixel\_sdk\_custom\_interfaces
      * dynamixel\_sdk\_examples
      * CONTRIBUTINH.md
      * Doxyfile
      * LICENSE
      * README.md
  + laser\_line\_extraction

build

images

include

line.h

line\_extration.h

line\_extraction\_node.h

line\_extraction\_ros.h

install

laser\_line\_msgs

log

msg

src

line.cpp

line\_extration.cpp

line\_extract ion\_node.cpp

line\_extraction\_ros.cpp

CMakeLists.txt

config

LICENSE

pakage.xml

README.md

* + laser\_line\_msgs

msg

LineSegmet.msg

LineSegmentList.msg

CMakeList.txt

pakage.xml

README.md

* + turtlebot3
  + turtlebot3\_msgs
  + turtlebot3\_simulations

turtlebot3\_fake\_node

Turtlebot3\_gazebo

Turtlebot3\_simulations

CONTRIBUTING.md

LICENSE

README.md

* Create workspace

Mkdir -p ~/docking\_ws /src

* Copy and paste the package
* Source your ws

Source install/setup.bash

* Build your ws

Colcon build

* Launch your gazebo file

Ros2 launch turtlebot3\_gazebo empty\_world.launch.py

* Launch your auto dock file

Ros2 launch auto\_dock auto\_humble.launch.