Docking Solution for Autonomous Mobile Robots

Christopher Rowe

github.com/rwbot/docking

Supervised by Professor Deborah Fixel

Department of Engineering

Trinity College

May 4, 2020

A senior engineering capstone project submitted in partial fulfilment of the requirements for the degree of Bachelor of Science.

Abstract

Docking is a specific utilization of landmark localization – comparing detected local features against a known map containing unique "landmark" features to determine current local coordinates. This capstone aims to develop a ROS package that adds general docking functionality to an autonomous mobile robot of differential drive configuration. The package must be capable of determining a valid pose-sequence that can mate the robot with the dock. This design report details the system components of the proposed docking solution, the perception pipeline. The design and implementation of each of the stages within the perception pipeline is described, and the results obtained from running the perception pipeline are provided. Similarly, the design and implementation of the motion planner is described and analyzed, and results are provided.

Contents

List of Figures

List of Abbreviations

LiDAR	Light Detection and Ranging	1
ROS	Robot Operating System	2
CAD	Computer Aided Design	
RANSAC	Random Sample Consensus	9
ICP	Iterative Closest Point	11

1

Introduction

1.1 | Motivation

As the versatility of mobile robots increases, there grows a need for them to perform increasingly complex tasks. One such task is docking. Docking is the process of joining two bodies in an orientation that correctly mates the positive and negative features of the bodies's respective geometries. In mobile robotic applications, docking facilitates the interaction between a robot and a specific tool body. A tool body is any other body onto which the robot can perform an operation, or from which to accept an operation.

The most common application of docking would involve: a power station where the robot can recharge, a transportation platform that can move the robot, or a transportable body requiring unique pick-and-place procedures, such as a forklift precisely engaging a wood pallet. However, to achieve docking, the robot must be able to correctly estimate its pose relative to the dock and execute motion commands with single digit centimeter accuracy.

1.2 | Similar Work

There exist two commercially available robots that feature docking functionality. The first is the Fetch Robot, an autonomous mobile robot designed for use in warehouses. Using the onboard 2D Light Detection and Ranging (LiDAR), the robot is capable of recognizing the dock profile and steering onto the dock to charge itself. Unfortunately, the functionality is specific to the Fetch Robot and its proprietary dock.

The second is the Turtlebot/Kobuki research robot. This uses a dock equipped with three infrared emitters, each specifying a zone. On the robot, are three infrared sensors which detect the infrared light. Depending on which zone is detected by each sensor, the robot's pose relative to the docking station can be inferred. This implementation requires the user to have a Kobuki/Turtlebot robot and purchase the docking station. While the software used for docking is open source, to use it with their dock with a different robot would require the addition of the specified infrared sensors, as opposed to this project's proposed approach of using the onboard sensors. These two solutions are constrained to the robot platform, docking station and application. It requires the purchase of a proprietary dock or additional design requirement for infrared sensor integration.

1.3 | Approach

The proposed approach will use the existing onboard sensor (LiDAR or RGBD camera), will not be constrained to a proprietary docking station, and can be generalized to a variety of localization tasks requiring 10cm accuracy. It will require the user to provide the 3D CAD model of dock in an .STL format. The docking functionality will work as a Robot Operating System (ROS) package as an additional feature for an already autonomous robot. The process of docking can be divided into two stages. The first stage implements the perception pipeline required for converting raw sensor data into a set of position and orientation coordinates of the dock. The second stage implements the motion planning required to move the robot from its current position to the docking position.

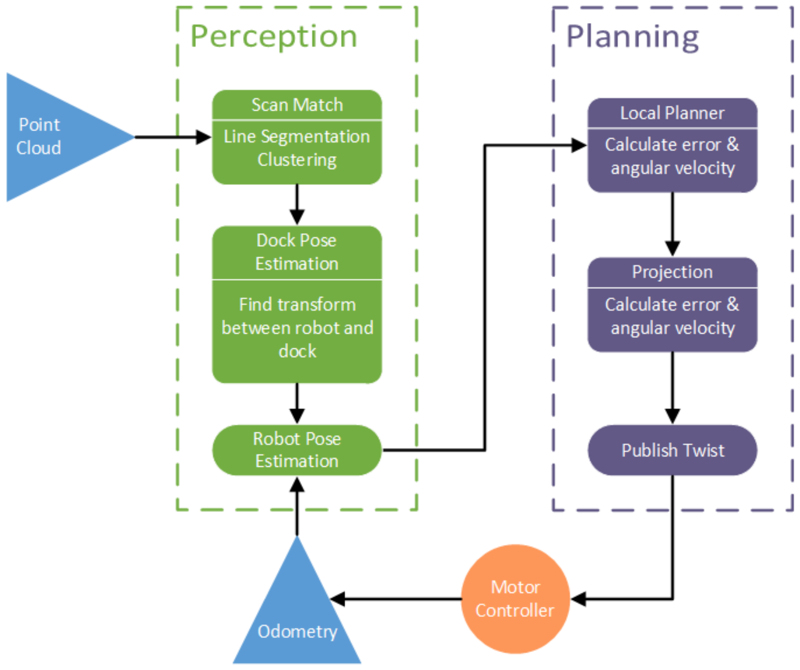


Figure 1.1: Docking System Architecture

2

Background

2.1 | Localization

The problem of estimating the robot's pose is called localization. Specifically, localization is the process by which a robot determines its pose (representation of its position and orientation) within a parent coordinate frame. The accuracy of localization ultimately comes down to the resolution and accuracy of the sensors being used, which influence the error propagation in odometry over time.

Although integrating pose estimates in higher quantities and from earlier times allows the current pose estimate to be more complete, more uncertainty is incorporated per reading, and the error grows exponentially over time. Thus, localization algorithms that factor in the entire sequence of the pose-graph are especially vulnerable to errors which inflate the variance of the current pose estimate.

In the majority of industrial applications of fixed, non-mobile robots in highly controlled environments, sub millimeter precision can be achieved. However, the average mobile robot in a dynamic and uncertain environment can achieve a localization accuracy of anywhere between 10cm-100cm, depending on the type and precision of the sensors and the total number of integrated poses. Although there exists advanced localization methods for mobile robots that provide sub-centimeter accuracy, such products are extremely expensive and require very specific environments and specialized hardware.

The most common workaround involves installing "landmarks" throughout the robot's workspace such as radio transceivers, barcodes, embedded floor rails etc. However these workarounds are not ideal as they incur costs as it requires more sensor hardware and software, in addition to the already expensive LiDAR scanner standard to autonomous robotic platforms, and changes to building infrastructure, while also requiring a tedious calibration process.

2.2 | Landmark Localization

Docking is a specific utilization of landmark localization – comparing detected local features against a known map containing unique "landmark" features to determine current local coordinates. Any task given to a mobile robot requires it to move from its current position and orientation to a goal position and orientation. Without knowing where it is with respect to the goal pose, it is unable to determine a path that can successfully get it to the goal pose. As such, docking and localization in general are fundamental components of any system involved in robotics, and autonomous navigation. Most common is the docking of spacecraft, docking of robot vacuum robots to their recharging station.

2.3 | Robot Operating System

ROS is a software framework created to standardize robotics development tools. By creating a standard for robotics software, developers can focus on the technology itself, rather than the implementation of the technology.

Software are provided through modules called packages, each of which provides a unique functionality. Among the packages available through ROS, there are some that perform odometry, path planning, localization, sensor data processing and much more.

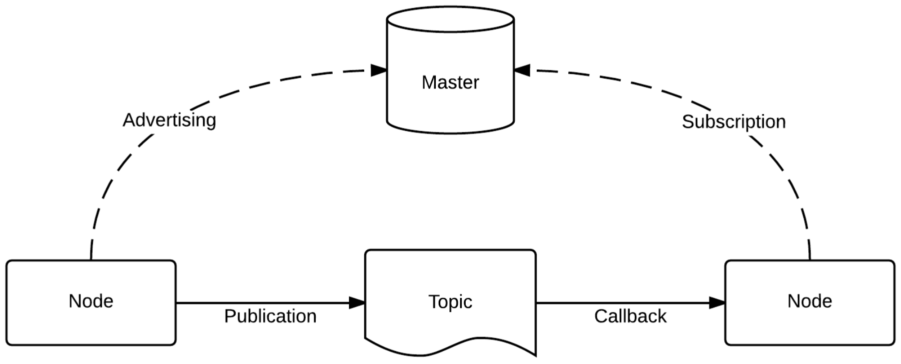


Figure 2.1: Basic overview of ROS

Project

3.1 | Goals

The objective of this capstone is to develop a ROS package that adds docking functionality to an autonomous mobile robot of differential drive configuration. The package must be able to capable of determining a valid pose-sequence that can mate the robot with the dock. Specifically, the demonstrative goal of this capstone is for an autonomous mobile robot equipped with a sensor providing point cloud data, to successfully recognize a specified dock geometry, localize within the dock's coordinate frame and subsequently execute a pose-sequence to mate the robot and dock bodies with an error no greater than 10cm. The goals are outlined as follows:

1. Extract the dock cluster from a raw LiDAR scan
2. Calculate the pose of the dock from the robot
3. Publish linear and angular velocities that lead the robot to the dock
4. Publish a ROS package that adds docking functionality to any LiDAR equipped robot

3.2 | Constraints

3.2.1 | LiDAR Resolution

Depending on the distance accuracy of a LiDAR, the resulting scan can look nothing like the scanned surface. However, to standardize functionality, the docking package must accommodate all levels of LiDAR.

3.2.2 | Processing Power

Computations performed during the perception pipeline and motion planning can put a significant load on the robot's processor. When this occurs, the computations slow down thus lowering the frequency of the program and its precision.

3.3 | Timeline

The project was divided into two stages. The first stage implements the perception pipeline required for converting raw sensor data into a set of position and orientation coordinates of the dock. The second stage implements the motion planning required to move the robot from its current position to the docking position. The second stage includes development of a test robot platform for a demonstration of the docking functionality, and also submitting the package to the ROS public repository. The fall semester focused on the first stage while the spring semester focused on the second stage.

4

Perception Pipeline

The perception pipeline describes the sequence of processes performed on sensor data in order to remove noise and extract meaningful features. The pipeline consists of six steps. First, the output from the LiDAR is taken in as a point cloud. The point cloud is then divided into smaller groups using density-based clustering. Each cluster is then run through a line detection algorithm. The detected lines are then parameterized, and those parameters are compared with the known parameters of the dock. Once the cluster representing the dock is identified, the pose of the dock is calculated and output from the perception pipeline.

4.1 | LiDAR

Various sensors can be used to extract data about the environment. For range-based perception, the most commonly used sensors are LiDAR sensors and colored depth (RGBD) cameras. Both of these sensors represent the depth map of the environment using a data structure known as a Point Cloud. A point cloud is a data structure where each point corresponds to a point on an object detected at that point in space. Point clouds generated from sensors only represent a point on a surface. Images are dimensioned as X pixels by Y pixels, where pixel is a portmanteau of picture element. Similarly, a point cloud can be thought of as a 3D image, made up of "volume elements" (voxels) dimensioned as X voxels by Y voxels by Z voxels. From a 3D LiDAR, the generated point cloud represents all the points in X, Y, Z space at which a depth was detected. A 2D LiDAR outputs a similar point cloud, with the difference being that only points at the Z height of the LiDAR will be detected. An RGBD Camera outputs the same 3D point cloud as the 3D LiDAR, but along with their X, Y and Z values, each point also has an RGB value representing the color of that point. The approach of the implemented algorithm currently accepts only 2D point clouds representing a planar laser scan of the environment. Data from 3D point clouds from RGBD cameras must be first converted into a planar laser scan before being passed into the pipeline. The raw input point cloud represents a list of (X,Y) coordinates in space where obstacles are detected.

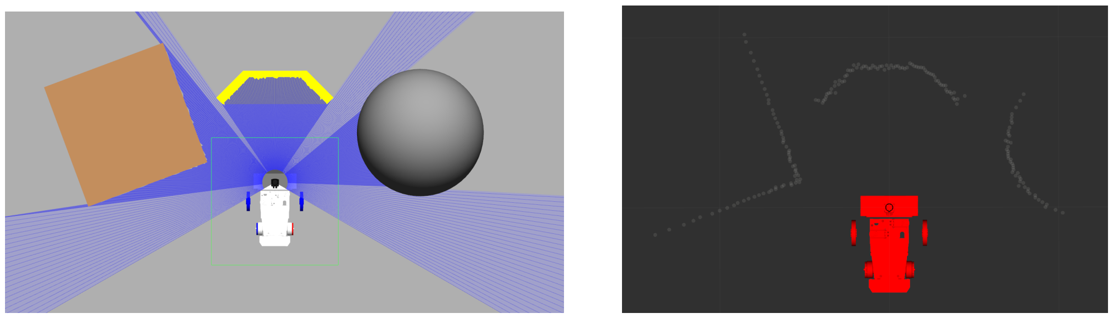

(a) Simulation Environment (b) Raw cloud

Figure 4.1: LiDAR Scan of Simulation Environment

4.2 | Point Clustering

To extract the dock, the points that belong to the dock must first be identified. Rather than process the entire point cloud, it is computationally more efficient to handle subsets of points. Since each point in the point cloud corresponds to a point on a detected surface, points close together can represent the surface of an object. Separating a point cloud into subsets of points, or clusters, is known as clustering.

Many different clustering algorithms exist. The algorithm used in this approach performs Euclidean Clustering using Nearest Neighbours (NN). The algorithm works by first organizing the point cloud into a kD-Tree structure. This organizes the points, in ascending order, based on their X values, and then by their Y values. This allows spatially close points to be stored in memory close together, which simplifies the algorithm and reduces computation time that would be spent repeatedly iterating through an unordered list.

Using the kD-Tree, the algorithm iterates through each point and calculates the euclidean distance between the current points and a number of its closest neighbours. If the euclidean distance between a neighbour point and the current point is below the specified threshold radius, then those two points are considered to be part of the same cluster. Once all the neighbours of the current point are checked, the current point is marked as “checked”, and the algorithm now considers another “unchecked” point previously identified as within the cluster of the previous point. If there exists no more “unchecked” points within the cluster of the previous point, the algorithm extracts all the points within the current cluster from the input cloud. The algorithm then moves on to process a new "unchecked" point, which begins the new cluster. The output of this algorithm gives us a set of clusters, each containing a list of points that are considered to belong to the same detected object.

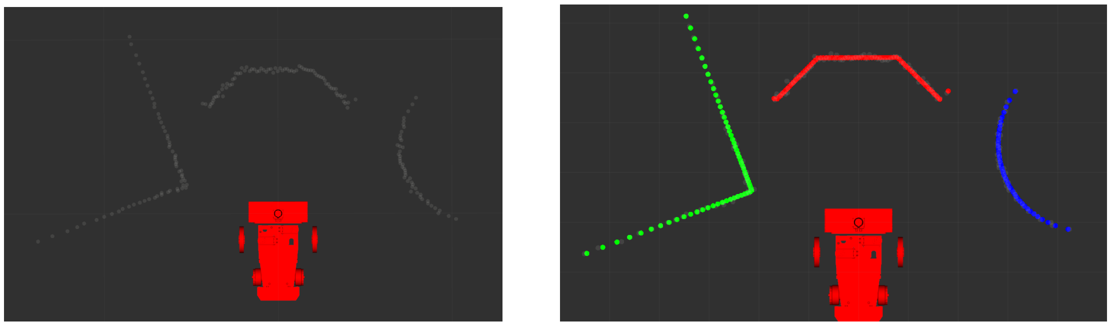
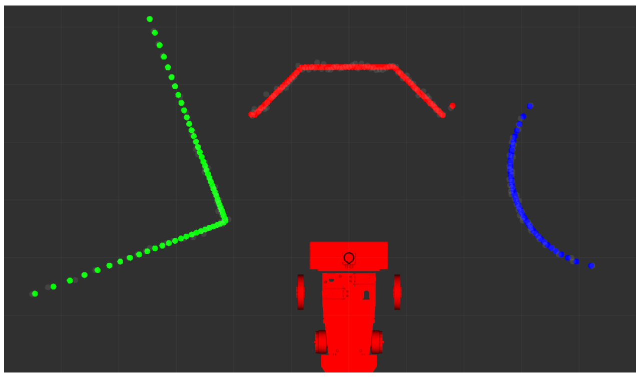

(a) Raw cloud (b) Clusters

Figure 4.2: Extracted clusters

4.3 | Line Detection

The main feature of the perception pipeline is the extraction of lines to identify the dock. Once the raw point cloud has been divided into clusters, each cluster each is processed to extract any existing lines. The line extraction algorithm used is called Random Sample Consensus (RANSAC). The algorithm works by picking two random points and calculates the line defined by those two points. The algorithm then iterates through all points in the cloud and calculates the perpendicular distance between the line and the point. All points within a specified threshold distance are marked as inliers for that line, and all points exceeding that specified threshold are marked as outliers for that line. The algorithm then selects another pair of points, and repeats the process until all pairs of points have been checked, or until a specified number of maximum iterations. The points from the line with the most inliers are then extracted from that cluster's point cloud. Once the point cloud of the detected line has been extracted, the centroid of the cloud is calculated. The algorithm then repeats from the beginning until all lines within the cloud have been extracted.


(a) Clusters

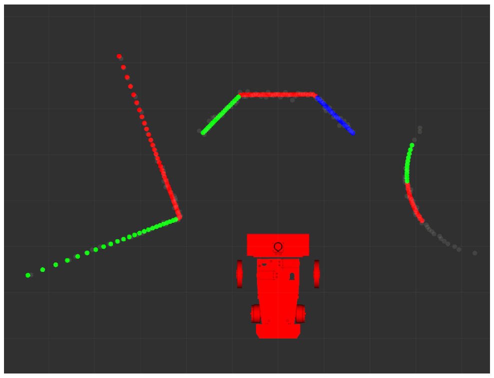

(b) Detected lines

Figure 4.3: Lines detected within each cluster

4.4 | Line Parameterization

Once the RANSAC line detection algorithm is complete, each of the detected lines is packaged into a Line message. First the array of inlier points and the line's model coefficients are added to the message. The model coefficients correspond to the coefficients for the general form equation of the line (Ax+By=C). Next, the centroid of the line's point cloud and the length between the two endpoints are calculated and added to the message. Finally, the ID number of the cluster the line was detected within is added to the message. Once each line has been packaged, all the Line messages are then packaged into a LineArray message containing an array of all the Line messages. The LineArray message is then added to the Cluster message from which the lines were detected.

4.5 | Parameter Matching

To match the dock model with the detected lines, the dock was similarly parameterized to define unique relationships between its geometries. Figure A.1 shows the dimensions of the dock model used in testing. Each cluster is iterated through, and in each cluster, each line in the LineArray is compared against the other lines. Once the parameters have been found to be fulfilled, within a specified tolerance, the containsDock boolean value of that cluster is set to True.

4.6 | Pose Estimation

The final step in the perception pipeline calculates the pose of the dock. This step is only called once the dock has been detected (ie. when a Cluster message with containsDock set to True is found). Initially, Principal Component Analysis was used to determine the pose of the point cloud. However, this method proved to be dependent on the shape of the dock. Thus, the pose estimate method was switched to Iterative Closest Point (ICP).

The ICP algorithm works by minimizing the distance between common points for a reference and target point cloud. The reference cloud is kept fixed, while the target cloud Undergoes a translation and rotation. For each point on the target, the distance to the closest point on the reference is calculated and summed.

This process is repeated until the sum of distances is minimized. To reduce the number of iterations and lighten the processing load, the centroid of each target object is calculated and used to initially align the target within the centroid of the reference as an approximate starting point. When the minimum distance sum is found, the corresponding translation and rotation applied to obtain that sum is returned. This represents the transformation between the robot's coordinate frame and the dock's coordinate frame.

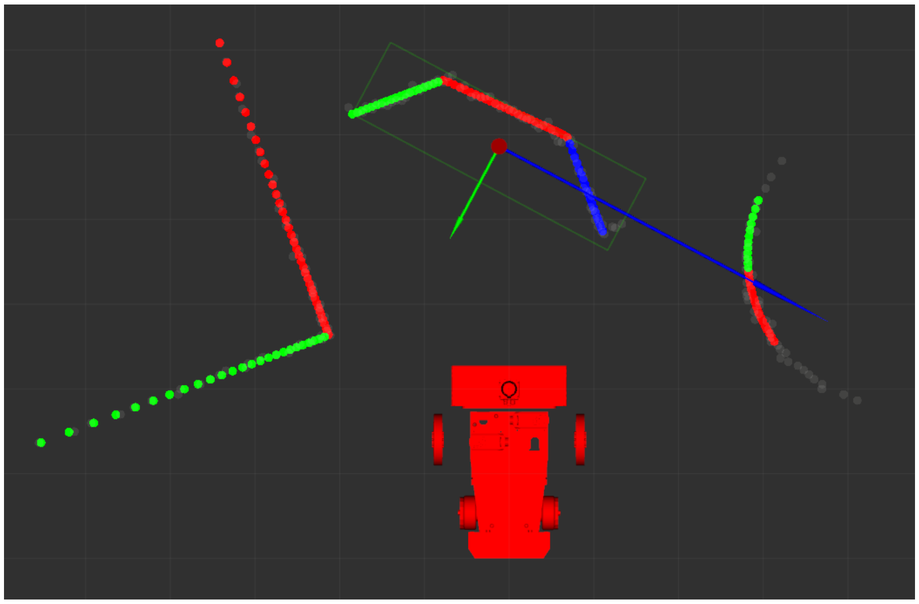


Figure 4.4: Pose given by ICP

5

Motion Planning

5.1 | Twist Message

One of the main advantages of ROS is platform independence. Regardless of the motors and motor controller used on a robot, the standard representation of a velocity is a Twist message. A Twist message expresses velocity in free space broken into its linear and angular parts. Any ROS enabled robot, with any motors and any motor controller, can be commanded to move at a specific velocity through a Twist message. For non-holonomic configurations, only velocities in their valid respective axes are given. Specifically, for the differential-drive configuration, only the linear X and angular Z velocities are specified as those are the only valid axes of motion.

From a motion planning perspective, a plan can be generalized to any robot as a sequence of Twist messages that, when executed, moves the robot along a path. This sequence of velocity targets between the initial pose and target pose is obtained through a control law.

5.2 | Kinematics

To obtain the control law, the system variables must be identified. Here, the robot is at pose P_{Robot}, with origin at rotational center (between the wheels), and the dock target is at pose P_{Target}. A line of sight is drawn from the robot pose P_{Robot} to the target pose P_{Target}. The line of sight denotes the radial distance, $\mathbf{r}$, between the robot and target poses. The angle delta $\boldsymbol{\delta}$ denotes the heading of the robot, which is the orientation of the robot frame with respect to the line of sight. The angle phi $\boldsymbol{\phi}$ denotes the orientation of the target pose with respect to the line of sight. The robot moves with linear velocity $\boldsymbol{v}$ (which is always parallel to the X axis in the robot coordinate frame) and angular velocity, omega $\boldsymbol{\omega}$, about the robot's Z axis.

Using these variables, the equations of motion for the robot can be expressed:

$$\begin{pmatrix} \dot{r} \\ \dot{\phi} \\ \dot{\delta} \end{pmatrix} = \begin{pmatrix} -v\cos\delta \\ \frac{v}{r}\sin\delta \\ \frac{v}{r}\sin\delta + \omega \end{pmatrix} \tag{5.1}$$

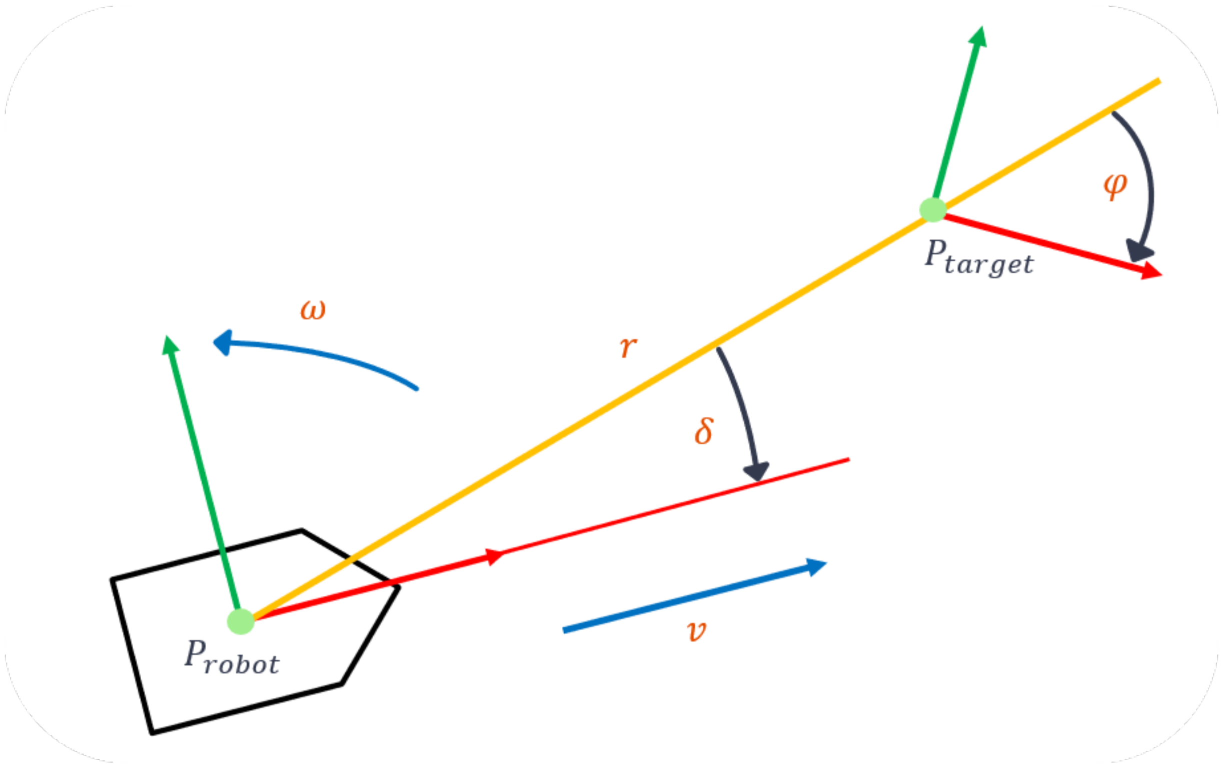


Figure 5.1: Control System Variables

5.3 | Feedback Control Law

The objective of the feedback control law is to determine values for the control variables that will ultimately drive the error variables to zero. From the variables defined in the previous section, the error variables are chosen to be the radial distance, $\mathbf{r}$, the heading, δ, and the target orientation, $\boldsymbol{\phi}$. This leaves the linear and angular velocities, v and $\boldsymbol{\omega}$, as the control variables of the system. Using the difference in distance and orientation between the initial and target poses as feedback error, the control law outputs the linear and angular velocities that will drive the errors to zero. [1]

To simplify the problem, we can let the linear velocity v be a nonzero positive, though not necessarily constant, leaving only the angular velocity $\boldsymbol{\omega}$ as the only control variable. Looking

[1]For more information see **Park, Jong Jin.** "Graceful Navigation for Mobile Robots in Dynamic and Uncertain Environments." (2016). `https://deepblue.lib.umich.edu/handle/2027.42/120760`

at the equations, we notice that the control variable $\boldsymbol{\omega}$ only affects the state variable δ. Furthermore, the states $\mathbf{r}$ and $\boldsymbol{\phi}$ are determined by state δ. States $\mathbf{r}$ and $\boldsymbol{\phi}$ completely represent the position of the robot, while δ represents the steering of the robot. This relationship helps with the analysis as it allows us to decompose the system into two subsystems. The expressions for $\dot{r}$ and $\dot{\boldsymbol{\phi}}$ can completely represent the position of the robot, and so comprises the position subsystem. The expression for $\dot{\delta}$ represents the heading of the robot, and can be referred to as the steering subsystem. The goal now is to find a virtual control δ of the steering subsystem, which will drive the position subsystem to zero, through a real control $\boldsymbol{\omega}$.

$$\begin{pmatrix} \dot{r} \\ \dot{\phi} \end{pmatrix} = \begin{pmatrix} -v\cos\delta \\ \frac{v}{r}\sin\delta \end{pmatrix} \tag{5.2}$$

$$\dot{\delta} = \frac{v}{r}\sin\delta + \omega \tag{5.3}$$

5.3.1 | Position Sub-System

Virtual control δ denotes the reference heading of the robot obtained from the current state ϕ. The virtual control for the position subsystem was chosen such that the function for the path followed an Archimedean spiral, which rotates about the origin with an increasing distance. This property (in reverse) is advantageous because as the virtual control, it moves the robot along a path with a smooth curve, and whose endpoint lies at the origin of the target pose. The gain K_ϕ controls the ratio in which the target orientation, ϕ, changes along with the radial distance, r.

$$\dot{\delta} = arctan(-K_\phi \phi) \tag{5.4}$$

Plugging in the expression for virtual control δ into the position subsystem, the trajectory of the robot along path specified by δ is:

$$\begin{pmatrix} \dot{r} \\ \dot{\phi} \end{pmatrix} = \begin{pmatrix} -v\cos(arctan(-K_\phi \phi)) \\ \frac{v}{r}\sin(arctan(-K_\phi \phi)) \end{pmatrix} \tag{5.5}$$

5.3.2 | Steering Subsystem

Let z denote the difference between the actual heading δ and the desired value $arctan(-K_\phi \phi)$.

$$z \equiv \delta - arctan(-K_\phi \phi) \tag{5.6}$$

Plugging in the equations for position and steering subsystems, and taking the derivative, we obtain :

$$\dot{z} = \left(1 + \frac{K_\phi}{1+(K_\phi \phi)^2}\right) \frac{v}{r} \sin(z + \arctan(-K_\phi \phi)) + \omega \tag{5.7}$$

Solving for $\boldsymbol{\omega}$, we get the expression for the real control of the robot's steering.

$$\omega = -\frac{v}{r}\left[K_\delta z + \left(1 + \frac{K_\phi}{1+(K_\phi \phi)^2}\right) \sin(z + \arctan(-K_\phi \phi))\right] \tag{5.8}$$

Rearranging and factoring out z, we get $\boldsymbol{\omega}$ in the original coordinates. K_δ controls how quickly the distance and orientation error is reduced.

$$\omega = -\frac{v}{r}\left[K_\delta(\delta - \arctan(-K_\phi \phi)) + \left(\frac{K_\phi}{1+(K_\phi \phi)^2}\right) \sin\delta\right] \tag{5.9}$$

5.4 | Implementation

Using this control law, once the dock pose, P_Dock, has been identified, the set of error variables E_0 are calculated for time zero and plugged into the expression for ω, giving us ω_0. Then at timestep t_0, ω_0 and a low, constant linear velocity v, are applied to the robot's initial pose, P_{R0} to get the projected pose P_{R1}.

The projected pose P_{R1} is then used to calculate the set of error variables E_1, and plugged in to find ω_1. Similarly, ω_1 and v are applied to pose P_{R1} over timestep t_1 to obtain the next projected pose P_{R2}. This cycle continues until the error variables go to zero, which would indicate the robot pose and target pose are the same.

In an ideal world, the variables would go to zero, but because errors due to noise occur in reality, the cycle repeats until the errors are within a specified error tolerance. Once the errors are within tolerance, the sequence of velocities are published as a sequence of Twist messages for the robot to execute.

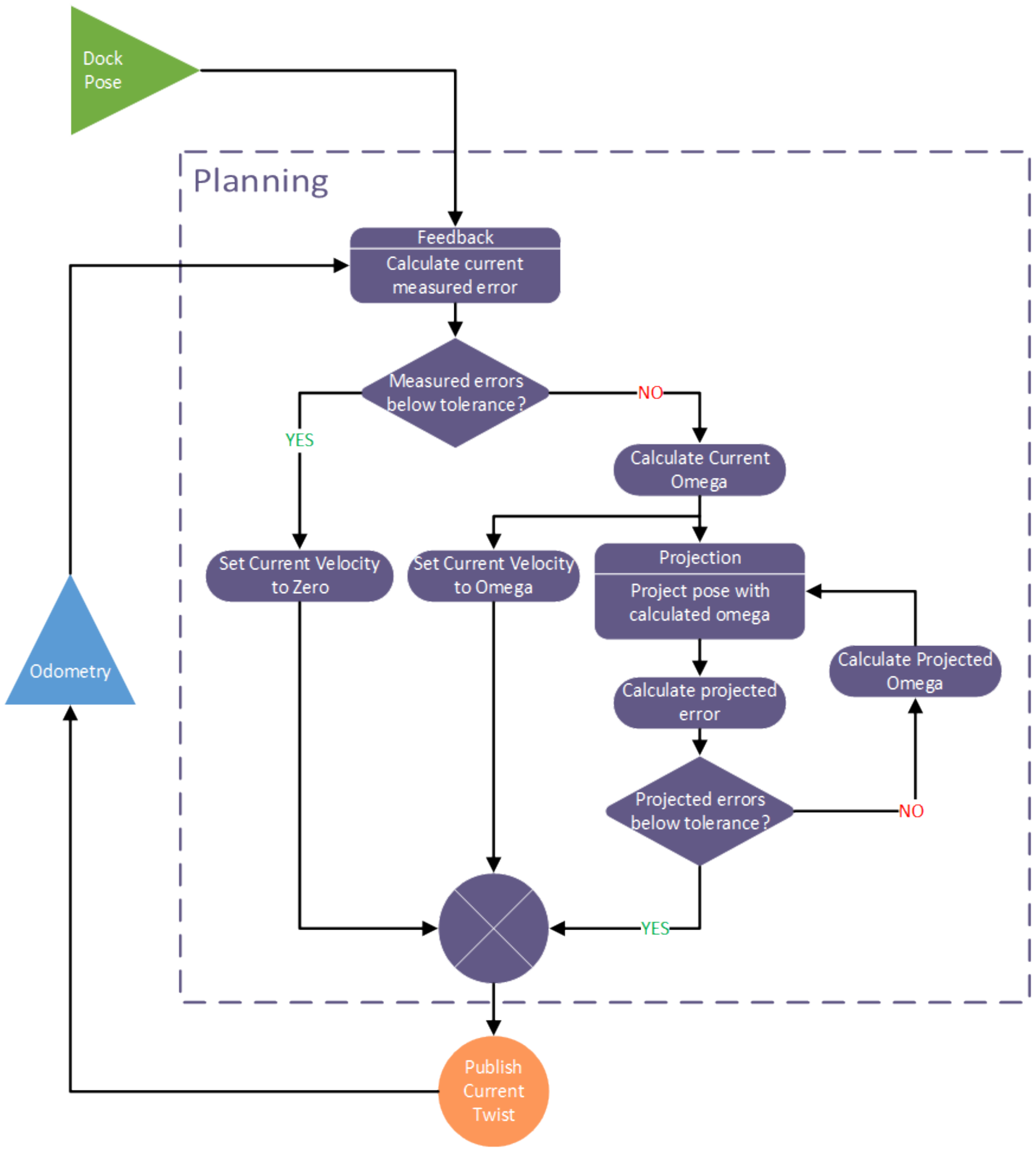
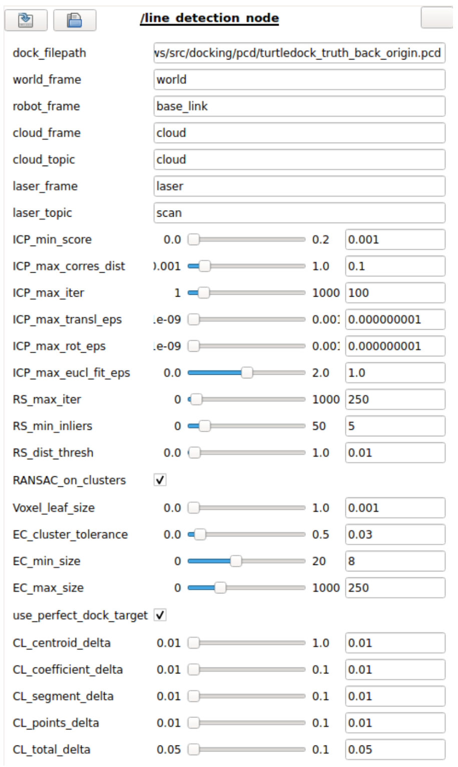

Dock Pose
Planning
Feedback
Calculate current measured error
Measured errors below tolerance?
NO
YES
Calculate Current Omega
Set Current Velocity to Zero
Set Current Velocity to Omega
Projection
Project pose with calculated omega
Calculate Projected Omega
Odometry
Calculate projected error
Projected errors below tolerance?
NO
YES
Publish Current Twist

Figure 5.2: Motion planning process flowchart

6

Software

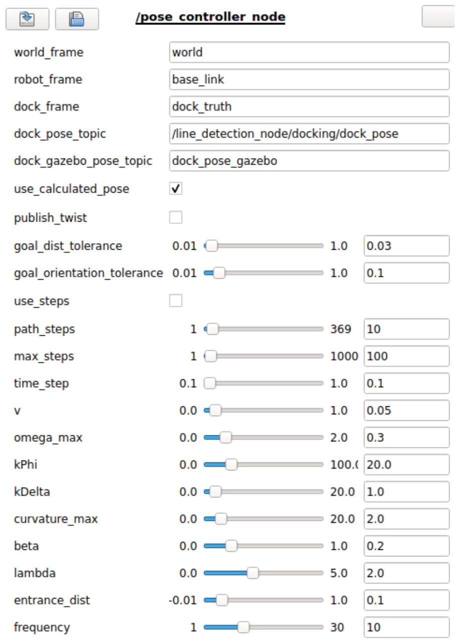
The ROS package for docking consists of two main nodes, `PoseEstimation.cpp`, which implements the perception pipeline, and `Planning.cpp`, which implements the motion planning. For each node, there is a GUI allowing the user to change the various parameters while the node is running. See Figure 6.1. The user can specify whether the GUI should appear when the docking node is launched. Depending on the robot platform, tuning certain parameters are crucial for successful operation, especially for the perception node. Perception parameters should be tuned based on the objects within the dock's vicinity and the capabilities of the LiDAR sensor used. Planning parameters should be tuned based on the dynamics of the robot's motors.



/line_detection_node

dock_filepath			vs/src/docking/pcd/turtledock_truth_back_origin.pcd
world_frame			world
robot_frame			base_link
cloud_frame			cloud
cloud_topic			cloud
laser_frame			laser
laser_topic			scan
ICP_min_score	0.0	0.2	0.001
ICP_max_corres_dist	).001	1.0	0.1
ICP_max_iter	1	1000	100
ICP_max_transl_eps	.e-09	0.00:	0.000000001
ICP_max_rot_eps	.e-09	0.00:	0.000000001
ICP_max_eucl_fit_eps	0.0	2.0	1.0
RS_max_iter	0	1000	250
RS_min_inliers	0	50	5
RS_dist_thresh	0.0	1.0	0.01
RANSAC_on_clusters			✓
Voxel_leaf_size	0.0	1.0	0.001
EC_cluster_tolerance	0.0	0.5	0.03
EC_min_size	0	20	8
EC_max_size	0	1000	250
use_perfect_dock_target			✓
CL_centroid_delta	0.01	1.0	0.01
CL_coefficient_delta	0.01	0.1	0.01
CL_segment_delta	0.01	0.1	0.01
CL_points_delta	0.01	0.1	0.01
CL_total_delta	0.05	0.1	0.05

(a) Perception GUI



/pose_controller_node

world_frame			world
robot_frame			base_link
dock_frame			dock_truth
dock_pose_topic			/line_detection_node/docking/dock_pose
dock_gazebo_pose_topic			dock_pose_gazebo
use_calculated_pose			✓
publish_twist			
goal_dist_tolerance	0.01	1.0	0.03
goal_orientation_tolerance	0.01	1.0	0.1
use_steps			
path_steps	1	369	10
max_steps	1	1000	100
time_step	0.1	1.0	0.1
v	0.0	1.0	0.05
omega_max	0.0	2.0	0.3
kPhi	0.0	100.(	20.0
kDelta	0.0	20.0	1.0
curvature_max	0.0	20.0	2.0
beta	0.0	1.0	0.2
lambda	0.0	5.0	2.0
entrance_dist	-0.01	1.0	0.1
frequency	1	30	10

(b) Motion Planner GUI

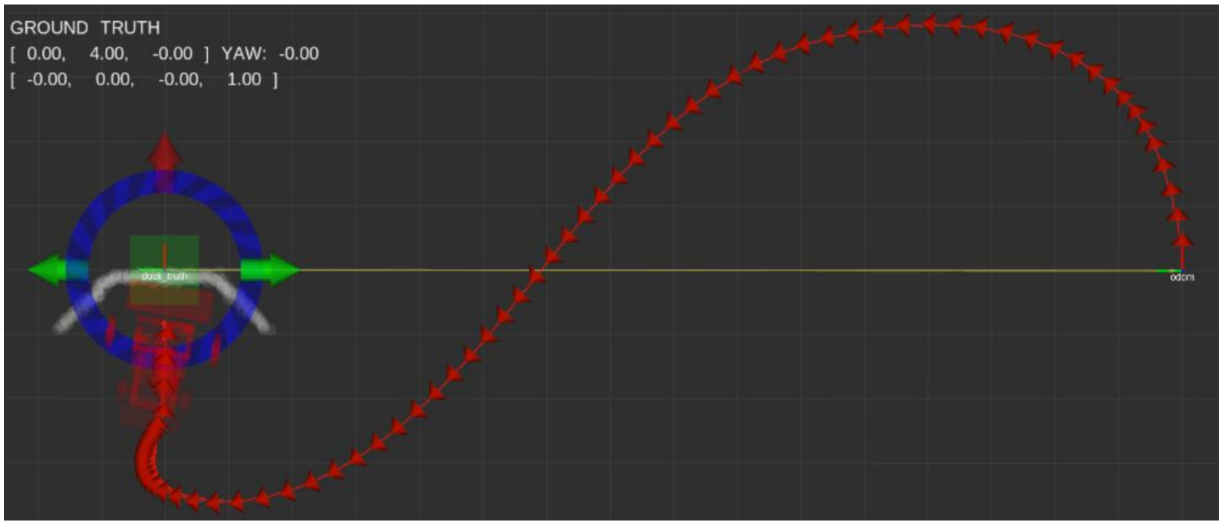
Figure 6.1: Parameter GUIs for perception and motion planning nodes

7 Results

7.1 | Perception pipeline

From a raw LiDAR point cloud, the program is able to use a supplied template of the dock to isolate the points belonging to the dock and extract them into a separate cluster. Once identified, the program is able to successfully estimate the pose of the dock using ICP.

In simulation, pose estimation can be accurate within 1-2cm depending on the specified distance resolution of the LiDAR simulation. Live testing was performed on the Turtlebot's LDS-01 LiDAR which has a distance accuracy of $\pm$ 1.5cm. Live pose estimation achieved accuracy within 5-8cm.



```
GROUND  TRUTH
[  0.00,    4.00,    -0.00  ]  YAW:  -0.00
[  -0.00,    0.00,    -0.00,    1.00  ]
```

Figure 7.1: Simulated Run

7.2 | Motion Planning

With a supplied dock pose, the motion planner can successfully generate smooth trajectories that respect the dynamic constraints of the robot. Using the GUI, the gains can be adjusted to fit the curvature, velocity and acceleration preferences of the user. At the default planning rate of 10Hz, the planner will output successful trajectories. However, the lower the frequency, the lower the update rate, meaning the robot can veer off course longer without the planner readjusting.

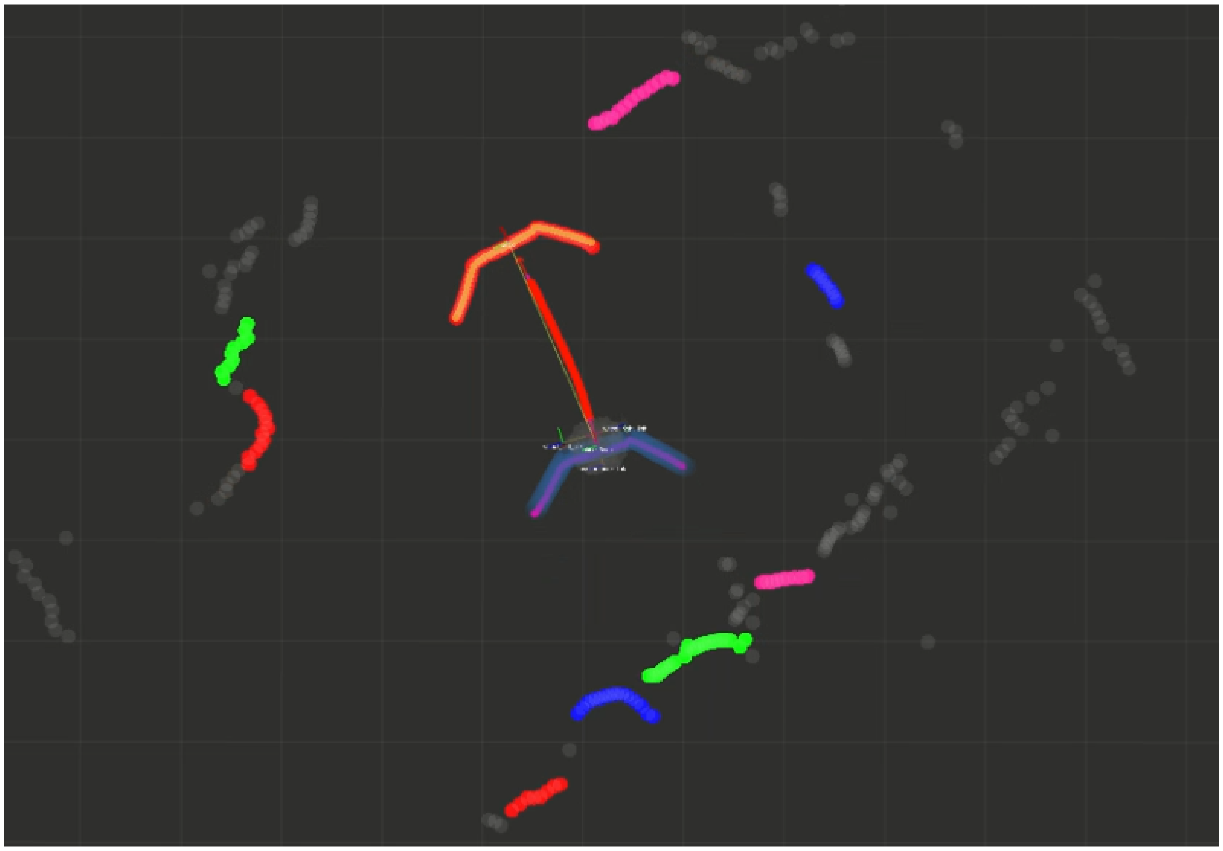


Figure 7.2: Live Run

In Fig 7.1, the robot has successfully docked, and the traversed trajectory is shown in red. The curve present in the path indicates that the control law successfully brings the distance, and more importantly, orientation error to zero.

In Fig 7.2, the robot is halfway through the calculated trajectory, and the minimal curve indicates that the control law brings down the orientation error as quickly as possible. This is a priority because for wheel odometry, linear steps have much less variance than angular steps.

It follows that differential-drive robots especially execute linear velocities more accurately than angular velocities. As such, the motion planner is most accurate when the gain K_ϕ is set appropriately high.

7.3 | Software

A ROS package was successfully created that enables a LiDAR equipped robot to accept a dock template in .PLY point cloud format and perform docking. The package was successfully run on several simulations of popular ROS robots, and live on Turtlebot3, the most widely used ROS robot, thus achieving the goal of platform independence.

8

Conclusions

8.1 | Critique and Limitations

8.1.1 | Dock Model Intake

One of the functionalities for the ROS package was to accept a .STL file of the dock and use it to generate a .PLY point cloud file representing a LiDAR scan of the target face. However, this goal was not able to be completed due to time constraints. Presently, obtaining the dock template is done by a three step process detailed on the documentation page for the package. This was considered a ’reach’ goal as it does not affect the core functionality of the package, but is rather a matter of convenience.

8.1.2 | Incompatible With Nav Stack

The ROS Nav Stack is a collection of packages that support autonomous navigation. The Nav Stack, as shown in Figure: 8.1, is organized into modules and have a set of communication standards so that there are multiple packages that can be chosen from when setting up a module. For example, there is a module for odometry, which needs to publish to a topic `/odom`. There exists a large number of packages capable of doing this, each with a different approach. There are packages that offer wheel odometry, visual odometry, inertial odometry, and there are packages which can fuse many types of odometry to create odometry that is much more accurate.

In its current state, the docking ROS package functions only as a standalone package. Due to time constraints, the package has not yet been modularized into component that can integrate easily with the nav stack. What this means is that the robot can either use the nav stack to navigate, or use the docking package to perform docking, but not both at the same time.

8.1.3 | Prerequisite of Unobstructed View

The perception pipeline works under the assumption that there are no obstacles between the robot and the dock. This is a logical conclusion, since an obstacle between the LiDAR and the dock would prevent the LiDAR from scanning the dock. Similarly, the feedback control law

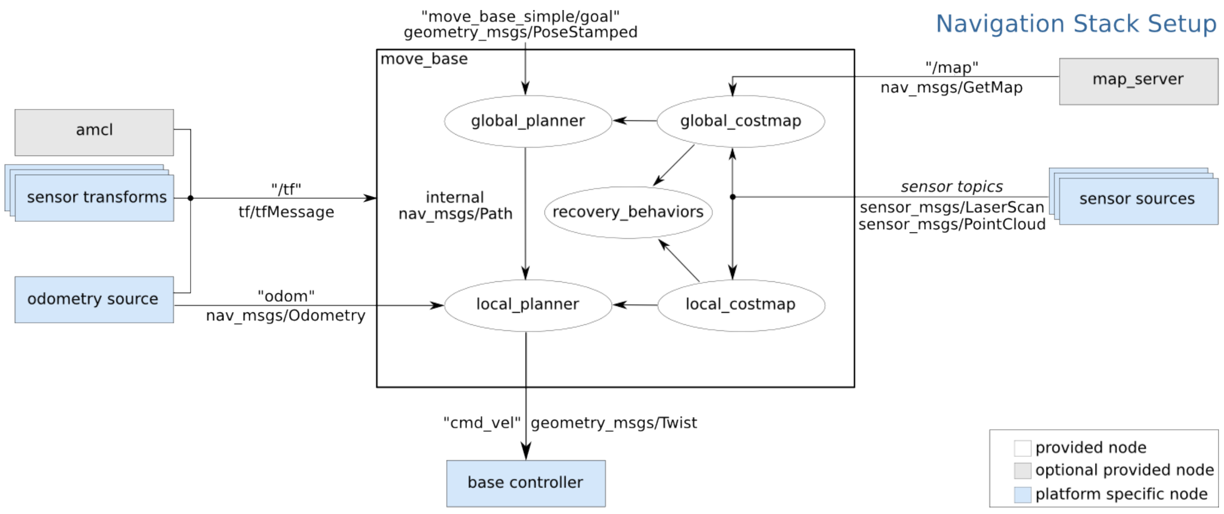


Figure 8.1: ROS Navigation Stack

used by the motion planner also shares this limitation, not only because it depends on an input dock pose from the perception pipeline in order to work, but also because the fundamentally, the control law does not consider obstacles. As such, for the robot to be of practical use, an obstacle avoidance would be required to handle navigation/docking in the presence of static and dynamic obstacles.

8.2 | Future Work

Having achieved functionality, the docking ROS package can be built upon from the limitations outlined previously. Doing so will make the package easily integrable with the nav stack. For more functionality, the motion planner can be restructured to include obstacle avoidance functionality, thus removing the requirement for an obstacle avoidance module from the nav stack. Additionally, for convenience, a GUI could be designed to streamline the dock mode intake process.

8.3 | Final Remarks

Overall, the senior engineering capstone project was a success. All of the core goals were accomplished, as there now exists a ROS package that enables docking functionality on any LiDAR equipped robot. This project widened my range of robotics knowledge, and deepened my understanding of point cloud processing, control systems and C++. As a result, I've become a more capable robotics engineer.

A

Appendix

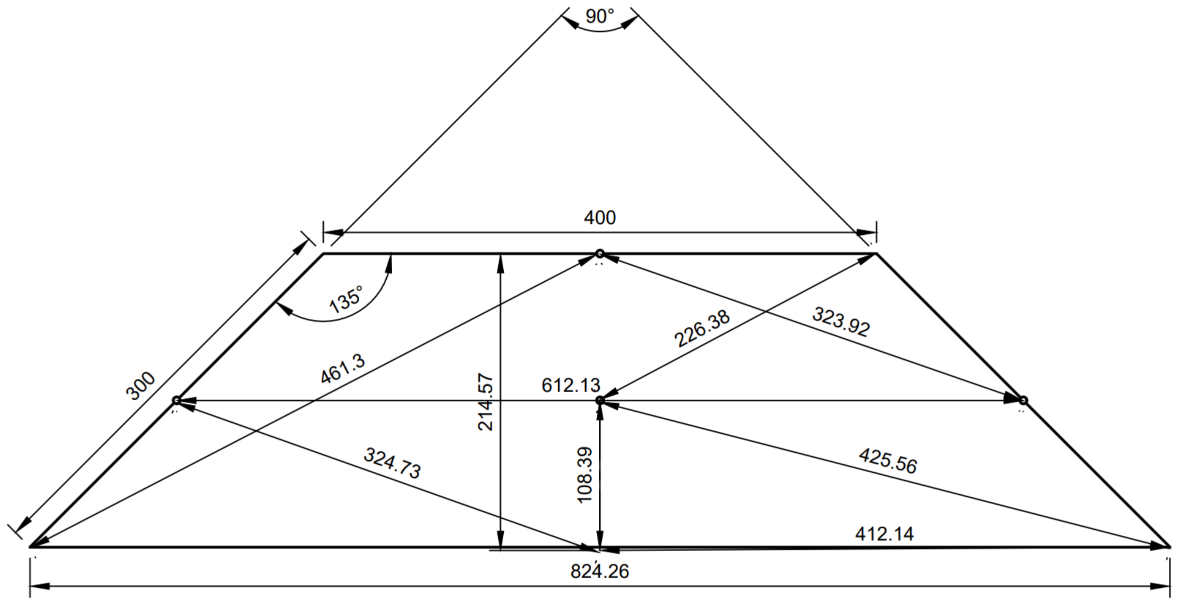


Figure A.1: Geometric features of the test dock